

**Activity 2.1.5 Secure Protocols**

**Introduction**

|  |  |
| --- | --- |
| People send data through the Internet for friendship, business, research, politics, and more. These realms of culture are thousands of years old. Betrayal is every bit as old. However, imposters, spies, and thieves now have global reach like never before. Remember, the Internet is new, with pervasive use beginning in the 1990s. Today all data are vulnerable.  Consider an exchange between Allie and Baba. Allie sends Baba a message, and gets a response from Baba – or so she thinks.  How can Allie know that the reply is from Baba and not some lurker in the Internet?    How can she be sure her message even got to Baba?  Packets on the Internet hop from host to host, taking multiple routes past potentially hostile parties. How can Allie and Baba exchange private information like credit card numbers and be certain that no one eavesdropped on their communication? |  |

Materials

* Computer with Canopy
* ciphers.py and paired\_keys.py

Procedure

1. Form pairs as directed by your teacher. Read the passages below and discuss the questions together.

**Part I: Cryptography and Ciphers**

1. As you learned in a previous activity, the Internet has scaled well because it is decentralized. Packets pass through multiple routes controlled by potentially unfriendly parties. Yet we depend on privacy for many uses of the Internet.

**Cryptography** is the science and art of delivering a message securely and confidentially. The discipline is thousands of years old. An **encryption** algorithm is used to turn **plaintext** into **ciphertext** so that (hopefully) only the recipient can read it. **Decryption** restores the plaintext.

The "s" in the "https" of a URL's scheme indicates that information exchanged in a Web request and response will be encrypted. Give an example of information you might exchange over the web that you would want to be encrypted.

1. Julius Caesar, who took dictatorial control of the Roman Republic in 44 BC, used an encryption algorithm now known as a **substitution cipher**. In a substitution cipher, each letter of the plain text is replaced another letter. Caesar's algorithm shifted each letter by three within the alphabet so that plain text "A" becomes ciphertext "D," and so on, as shown in the table below. The table is an example of a **key**, which is data used by an encryption or decryption function.

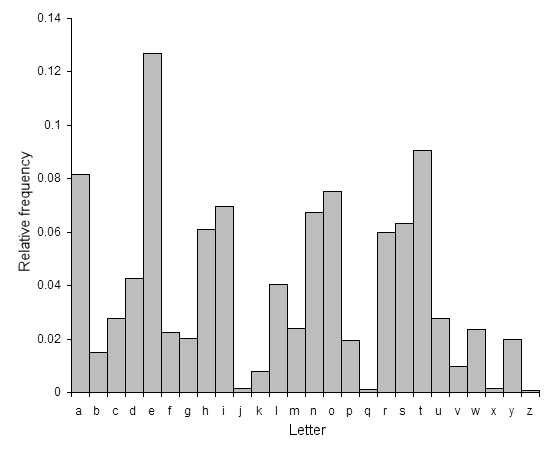
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| plaintext | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| ciphertext | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |

The following message has been encrypted with Caesar's cipher. Decrypt it by shifting each letter three in the opposite direction.

"*Wub wklv*!"

1. Simple ciphers are easily decrypted, even if the key is not known. If the ciphertext is long, the key can be deduced by using letter frequencies; the **relative frequencies** of letters in English are shown below. The relative frequency tells what percent of the time a particular letter is used.

Based on the histogram, which letter is most common if the plaintext is English?



1. Another reason that simple ciphers are easily decrypted is that an individual can try all possibilities—to stumble upon it with certainty by **brute force**. Brute force is trial and error— with computational muscle. Here you will apply brute force to decipher the following ciphertext. The ciphertext was created with a Caesar-like cipher: a rotation of the alphabet, other than 3 letters. Instructions are below.

