[9144965]

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

Words: 1010

[Cryptography]

[The implementation of cryptography to protect the Zambeesi webstore]



Contents

[Figures 1](#_Toc59116515)

[Introduction 2](#_Toc59116516)

[X.509 Certificate 2](#_Toc59116517)

[Diffie-Hellman Key Exchange 6](#_Toc59116518)

[AES-256-GCM 7](#_Toc59116519)

[Conclusion 11](#_Toc59116520)

[References 11](#_Toc59116521)

[Appendix 12](#_Toc59116522)

[Appendix 1: Creating the RSA key With Python 12](#_Toc59116523)

[Appendix 2: Generating the X.509 certificate With Python 13](#_Toc59116524)

[Appendix 4: Diffie-Hellman Key Exchange Python Code 14](#_Toc59116525)

[Appendix 5: AES-256-GCM Python code 15](#_Toc59116526)

# Figures

[Figure 1: Creating a 2048bit RSA key. 3](https://livecoventryac-my.sharepoint.com/personal/lovells3_cuscuni_coventry_ac_uk/Documents/Y2B2/Assignment%202/Y2B2A2%20-%20final.docx#_Toc59116527)

[Figure 2: Using Python to create an X.509 certificate and signing it with the previously generated RSA key. 4](https://livecoventryac-my.sharepoint.com/personal/lovells3_cuscuni_coventry_ac_uk/Documents/Y2B2/Assignment%202/Y2B2A2%20-%20final.docx#_Toc59116528)

[Figure 3: Shows the save location of the certificate, private key and the public key. 5](#_Toc59116529)

[Figure 4: Using PowerShell to install the certificate 5](#_Toc59116530)

[Figure 5: The certificate opened with Windows certificate manager 6](#_Toc59116531)

[Figure 6: Diffie-Hellman key exchange in python. 7](#_Toc59116532)

[Figure 7: Python code creating and encrypting file using AES-256-GCM. Part 1 of 2 9](https://livecoventryac-my.sharepoint.com/personal/lovells3_cuscuni_coventry_ac_uk/Documents/Y2B2/Assignment%202/Y2B2A2%20-%20final.docx#_Toc59116533)

[Figure 8: Python code checking the integrity of the files using SHA-256. Part 2 of 2 10](#_Toc59116534)

[Figure 9: Using Python to generate a HMAC code 10](#_Toc59116535)

# Introduction

When choosing suitable cryptographic methods to implement, it is important to first understand the necessary security requirements. This report will show which cryptographic methods were chosen to help protect the Zambesi webstore, why they were chosen and how they meet the security requirements for the Zambesi web store. The core security requirement is protecting Zambesi’s customers. Zambesi customers save their banking information, name and other sensitive data to the Zambesi store with the expectation that it is safe and secure. Zambesi has a social responsibility to ensure users data is secure and protected from unauthorised users. That is why Transport Layer Security (TLS) was chosen to create end to end encryption between Zambesi’s and its customers.

# X.509 Certificate

In order to use TLS on the Zambesi website, an X.509 certificate must be created. One of the requirements for an X.509 certificate is a digital signature. The digital signature will be created using Rivest, Shamir, & Adleman (RSA) to create a 2048-bit asymmetric key pair.

RSA was picked over Digital Signature Algorithm (DSA) due to it being faster at validating the certificate and encrypting versus DSA being faster at generating the key and decrypting. You only need to create the signature once whereas every end user will have to verify the signature. Another concern with DSA is there have been vulnerabilities which have allowed attackers to decrypt the private key from the signature (Lawson, 2010).

This key will be used to sign the X.509 certificate, this is a method of authentication. When users connect to the Zambesi webstore, the first response from Zambesi will be to send the Zambesi certificate and the Zambesi public-key to authenticate the webstore. Now Zambesi users can compare the digital signature on the certificate to that of Zambesi.

There are two ways forward regarding certificates. They can either be self-signed as in signed by Zambesi or signed by a certificate authority (CA). A CA offers an increased level of trust by vetting the organisation and guaranteeing they are a legitimate business. Another advantage of using a CA is auto-renewal. TLS certificates have a lifespan of 398 days before needing to be renewed (CA/Browser Forum, 2020). DigiCert is a CA which offers automated renewal and increased control over the certificate’s lifecycle. This removes one of the major complexities of using certificates and will cut down on certificate maintenance (DigiCert, 2020).

