CYB631 – Lab 4: Confidentiality

Vijaysingh Puwar

Seidenberg School of CSIS, Pace University

CYB 631: Automating Information Security with Python & PowerShell

Prof. Darren Hayes

September 29, 2025

Contents

ABSTRACT

This lab operationalizes core confidentiality principles in Windows using PowerShell and X.509 public-key infrastructure. First, file integrity was demonstrated by generating cryptographic digests with Get-FileHash and building a two-column SHA-256 report, followed by validating a vendor download via hash comparison to confirm authenticity. Next, sensitive data handling was explored with SecureString, including converting secrets to AES-encrypted ciphertext and decrypting them using explicitly managed 16/24/32-byte keys stored in a protected key file. Secure credential storage was reinforced by exporting an encrypted password and analyzing the risks of exposed key material. The lab then implemented script trust through a self-signed **Code Signing** certificate, applying Set-AuthenticodeSignature and verifying signature status to satisfy execution policy controls. Finally, confidentiality at rest and in transit was exercised through CMS encryption: generating a Document Encryption certificate, protecting plaintext with Protect-CmsMessage, and recovering it with Unprotect-CmsMessage under the original user profile. Evidence was captured via reproducible commands, aligned screenshots, and brief interpretations. Collectively, these tasks connect theory to practice, illustrating how hashing, symmetric and asymmetric cryptography, and code signing mitigate integrity and confidentiality risks in day-to-day security operations.

EXERCISES:[ENVIRONMENT: STARTING WITH POWERSHELL ISE]

Step 1

Here I install Windows 2022 Server in Virtual box and using the CLI command to navigate the powershell

```
Powershell
```

This command help us to install the powershell. And in order to launch the powershell use this command

```
Powershell -Command "Start-Proccess powershell -Verb runAs"
```

Step 2

Now, open a new PowerShell ISE and run as an administrator. Make sure that you navigate to the directory where your script is. Change the executive policy to RemoteSigned.

```
cd C:\Users\cybusr\cyb631
Set-ExecutionPolicy remotesigned
```