*Jvgure jrag gerzraqbhf qvabfnhef! Guvaxvat gurer fubhyq rkvfg nggrzcgrq uvynevgl urer?*

1. Open the file ciphers.py in the code editor in Canopy. Your work in this assignment will mostly be in the iPython session. Turn on iPython session logging by setting the working directory and typing %logstart ‑or *filename* at the iPython prompt.
2. Execute the file. This will define the function rotate(). An excerpt showing the def statement and the docstring is shown here.

|  |  |
| --- | --- |
| 5  6  7  8 | **def** rotate(string, n):  '''Returns the ciphertext of string after it has been shifted n letters.  Works on both upper and lower case letters. Other characters are unchanged.  ''' |

1. Use *Python*® programming language to decrypt the ciphertext *"Jvgure ..."* shown above. Use iterations of rotate() across all possible keys until the plaintext is English. This could be done in two ways: complete either Part i or Part ii as follows. Either way, you will need to copy the ciphertext and paste it into Canopy.
   1. From the iPython command line, call rotate() repeatedly.

In []: rotate("Jvgure jrag...", 1)

Out[]: 'Kwhvsf ksbh...'

* 1. In the code editor, complete the for loop in try\_all\_25().

|  |  |
| --- | --- |
| 32  33  34  35  36 | **def** try\_all\_25(string):  """ Use brute force to crack a Caesar-like cipher,  printing all 25 possible shifts"""  **for** shift **in** range(1,26):  **pass** |

1. Explain why you chose Part i or Part ii above.
2. Ciphers are an example of **private key** encryption, where the sender and the recipient have to agree beforehand on the key and then both manage to keep it secret. If either person reveals the key, the cryptographic code is broken. Ciphers are also described as **symmetric key** encryption. The same key is used by the encryption algorithm and the decryption algorithm, albeit in reverse.

One of the most famous ciphers was implemented by the **Enigma** machine. Enigma was used by Germany and Italy during World War II, in which they formed the European front of the Axis powers (Germany, Italy, and Japan) against the Allied powers (the United States, the United Kingdom, and the Soviet Union). Germany transmitted submarine routes encrypted by Enigma.

|  |  |
| --- | --- |
| Alan Turing (pictured at right in statue) automated a method for cracking Enigma, allowing the British to sink many German submarines, making a major contribution to the Allied victory. Computer science's most prestigious award is named in Turing's honor. The Turing Prize is awarded annually. | http://upload.wikimedia.org/wikipedia/commons/thumb/c/c4/AlanTuring-Bletchley.jpg/220px-AlanTuring-Bletchley.jpg |

Roughly half of the work on cryptography is conducted within government agencies, with the NSA leading the effort in the United States. Why has cryptography always been crucial to nations and empires?

**Part II: Public Key Encryption**

1. Internet traffic is rarely encrypted with private key encryption. Explain why it is not practical for a consumer to use private key encryption when buying something through the Internet from a vendor.
2. Most Internet traffic that is encrypted uses the RSA algorithm, discussed in a later activity. RSA is an example of **public key encryption**, which uses two **paired keys** that belong to one owner. The owner of the keys shares one of them publicly and keeps the other secret. The keys are large numbers used by a function. Using either key with the function will transform plaintext into ciphertext. Using the other key will undo the encryption.

This encryption can be used for two different purposes.

* Encrypt a message with someone else's public key. Only they will be able to read the message.
* Encrypt a message with your own private key. Anyone with the message will know the message came from you.

Suppose you get a message from Target, Inc. It looks like gibberish. Target's public key turns it into sensible English. Explain why that should increase your confidence that Target actually sent you that message.

1. You send Target a response that contains your credit card number. You use Target's public key. Explain why that should increase your confidence that only certain Target employees will be able to read your credit card number.
2. To have hands-on practice with these two uses of RSA encryption, generate a pair of RSA keys.
   1. Open paired\_keys.py in the Canopy code editor.
   2. Execute the code to define the functions make\_keys() and use\_key().
   3. In the iPython session, make two keys akey and bkey as shown below.

|  |  |
| --- | --- |
| In []: In []: Out[]: In []: Out[]: | akey, bkey = make\_keys()  print akey  (23707, 3343)  print bkey  (23707, 7) |

* 1. Each of the keys contains two numbers. The **modulus** is the number that is the same in the two keys. In the example output shown above, 23707 is the modulus. Repeat Step c until your key’s modulus is at least 10000.
  2. Publish one of your keys for the class to use. This will be your public key. For everyone's convenience, use the smaller key as the public key. You can use a class-wide **Google document** to publish your keys. Alternatively, write your public key and your name on a notecard and place your notecard on a bulletin board for people to use.