The next step of TLS is to create a symmetric key for end to end encryption using Diffie-Hellman key exchange (DHE).

Text

Description automatically generated

Figure 1: Creating a 2048bit RSA key.

Text

Description automatically generated

Figure 2: Using Python to create an X.509 certificate and signing it with the previously generated RSA key.

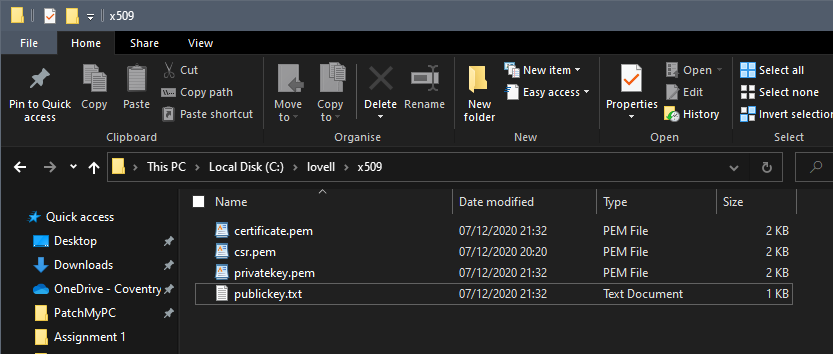


Figure 3: Shows the save location of the certificate, private key and the public key.

Graphical user interface, text, application

Description automatically generated

Figure 4: Using PowerShell to install the certificate

Graphical user interface, text, application

Description automatically generated

Figure 5: The certificate opened with Windows certificate manager

# Diffie-Hellman Key Exchange

In order to create end to end encryption, a symmetric key must be created and shared between Zambesi and the end user. DHE enables the creation of a secret symmetric key by public means using modular arithmetic.

Modular arithmetic is one of the mathematical cores of cryptography. In modular arithmetic numbers wrap around when reaching the modulus. For example, a 12-hour clock uses modulo 12. This would mean 15 modulo 12 would be 3. That is why on a 24hour clock 15:00 is 3 pm. Modular arithmetic is fast to work out forwards but time consuming to work out backwords. This makes it ideal for cryptography (Menezes et al., 1997, p. 600). Figure 6 contains a more in-depth example of how DH uses modular arithmetic.

Once the exchange has finished end to end encryption is enabled using the shared key as a symmetric key. This allows the connection to be encrypted using AES-256-Galois/Counter Mode (GCM).