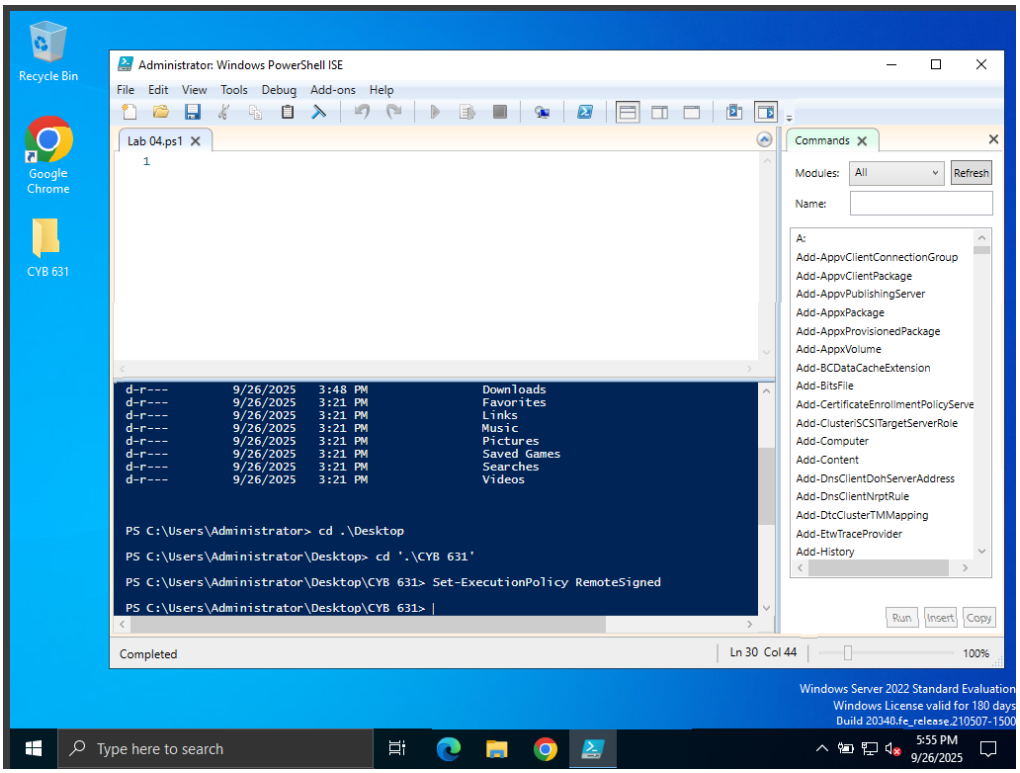


Figure 1

EXERCISE I — HASHING

Step 3

PowerShell has a Get-FileHash cmdlet for hashing files. The default hash function is **SHA256** if it is not specified.

Step 4

Create some files for testing file hash. Let us learn about how to do this with a loop. Can see in below **screenshot.**

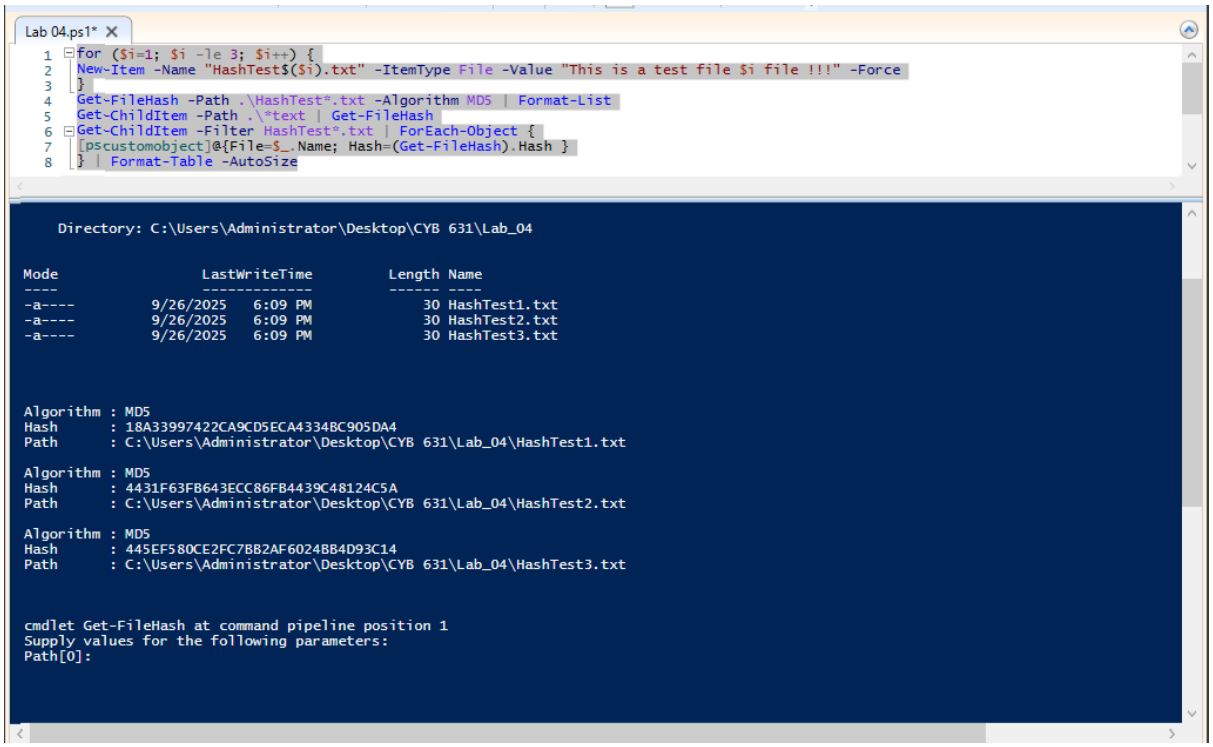


Figure 2

Step 5

We can then generate hash for all the files that you have just created. Here, we specify **MD5** as the hashing algorithm.

```
Get-FileHash -Path .\HashTest*.txt -Algorithm MD5 | Format-List
```

