

Wonderville IT: Building a Public Key Infrastructure

Prepared by: Mohammed Fouzan Aamiri

Date of Completion: October 15th, 2024

Executive Summary:

This report highlights the implementation of a Public Key Infrastructure (PKI) to enhance document security for Wonderville's town network. As part of my internship, I worked on encrypting sensitive files, ensuring they remain protected, even when accessed remotely by town employees. Using a Self-Signed Certificate created on windows server, I developed a method for encrypting files so that they can be easily decrypted by authorized users, no matter where the files are moved within the network.

The project was tested on a cyber range, set up with three virtual machines: a Windows Server, a Windows 10 machine, and an Ubuntu terminal, each on different subnets and Windows Server acting as a gateway between the other two. The Windows Server acted as the certificate authority, managing the encryption process. I tested how these encrypted files behaved when transferred between hosts on different networks (Windows 10 and Ubuntu) to ensure they remained secure.

The goal was simple: make sure encrypted files could only be decrypted by users with the right permissions and certificates, preventing unauthorized access. The test results showed that this encryption method worked as intended, securing documents across different systems without requiring additional expensive software.

By implementing this solution, Wonderville has taken an important step toward securing its internal documents, protecting sensitive information from potential cyber threats, especially when accessed remotely by staff.

Technical Solutions:

We have a network setup that includes a **Windows Server**, **Windows 10**, and **Ubuntu** terminal, each serving a unique purpose. Our main goal is to create a secure system that can encrypt and decrypt internal documents, addressing concerns raised by Linda about the lack of document security.

Network Topology Overview

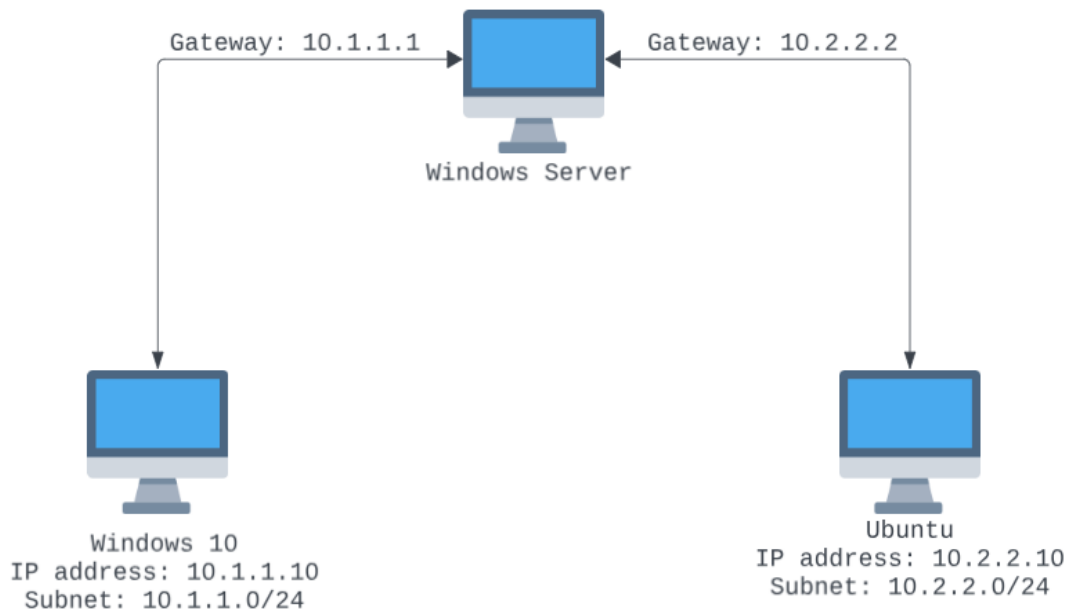


Figure 1: Network Topology

Overview of the Current situation

Right now, we don't have any encryption in place for the internal documents being used on either the Windows 10 or Ubuntu systems. This absence of encryption is a significant risk, especially when documents are shared or accessed externally. To tackle this issue, Linda has emphasized the need for a solid encryption solution that can take advantage of the existing Windows certificate functionalities while allowing for easy access and decryption of documents across both Windows and Linux platforms.

Proposed Network Topology Diagram

- **Figure 1:** This diagram illustrates our network topology, showing how the Windows Server functions as a gateway between the two subnets and highlighting that it does not have an assigned IP address.