1. Follow the steps below to send a message that only the recipient will be able to decrypt. Include your name and at least one other word in the message.
   1. Turn your secret message into a numeric message. As shown below, use the numerize() function that was defined when you executed paired\_keys.py. This is a substitution cipher that replaces each character with a 2-digit number. numerize() returns the message as ciphertext. The ciphertext is a sequence of numbers smaller than the RSA key's modulus.

|  |  |
| --- | --- |
| In []: In []: Out[]: | message = numerize("Hey Jon, see you after school? -Ted")  message  '0646-7595-0648-8584-1806-8975-7506-9585-9106-7176-9075-8806-8973-7885-8582-3706-1958-7574' |

* 1. Use the recipient's public key as shown below to encrypt the numeric message. Use the use\_key() function that was defined when you executed paired\_keys.py. The example below uses Jon's public key.

|  |  |
| --- | --- |
| In []: In []: In []: Out[]: | jons\_key = (20453, 61) *# Use recipient's public key*  encrypted\_message = use\_key(jons\_key, message)  encrypted\_message  '14139-16645-3362-9259-18839-14535-16569-5603-13993-3556-2559-15257-4497-6163-10262-2801-6742-12413' |

* 1. Pass the encrypted message to its recipient. Use email, Skype, a class-wide Google document, pencil and paper, or any other method as directed by your teacher.
  2. Assign the message that you have received to encrypted\_message, as shown below. Decrypt the message using your private key. In the example below, Jon (the recipient) has typed the message he received at the iPython prompt to assign encrypted\_message and used his private key.

|  |  |
| --- | --- |
| In []:  In []: Out[]: In []: In []: Out[]:  In []: In []: Out[]: | encrypted\_message = '14139-16645-3362-9259-18839-14535-16569-5603-13993-3556-2559-15257-4497-6163-10262-2801-6742-12413'  bkey *# Jon's private key; you might have kept akey as private*  (20453, 661)  decrypted\_message = use\_key(bkey, encrypted\_message)  decrypted\_message  '0646-7595-0648-8584-1806-8975-7506-9585-9106-7176-9075-8806-8973-7885-8582-3706-1958-7574'  plain\_message = denumerize(decrypted\_message)  plain\_message  'Hey Jon, see you after school? -Ted' |

1. Follow the steps below to publicize a message that only you could have sent. Other people will use your public key to decrypt your message. That will authenticate that the message came from you and not from someone else.
   1. Enter your message at the iPython prompt. Use the substitution cipher numerize() to make it a numeric message. Encrypt your message using your private key.

|  |  |
| --- | --- |
| In []: In []: Out[]:  In []: Out[]: | message = "I make this payment -JSmith."  The underscore character  \_  is a special variable in the iPython session. It contains the value returned by the previous command.  numerize(message)  '4706-7183-0690-7875-0685-8475-68  7989-0689-7879-8890'  use\_key(my\_private\_key, \_ )  '15790-19762-15967-10221-16029-82  52-16029-28499-25391-14919' |

* 1. Publish your message. The whole class should publish everyone's messages in a single Google document. A bulletin board is an alternative.
  2. Select one or more of the published statements as directed by your teacher. Enter that person's published message at the iPython prompt. Decrypt it with their public key, as shown below.

|  |  |
| --- | --- |
| In []:  In []: In []: Out[]: In []: Out[]: | authenticated\_message = '15790-19762-15967-10221-16029-82  52-16029-28499-25391-14919'  jons\_public\_key = (20453, 61)  use\_key(jons\_public\_key, authenticated\_message)  '0646-7595-0648-8584-1806-8975-7506-9585-9106-7176-9075-8806'  plain\_message = denumerize(\_)  'I make this payment -JSmith' |

Record the message here.