Step 6

Here is the resule of the above command

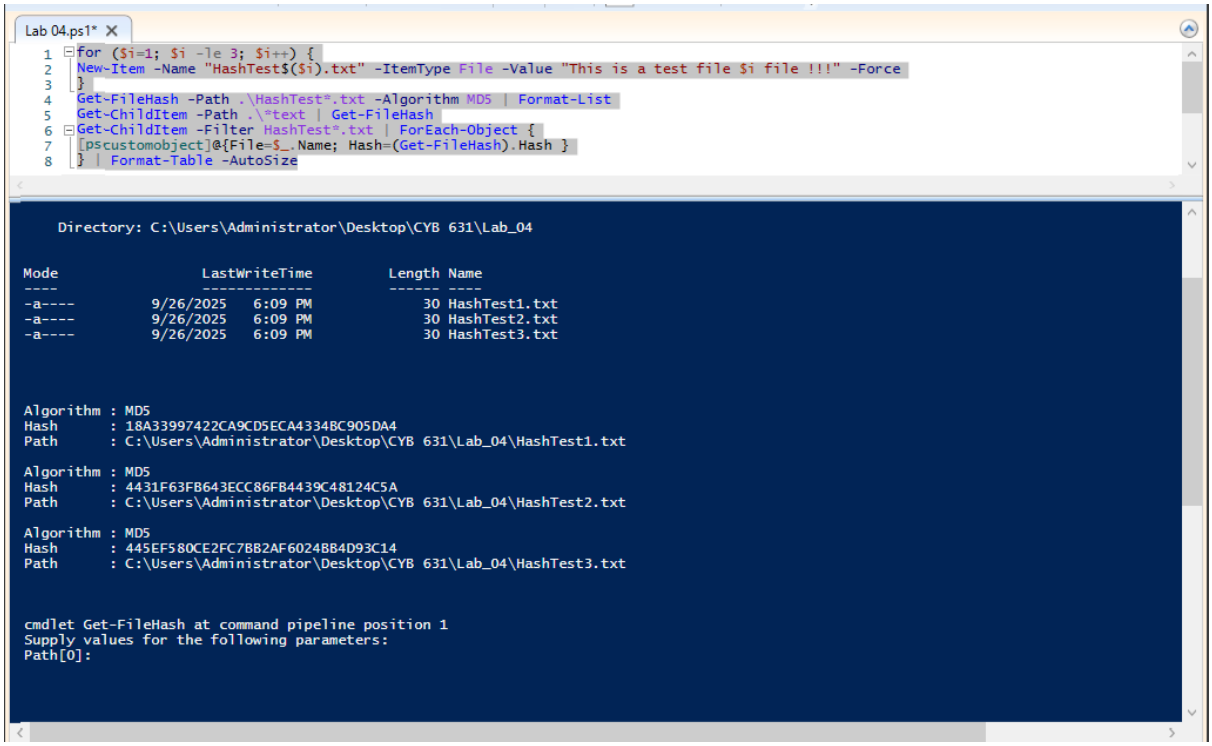


Figure 3

Step 7

Alternatively, we can search for all the files under a directory and pipe them to the function.

```
Get-ChildItem -Path .\*.txt | Get-FileHash
```