Building a Public Key Infrastructure

To address the current lack of encryption for internal documents, we proposed a solution that leverages a self-signed certificate created on the Windows Server. This approach utilizes Windows' existing certificate functionality, allowing us to encrypt internal documents in a way that facilitates easy access and decryption across different operating systems.

Proposed Solution Overview

1. **Generate a Self-Signed Certificate:** We will begin by creating a self-signed certificate on the Windows Server. This certificate will serve as the foundation for our public key infrastructure, providing the necessary keys for encryption and decryption processes. We utilized the code outlined in **Appendix 1**, located in the attachment section of the report.

```
PS C:\Users\Administrator> $cert

PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint                               Subject
-----
086AF7C548B77095F4F69AEB9C8AD01E6E54EC4C  CN=WondervilleEncryption
```

Figure 2: Self-Signed Certificate

2. **Export the Self-Signed Certificate:** After successfully generating the certificate, the next step is to export it along with its private key. It is crucial to export the certificate with its private key, as this allows other machines to securely decrypt the files. Without the private key, the encrypted documents cannot be accessed, rendering them useless.
3. **Encrypt Internal Documents:** Using the public key from the self-signed certificate, we will encrypt the internal documents on the Windows Server. This process ensures that the documents are secure and can only be decrypted by someone possessing the corresponding private key. We utilized the code outlined in **Appendix 2**.

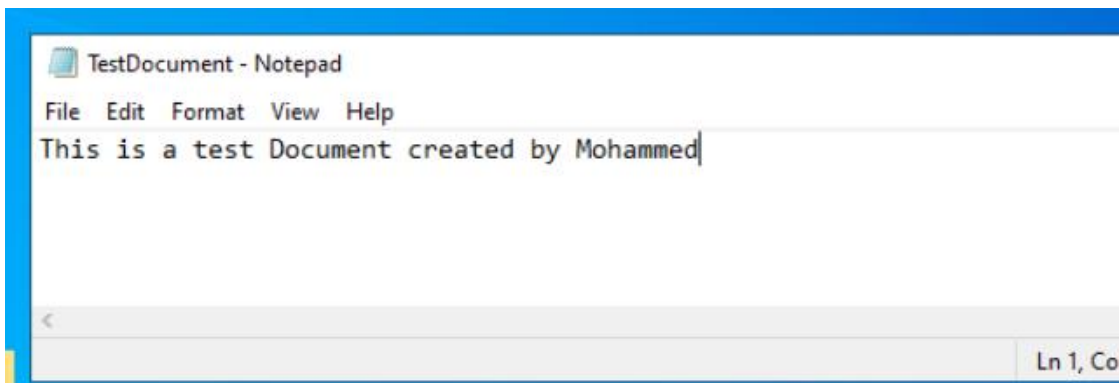


Figure 3: Test Document

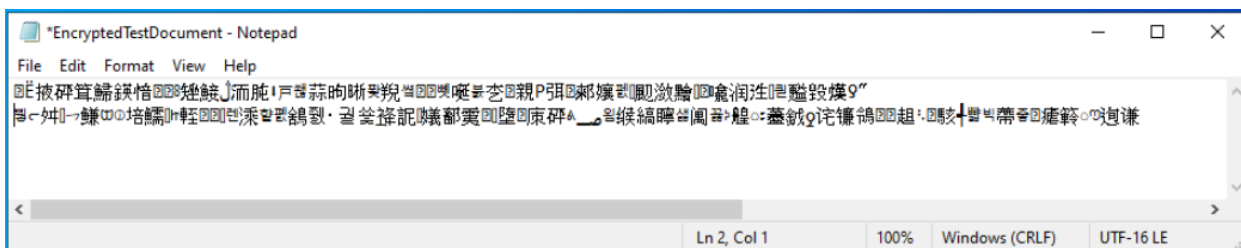
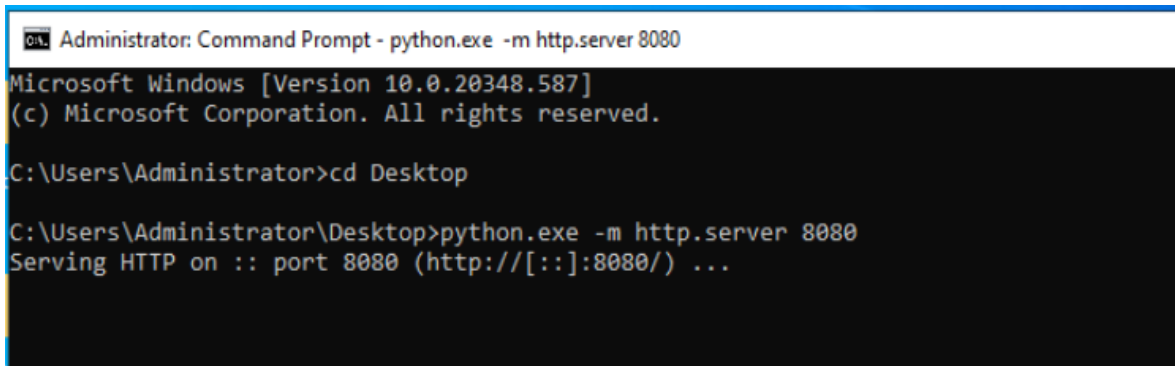