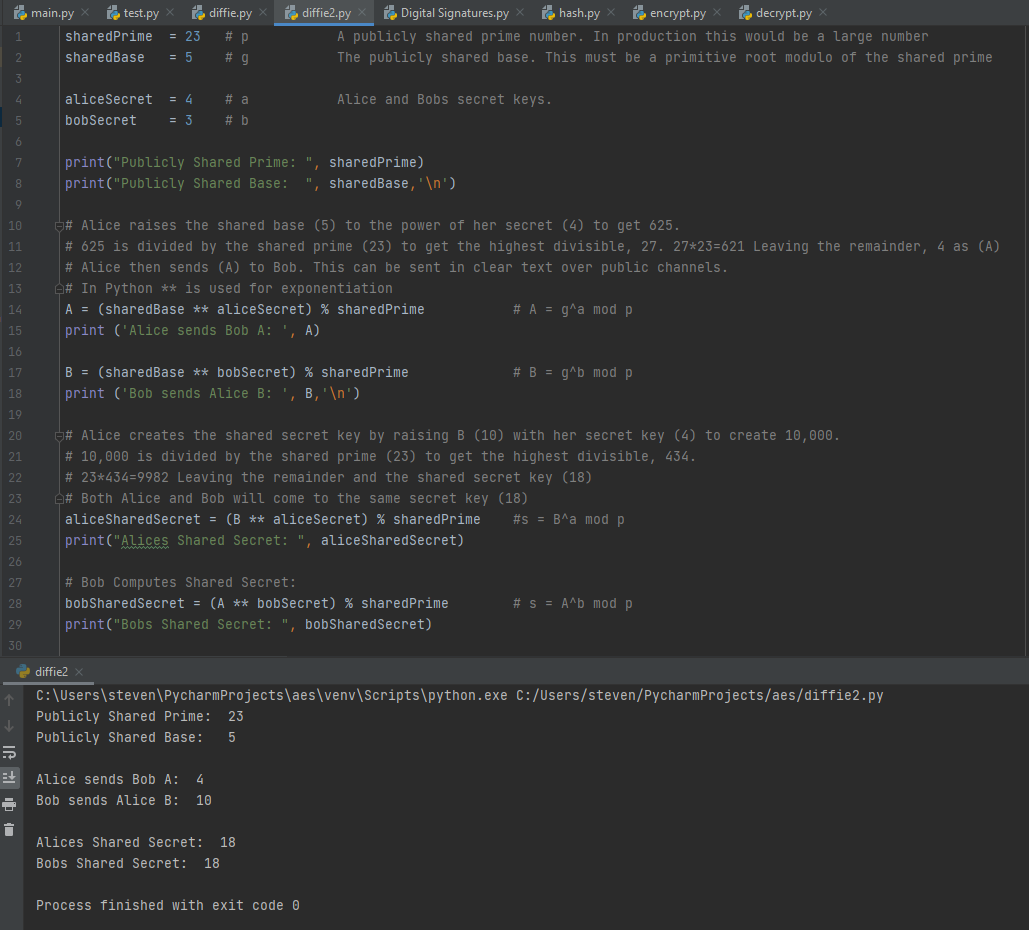


Figure 6: Diffie-Hellman key exchange in python.

# AES-256-GCM

AES is a symmetric encryption system designed to encrypt data. A downside to symmetric encryption is that it ensures confidentiality but not integrity or authentication of the data. The ciphertext could be tampered with in transit and the recipient would not know. However, a modern trend in cryptography is to mix and match different methods of cryptography. For example, GCM uses the counter mode of operation which converts the AES block cipher into a stream cipher and uses a random value along with an incrementing counter to encrypt each block.

GCM can be used with AES to provide encryption, authenticity and integrity, this is known as authenticated encryption with associated data (AEAD).

AEAD sends two pieces of data, one is the encrypted data and the second is a tag, also known as a Message Authentication Code (MAC). The MAC hashes the ciphertext and the nonce and appends this to the encrypted data. This is referred to as Encrypt-then-MAC and makes the message tamperproof. AES-GCM allows each block to be encrypted and decrypted in parallel. This makes AES-GCM not only secure but fast, hence why it was chosen for the Zambesi webstore (Stallings, 2014, p 376).

The alternative to AES-GCM is ChaCha20-Poly1305. Both are AEAD and the two best options for TLS 1.3. When picking one it is important to understand how Zambesi customers will be accessing the webstore. Whether they are using their phone or desktop computers. If most Zambesi customers access the webstore with their phone, then ChaCha20-Poly1305 is the better option as it is designed to run faster on older Android phones. Whereas most modern CPUs will have an AES chip built into them, this makes AES faster on desktop computers. (Sullivan, 2015). ‘The 2018 Amazon Shopper behavior study’ showed that while more people browsed Amazon on their phone, they still made more purchases from their desktop computers (cpcstrategy 2018). Therefore AES-GCM was the preferred choice for the Zambesi webstore.