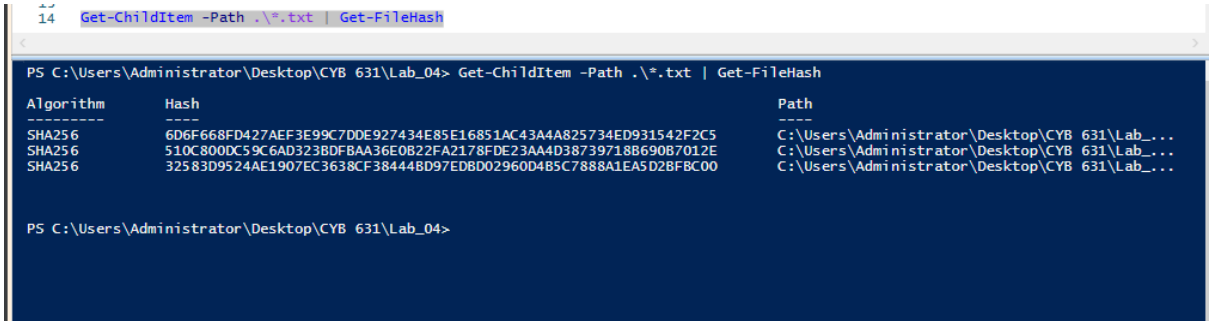


Figure 4

Step 8

Develop a script to print a concise two-column report (**File** | **Hash**) for all HashTest*.txt files. The first column lists the file name and the second column lists the corresponding **SHA-256** hash value of the file.

```
Get-ChildItem -Filter 'HashTest*.txt' | ForEach-Object {
$h = Get-FileHash -Algorithm SHA256 -Path $_.FullName
'{0,-20} {1}' -f $_.Name, $h.Hash
      }
```

Step 9-10

As you can see in the figure the hash of the file. In SHA256.

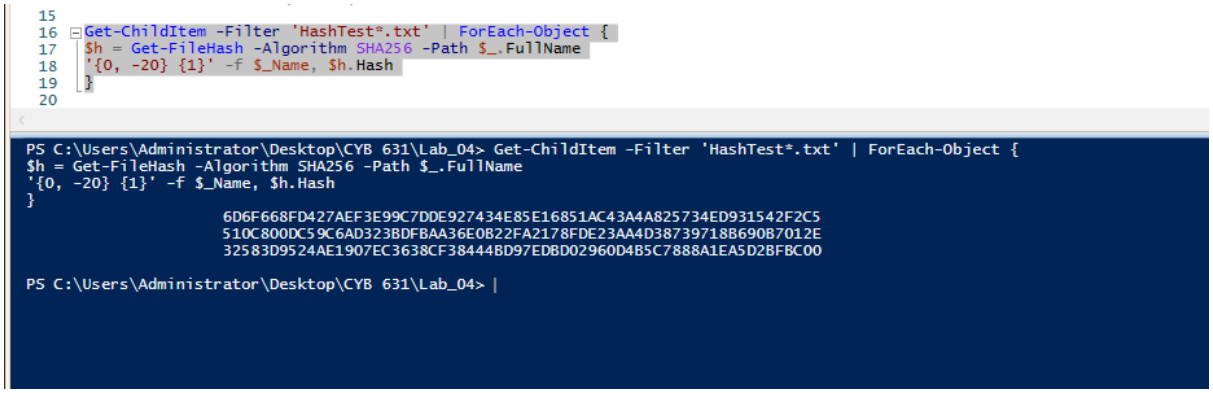


Figure 5

Exercise II: Integrity Check for Downloading Files

Step 11

Further, we can use Get-FileHash to perform an integrity check for downloading a file (software) from the Internet.

Step 12

Create a Web Client.

```
20
21  $wc = [System.Net.WebClient]::new()
```

```
PS C:\Users\Administrator\Desktop\CYB 631\Lab 04> $wc = [System.Net.WebClient]::new()

PS C:\Users\Administrator\Desktop\CYB 631\Lab_04>
```

Figure 6

Step 13

We will use a download link from Microsoft and verify it with the pre-calculated file hash of the software.

```
22
23  $pkgurl = 'https://github.com/PowerShell/PowerShell/releases/download/v6.2.4/powershell_6.2.4-1.debian.9_amd64.deb'
24  $publishHash = '8E28E54D601F0751922DE24632C1E716B4684876255CF82304A9B19E89A9CCAC'
```

```
PS C:\Users\Administrator\Desktop\CYB 631\Lab 04>
$pkgurl = 'https://github.com/PowerShell/PowerShell/releases/download/v6.2.4/powershell_6.2.4-1.debian.9_amd64.deb'
$publishHash = '8E28E54D601F0751922DE24632C1E716B4684876255CF82304A9B19E89A9CCAC'

PS C:\Users\Administrator\Desktop\CYB 631\Lab_04>
```

Step 14

Get the file hash from the file being downloaded via the web client link. Paste screenshot.

```
$FileHash = Get-FileHash -InputStream ($wc.OpenRead(pkgurl))
```

Step 15

Compare the Hash

```
$result = $FileHash.Hash -eq $publishedHash

Write-Host "The result of file integrity check is ", $result
```