Figure 4: Encrypted Test Document

4. **Transfer the Certificate and Encrypted Files:** To facilitate the transfer of the certificate and the encrypted files, we will utilize a Python HTTP server. It is essential to ensure that the firewall is turned off on the machines involved, allowing smooth communication between the systems. Additionally, when running the Python HTTP server on the Windows Server, make sure you are in the directory where the encrypted files and the exported certificate are located. This setup enables users on Windows 10 and Ubuntu to securely download the necessary files.



```
Administrator: Command Prompt - python.exe -m http.server 8080
Microsoft Windows [Version 10.0.20348.587]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>cd Desktop

C:\Users\Administrator\Desktop>python.exe -m http.server 8080
Serving HTTP on :: port 8080 (http://[::]:8080/) ...
```

Figure 5: Python http server initiated on windows server

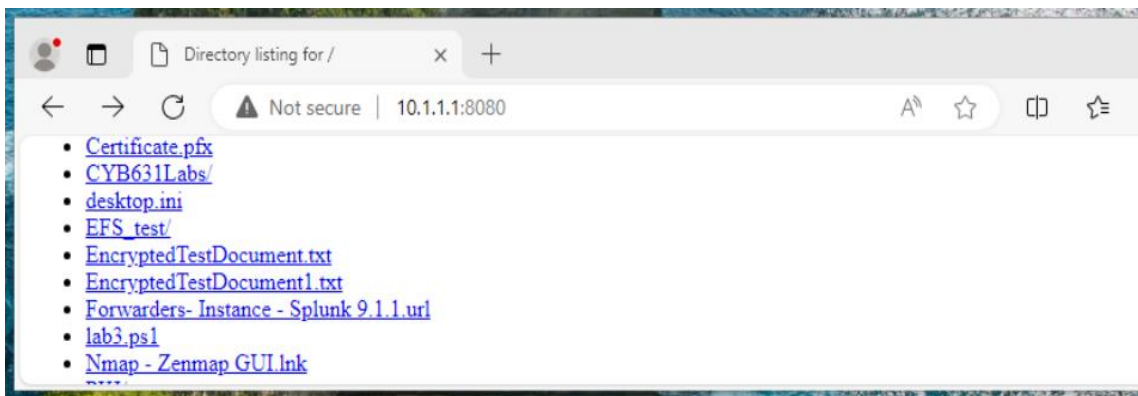


Figure 6: Accessing the Desktop location of Windows server on windows 10

5. **Import the Certificate on Client Machines:** Once the certificate is on the client machines, we will import it into the certificate store. This will allow the decryption process to utilize the private key securely stored in the certificate.
6. **Decrypt the Documents:** Finally, we will demonstrate how to decrypt the encrypted documents on both Windows 10 and Ubuntu systems. This step showcases the functionality of our solution and ensures that documents remain accessible, regardless of the host they are moved to. We utilized the code outlined in **Appendix 3**.



Figure 7: Decrypted Test Document on Windows 10

Recommendations:

Our technical solution building a Public Key Infrastructure (PKI) using a self-signed certificate on the Windows Server proved effective based on the results of our implementation. By utilizing Windows' existing security features, we were able to securely encrypt internal documents, ensuring they can be easily decrypted on other machines, particularly Windows systems. This approach addresses the need for document encryption while allowing files to be transferred and decrypted between different hosts.

Why the Solution Works:

- **Secure Document Encryption:** The encryption process using a self-signed certificate ensures that only users with the corresponding private key can decrypt and access the files. This provides robust security, even when files are transferred between systems, preventing unauthorized access.
- **Cross-Machine Compatibility:** By exporting the certificate and using it on other machines, we've demonstrated that decryption can happen smoothly across different operating systems (e.g., Windows 10). This cross-machine functionality makes the solution adaptable for various environments.
- **Leveraging Existing Resources:** The solution relies on Windows' built-in certificate and security features, which means no extra software or tools are required. This keeps costs low while maintaining high security for internal documents.