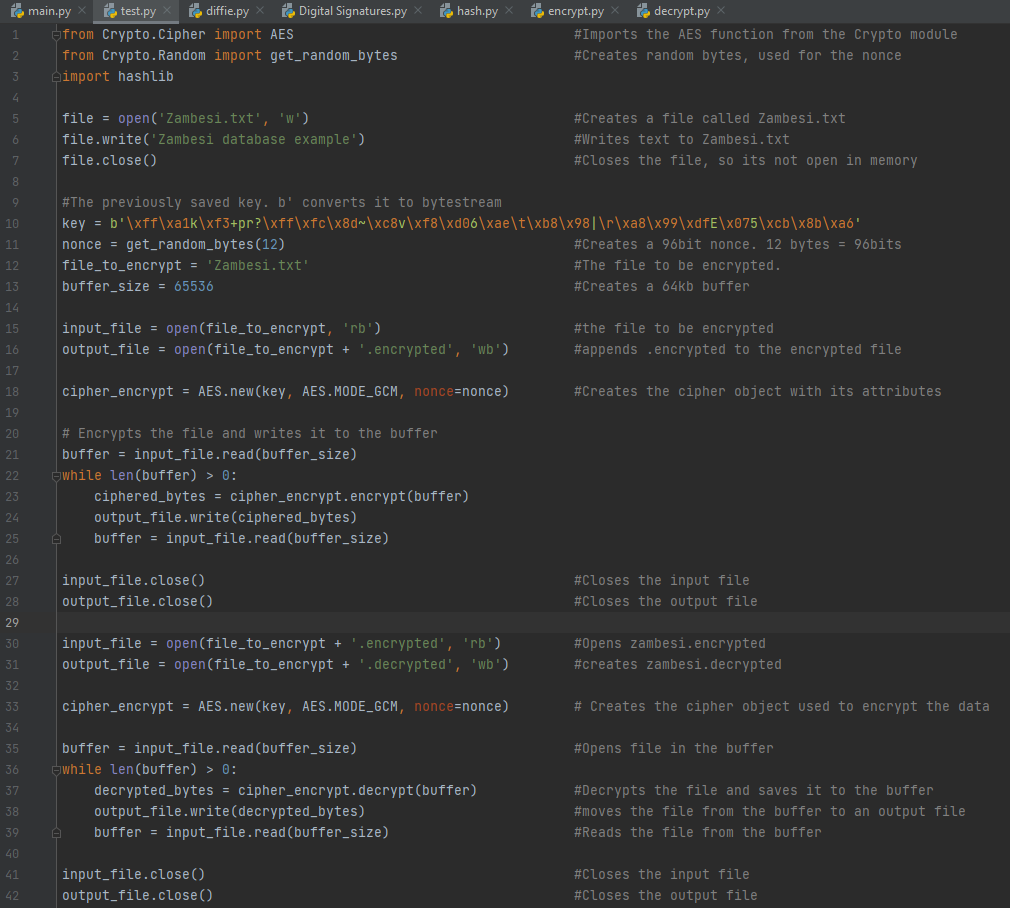


Figure 7: Python code creating and encrypting file using AES-256-GCM. Part 1 of 2