Step 16

Here is the screenshot

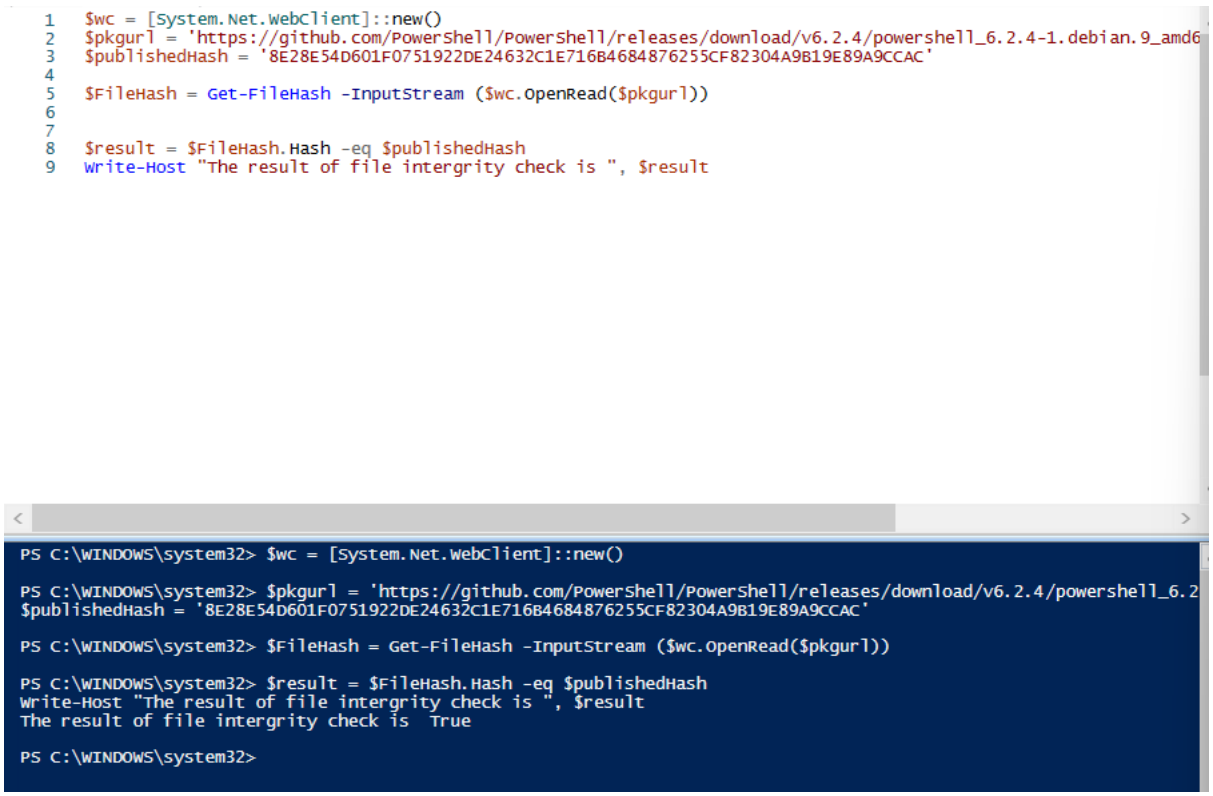


Figure 7

Step 17- What does the result mean? Please explain it.

The result True indicates that the hash value of the downloaded file exactly matches the published hash provided by the software vendor. This confirms that the file has not been altered, corrupted, or tampered with during transmission and is therefore authentic. If the result had been False, it would suggest that the file might be damaged, incomplete, or potentially compromised, and it should not be trusted for installation.

Exercise III: SecureString and AES Encryption

Step 18

PowerShell has a data type called SecureString. It stores data in the memory that cannot be viewed by users but the content is not encrypted in the memory. Read a string from command line and store it as a SecureString.