* 1. Explain why using the sender's public key to read the message increased your confidence that the message was really from them.
  2. Why doesn't the use of the sender's public key guarantee that the message was from them?

**Part III: Authentication and Trusted Authorities**

1. A public key is an important part of an **SSL certificate**. SSL certificates are encrypted messages that confirm the key owner's identity and provide their public key. In the previous example, you assumed that the public key came from the person who said it was theirs. They could have been an imposter, of course. When you access a URL that begins with https, your browser downloads an SSL certificate from that website. Your browser will then use the public key contained in the certificate to send your confidential information. But what are the implications for you if an imposter makes a pair of keys and gives you one, tricking you into thinking it is the website's public key?
2. A Web browser **authenticates** that a public key really comes from the owners of the domain name that you are accessing and not from an imposter. When a company gets an SSL certificate, they apply to a **Certificate Authority** (like Symantec) to sign their certificate. The certificate authority uses their own private key to encrypt the applicant's public key and the applicant's domain name. The certificate authority's public key is built into the browser; if that public key can decrypt the certificate, it must be from the certificate authority. The decrypted certificate contains the domain name and public key of the owner of the certificate. If it matches the domain name you are trying to reach, you know they are who they say they are.

To simulate the way the SSL relies on a handful of trusted authorities, use Python and trust authentication to get a message from an untrusted sender, as follows.

* 1. Select a small number of people in your class to trust, as directed by your teacher. Who are the trusted authorities in your class?
  2. As a simulation, apply for an SSL certificate from one of the trusted authorities. Do this by giving them your public key and your name. This can be done with a Google document.
  3. The trusted authority will confirm with you that the public key is in fact yours.
  4. The trusted authority will encrypt a message containing your public key and your name, using their private key. An example is shown below.

|  |  |
| --- | --- |
| In []: In []: Out[]: In []: Out[]: | applicant = '(20453, 61) Jon Smith' *# Applicant's key and name*  numerize(applicant)  '0646-7595-0648-8584-1806-8975-7506-9585-9106-7176-9075-8806'  use\_key(authoritys\_private\_key, \_ )*# Returns the certificate*  '10252-15596-1552-3433-2078-16302-16914-26648-10542-14912' |

* 1. The trusted authority will return the result to you, the applicant. This is your SSL certificate. It has been signed by the trusted authority. You can create a new column in the same Google document for each student's certificate.
  2. Select an "untrusted" partner as directed by your teacher. Obtain your partner's public key, as follows.
     1. Obtain your untrusted partner's certificate.
     2. Use the trusted authority's public key to obtain your untrusted partner's public key, as shown below.

|  |  |
| --- | --- |
| In []: In []: In []: Out[]: | certificate = '10252-15596-1552-3433-2078-16302-16914-26648'  CAs\_public\_key = (44719, 11) *# from certificate authority*  use\_key(CAs\_public\_key, certificate)  '(20453, 61) Jon Smith' |

* 1. Exchange messages with the untrusted partner using each other's public keys, as directed in Step 11 above. Record your untrusted partner's plaintext message here.
  2. Compare the use of public keys in Part d of Step 11 with the use of signed public keys in Part g of Step 14. Explain why the involvement of the Certificate Authority in Step 14 should increase your confidence that your message in Step 14 was only readable to its intended recipient.

1. Save your iPython session log by typing %logstop at the iPython prompt.

**Conclusion**

1. Explain the relationship among the following concepts.

* The https:// designation in your browser
* Public key encryption
* SSL certificate
* Certificate Authority
* Domain name

1. You typed https://www.google.com in the address bar in your browser and received the notice shown below. Choose "Get me out of here!" or "Add Exception."

Explain why you made the choice you did.