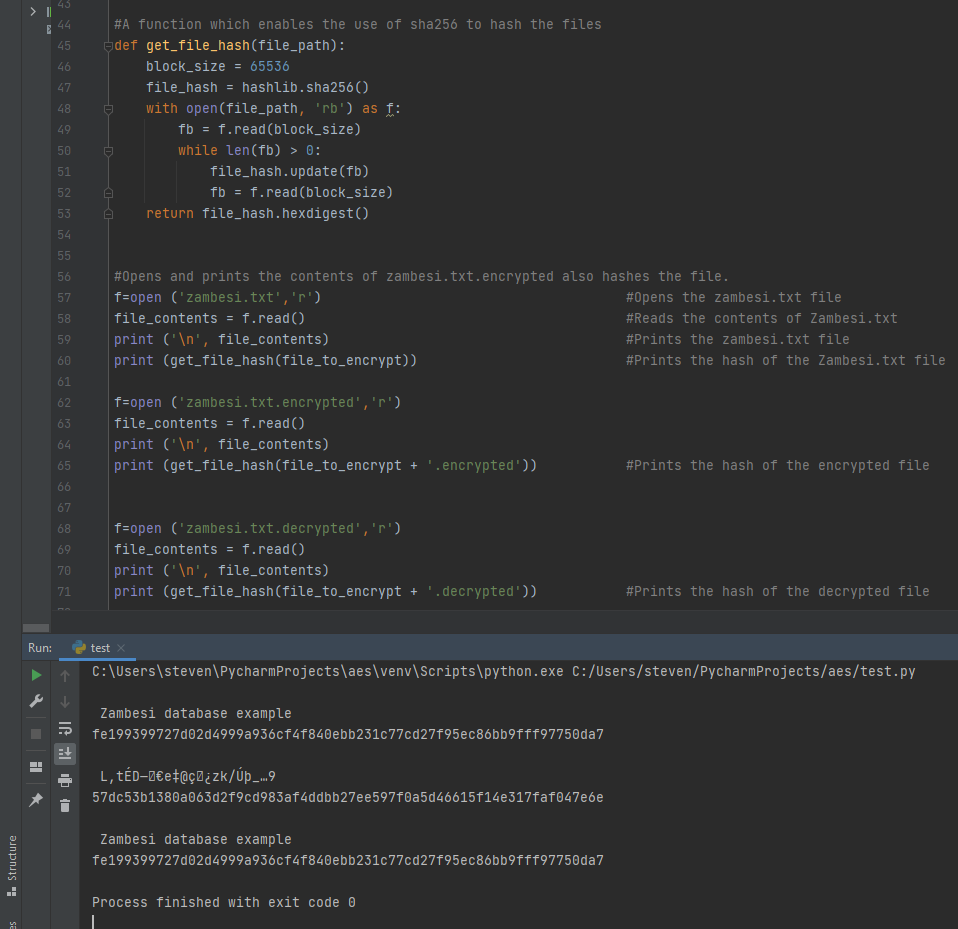


Figure 8: Python code checking the integrity of the files using SHA-256. Part 2 of 2

Text

Description automatically generated

Figure 9: Using Python to generate a HMAC code

# Conclusion

This report has shown how each part of TLS comes together to create a secure method for protecting both Zambesi and the end user. X.509 certificates are the primary certificates used with TLS. It is recommended to use a CA to streamline certificate renewal. This will give the end user confidence in using the Zambesi webstore. DHE is the only method of public-key cryptography considered to be secure enough to use in TLS 1.3 (Rescorla, 2018). Of the two methods available in TLS 1.3 for generating secure signatures, RSA was chosen due to the speed advantage for the end users and security reasons. AES-256-GCM was chosen due to it being the faster option for the majority of Zambesi’s paying customers. All these cryptographic methods come together to form TLS 1.3 and help to create a secure and trusting relationship between Zambesi and its customers.

# References

CA/Browser Forum (2020). *Baseline Requirements for the Issuance and Management of Publicly-Trusted CertificatesVersion 1.7.3.* https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.7.3.pdf

cpcstrategy (2018). *The 2018 Amazon Shopper Behavior Study.* <http://learn.cpcstrategy.com/rs/006-GWW-889/images/2018-Amazon-Shopper-Behavior-Study.pdf>

DigiCert (2020). *Simplify. Customize. Automate. Repeat.* https://www.digicert.com/certificate-management/

Lawson, N. (2010). *DSA requirements for random k value.* Rdist. <https://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/>

Legrandin (2014) *A self-contained cryptographic library for Python.* https://github.com/Legrandin/pycryptodome

Menezes, A., Vanstone, S. and Van Oorschot, P. (1997). *Handbook Of Applied Cryptology (1st ed.)*. [Place of publication not identified]: CRC

Percival, M., Reynolds, N., Kislyuk, A. (2011). *Python One-Time Password Library.* [*https://github.com/pyauth/pyotp*](https://github.com/pyauth/pyotp)

Python Cryptographic Authority. (2013) *cryptography is a package designed to expose cryptographic primitives and recipes to Python developers.* [*https://github.com/pyca/cryptography*](https://github.com/pyca/cryptography)

Rescorla, E. (2018). Rfc 8446: The transport layer security (tls) protocol version 1.3. Internet Engineering Task Force (IETF).

Stallings, W. (2014). *Cryptography And Network Security.* (6th ed.). Boston: Pearson

Sullivan, S. (2015). *Do the ChaCha: better mobile performance with cryptography*. Cloudflare. <https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/>

# Appendix

# Appendix 1: Creating the RSA key With Python

#This imports the cryptography module  
from cryptography.hazmat.primitives import serialization #Serialisation converts text into bytestream  
from cryptography.hazmat.primitives.asymmetric import rsa #Imports RSA, this is used to generate a private key  
  
# Generates an RSA key  
key = rsa.generate\_private\_key(  
 public\_exponent=65537,  
 key\_size=2048, #The Key size will be 2048 bits.  
)  
# Writes the key to disk  
with open("c:/lovell/x509/key.pem", "wb") as f: #The save location  
 f.write(key.private\_bytes(  
 encoding=serialization.Encoding.PEM, #Will encapsulate the key using Privacy Enhanced Mail  
 format=serialization.PrivateFormat.TraditionalOpenSSL, #Will use the PKCS#1 to format the key  
 encryption\_algorithm=serialization.BestAvailableEncryption(b"steven.lovell"), #Converts to bytestream  
 ))