Reflections:

Challenges with EFS Certificate: The case study initially required using the Windows Encrypting File System (EFS) certificate to encrypt the file. While I was able to encrypt the file using EFS, I faced a significant limitation: the EFS certificate could not be exported along with its private key. Without the private key, successful decryption on other machines was impossible. This issue led me to pivot using a self-signed certificate, which allowed me to control the keys and successfully complete the encryption and decryption process across different systems.

Decryption Issues on Ubuntu: I successfully transferred both the encrypted file and the self-signed certificate to the Ubuntu machine. However, decryption on Ubuntu was unsuccessful. This could be due to compatibility issues between the encryption algorithms. I used RSA encryption with OAEP padding, which worked seamlessly on Windows 10 but not on Ubuntu. Additionally, the lack of a graphical user interface (GUI) on the Ubuntu system limited my ability to diagnose and resolve the issue. Restrictions such as no internet access also prevented me from downloading libraries or tools that could have facilitated troubleshooting and decryption.

```
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ ls
Certificate.pfx  EncryptedTestDocument.txt  scripts
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ _
```

Figure 8: Successful transfer of the files to ubuntu

```
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ openssl pkcs12 -in Certificate.pfx -nocerts -out private_key.pem -nodes
Enter Import Password:
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ ls
Certificate.pfx  EncryptedTestDocument.txt  private_key.pem  scripts
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ openssl pkeyutil -decrypt -in EncryptedTestDocument.txt -out DecryptedTestDocument.txt -inkey private_key.pem
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ ls
Certificate.pfx  DecryptedTestDocument.txt  EncryptedTestDocument.txt  private_key.pem  scripts
student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ cat DecryptedTestDocument.txt
kcfR####s-####hVg####XT##P##6MI#N'[[[student@cyb631fa241c1-cr-ma35224n-pool-ubuntu:~$ _
```

Figure 9: Issues with Decryption

Attachments:

Appendix 1:

```
$selfcert=New-SelfSignedCertificate -CertStoreLocation Cert:\LocalMachine\My -Subject "CN=WondervilleEncryption"
```

\$selfcert

Appendix 2:

```
# Load the self-signed certificate
```

```
$cert = Get-ChildItem Cert:\LocalMachine\My | Where-Object { $_.Thumbprint -eq "086AF7C548B77095F4F69AEB9C8AD01E6E54EC4C" }
```

```
# Create an RSA provider from the public key in the certificate
```

```
$rsa =  
[System.Security.Cryptography.X509Certificates.RSACertificateExtensions]::GetRSAPublicKey  
($cert)
```

```

# Read the content of the file to encrypt

$fileToEncrypt = " C:\Users\Administrator\Desktop\TestDocument.txt"

$contentToEncrypt = [System.IO.File]::ReadAllBytes($fileToEncrypt)

# Encrypt the content using the RSA public key

$encryptedData = $rsa.Encrypt($contentToEncrypt,
[System.Security.Cryptography.RSAEncryptionPadding]::OaepSHA1)

# Write the encrypted content to a new file

$encryptedFile = " C:\Users\Administrator\Desktop\EncryptedTestDocument.txt"

[System.IO.File]::WriteAllBytes($encryptedFile, $encryptedData)

```

Appendix 3:

```

# Load the self-signed certificate

$cert = Get-ChildItem Cert:\LocalMachine\My | Where-Object { $_.Thumbprint -eq
"086AF7C548B77095F4F69AEB9C8AD01E6E54EC4C " }

# Create an RSA provider from the private key in the certificate

$rsa =
[System.Security.Cryptography.X509Certificates.RSACertificateExtensions]::GetRSAPrivateKey($cert)

# Read the content of the encrypted file

$fileToDecrypt = "C:\Users\Student\Downloads\EncryptedTestDocument.txt"

$encryptedData = [System.IO.File]::ReadAllBytes($fileToDecrypt)

# Decrypt the content using the RSA private key

$decryptedData = $rsa.Decrypt($encryptedData,
[System.Security.Cryptography.RSAEncryptionPadding]::OaepSHA1)

# Write the decrypted content to a new file

$fileDecrypted = "C:\Users\Student\Downloads\decryptedTestDocument.txt"

[System.IO.File]::WriteAllBytes($fileDecrypted, $decryptedData)

```