# Appendix 2: Generating the X.509 certificate With Python

#This imports the cryptography module and requisite functions  
from cryptography.hazmat.primitives import serialization #Serialisation converts text into bytestream  
from cryptography.hazmat.primitives.asymmetric import rsa #Imports RSA, this is used to generate a private key  
from cryptography import x509  
from cryptography.x509.OID import NameOID  
from cryptography.hazmat.primitives import hashes  
import datetime  
  
# Generates an RSA key  
key = rsa.generate\_private\_key(  
 public\_exponent=65537,  
 key\_size=2048, #The Key size will be 2048 bits.  
)  
  
# Writes the key to disk  
with open("c:/lovell/x509/privatekey.pem", "wb") as f: #The save location  
 f.write(key.private\_bytes(  
 encoding=serialization.Encoding.PEM, #Will encapsulate the key using Privacy Enhanced Mail  
 format=serialization.PrivateFormat.TraditionalOpenSSL, #Will use the PKCS#1 to format the key  
 encryption\_algorithm=serialization.BestAvailableEncryption(b"steven.lovell"), #Converts to bytestream  
 ))  
with open("c:/lovell/x509/publickey.txt", "wb") as f:  
 f.write(key.public\_key().public\_bytes( #Saves the public key  
 encoding=serialization.Encoding.OpenSSH, #Encodes and formats the public key using OpenSSH  
 format=serialization.PublicFormat.OpenSSH  
 ))  
  
# A self-signed certificate for the Zambesi webstore  
# These are the attributes that will show up on the X509 certificate  
subject = issuer = x509.Name([  
 x509.NameAttribute(NameOID.COUNTRY\_NAME, u"GB"),  
 x509.NameAttribute(NameOID.STATE\_OR\_PROVINCE\_NAME, u"England"),  
 x509.NameAttribute(NameOID.LOCALITY\_NAME, u"Scarborough"),  
 x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"Zambesi"),  
 x509.NameAttribute(NameOID.COMMON\_NAME, u"Zambesi.com"),  
])  
cert = x509.CertificateBuilder().subject\_name(  
 subject  
).issuer\_name(  
 issuer  
).public\_key(  
 key.public\_key()  
).serial\_number(  
 x509.random\_serial\_number()  
).not\_valid\_before(  
 datetime.datetime.utcnow()  
).not\_valid\_after(  
 datetime.datetime.utcnow() + datetime.timedelta(days=10) #The certificate will be valid for 10 days.  
).add\_extension(  
 x509.SubjectAlternativeName([x509.DNSName(u"localhost")]), #Creates an alternative DNS name  
 critical=False,  
  
).sign(key, hashes.SHA256()) #Signs the certificate with a key  
  
with open("c:/lovell/x509/certificate.pem", "wb") as f: #Saves the certificate to disk.  
 f.write(cert.public\_bytes(serialization.Encoding.PEM)) #Encodes the file using PEM

# Appendix 4: Diffie-Hellman Key Exchange Python Code

sharedPrime = 23 # p A publicly shared prime number. In production this would be a large number  
sharedBase = 5 # g The publicly shared base. This must be a primitive root modulo of the shared prime  
  
aliceSecret = 4 # a Alice and Bobs secret keys.  
bobSecret = 3 # b  
  
print("Publicly Shared Prime: ", sharedPrime)  
print("Publicly Shared Base: ", sharedBase,'\n')  
  
# Alice raises the shared base (5) to the power of her secret (4) to get 625.  
# 625 is divided by the shared prime (23) to get the highest divisible, 27. 27\*23=621 Leaving the remainder, 4 as (A)  
# Alice then sends (A) to Bob. This can be sent in clear text over public channels.  
# In Python \*\* is used for exponentiation  
A = (sharedBase \*\* aliceSecret) % sharedPrime # A = g^a mod p  
print ('Alice sends Bob A: ', A)  
  
B = (sharedBase \*\* bobSecret) % sharedPrime # B = g^b mod p  
print ('Bob sends Alice B: ', B,'\n')  
  
# Alice creates the shared secret key by raising B (10) with her secret key (4) to create 10,000.  
# 10,000 is divided by the shared prime (23) to get the highest divisible, 434.  
# 23\*434=9982 Leaving the remainder and the shared secret key (18)  
# Both Alice and Bob will come to the same secret key (18)  
aliceSharedSecret = (B \*\* aliceSecret) % sharedPrime #s = B^a mod p  
print("Alices Shared Secret: ", aliceSharedSecret)  
  
# Bob Computes Shared Secret:  
bobSharedSecret = (A \*\* bobSecret) % sharedPrime # s = A^b mod p  
print("Bobs Shared Secret: ", bobSharedSecret)

# Appendix 5: AES-256-GCM Python code

from Crypto.Cipher import AES #Imports the AES function from the Crypto module  
from Crypto.Random import get\_random\_bytes #Creates random bytes, used for the nonce  
import hashlib  
  
file = open('Zambesi.txt', 'w') #Creates a file called Zambesi.txt  
file.write('Zambesi database example') #Writes text to Zambesi.txt  
file.close() #Closes the file, so its not open in memory  
  
#The previously saved key. b' converts it to bytestream  
key = b'\xff\xa1k\xf3+pr?\xff\xfc\x8d~\xc8v\xf8\xd06\xae\t\xb8\x98|\r\xa8\x99\xdfE\x075\xcb\x8b\xa6'  
nonce = get\_random\_bytes(12) #Creates a 96bit nonce. 12 bytes = 96bits  
file\_to\_encrypt = 'Zambesi.txt' #The file to be encrypted.  
buffer\_size = 65536 #Creates a 64kb buffer  
  
input\_file = open(file\_to\_encrypt, 'rb') #the file to be encrypted  
output\_file = open(file\_to\_encrypt + '.encrypted', 'wb') #appends .encrypted to the encrypted file  
  
cipher\_encrypt = AES.new(key, AES.MODE\_GCM, nonce=nonce) #Creates the cipher object with its attributes  
  
# Encrypts the file and writes it to the buffer  
buffer = input\_file.read(buffer\_size)  
while len(buffer) > 0:  
 ciphered\_bytes = cipher\_encrypt.encrypt(buffer)  
 output\_file.write(ciphered\_bytes)  
 buffer = input\_file.read(buffer\_size)  
  
input\_file.close() #Closes the input file  
output\_file.close() #Closes the output file  
  
input\_file = open(file\_to\_encrypt + '.encrypted', 'rb') #Opens zambesi.encrypted  
output\_file = open(file\_to\_encrypt + '.decrypted', 'wb') #creates zambesi.decrypted  
  
cipher\_encrypt = AES.new(key, AES.MODE\_GCM, nonce=nonce) # Creates the cipher object used to encrypt the data  
  
buffer = input\_file.read(buffer\_size) #Opens file in the buffer  
while len(buffer) > 0:  
 decrypted\_bytes = cipher\_encrypt.decrypt(buffer) #Decrypts the file and saves it to the buffer  
 output\_file.write(decrypted\_bytes) #moves the file from the buffer to an output file  
 buffer = input\_file.read(buffer\_size) #Reads the file from the buffer  
  
input\_file.close() #Closes the input file  
output\_file.close() #Closes the output file  
  
#A function which enables the use of sha256 to hash the files  
def get\_file\_hash(file\_path):  
 block\_size = 65536  
 file\_hash = hashlib.sha256()  
 with open(file\_path, 'rb') as f:  
 fb = f.read(block\_size)  
 while len(fb) > 0:  
 file\_hash.update(fb)  
 fb = f.read(block\_size)  
 return file\_hash.hexdigest()  
  
  
#Opens and prints the contents of zambesi.txt.encrypted also hashes the file.  
f=open ('zambesi.txt','r') #Opens the zambesi.txt file  
file\_contents = f.read() #Reads the contents of Zambesi.txt  
print ('\n', file\_contents) #Prints the zambesi.txt file  
print (get\_file\_hash(file\_to\_encrypt)) #Prints the hash of the Zambesi.txt file  
  
f=open ('zambesi.txt.encrypted','r')  
file\_contents = f.read()  
print ('\n', file\_contents)  
print (get\_file\_hash(file\_to\_encrypt + '.encrypted')) #Prints the hash of the encrypted file  
  
  
f=open ('zambesi.txt.decrypted','r')  
file\_contents = f.read()  
print ('\n', file\_contents)  
print (get\_file\_hash(file\_to\_encrypt + '.decrypted')) #Prints the hash of the decrypted file