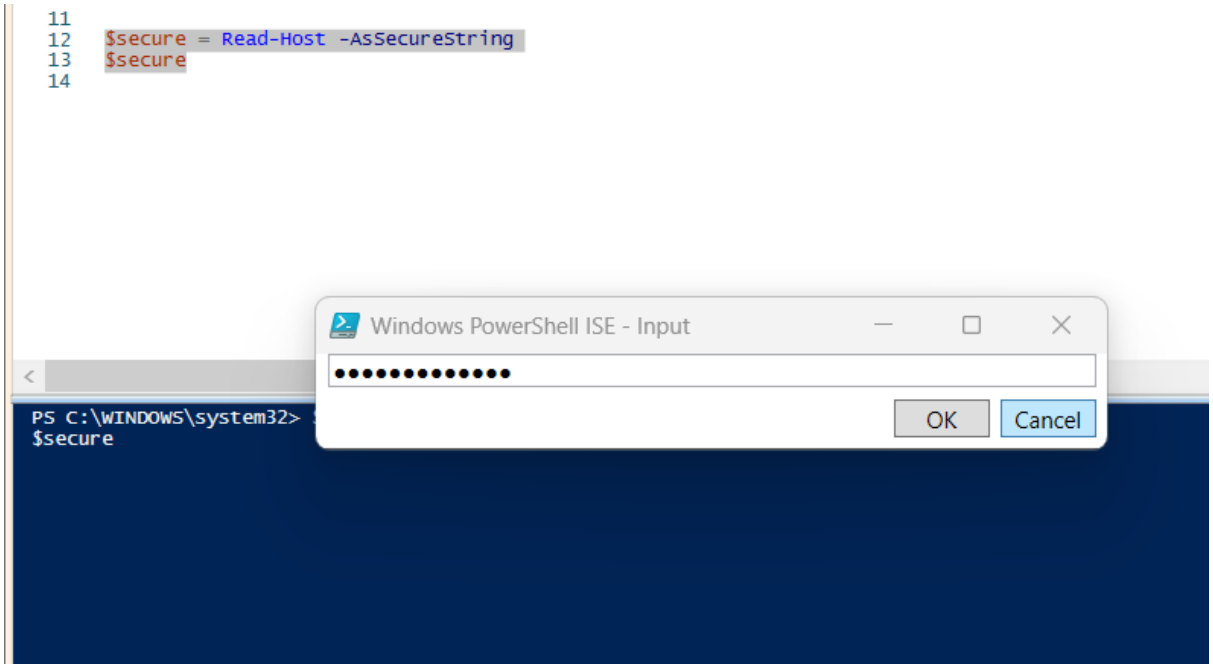


Figure 8

Step 19

Convert the SecureString into an encrypted text. The default encryption used in PowerShell is AES with a key in 16 Bytes, if not specified.

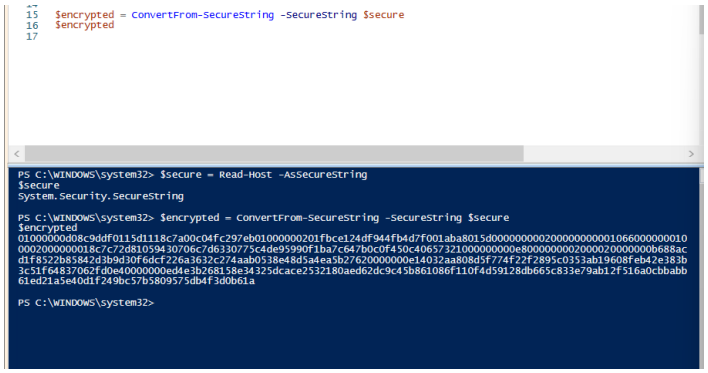


Figure 9

Step 20

You can always decrypt it on the same machine. On a different machine, you will need to specify a file that stores the encryption key since it is encrypted by AES previously.

```
17
18  $decrypted = $encrypted | ConvertTo-SecureString
19  $decrypted
20
```

```
PS C:\WINDOWS\system32> $decrypted = $encrypted | ConvertTo-SecureString
$decrypted
System.Security.SecureString

PS C:\WINDOWS\system32>
```

Figure 10

Step 21

To convert the Secure String to plain text, you will need to copy the SecureString in the memory to an unmanaged binary string (BSTR). Then you convert the BSTR variable to a managed string.

```
20
21  $bstr     = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secure)
22  $unsecure = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
23  $unsecure
24
```

```
PS C:\WINDOWS\system32> $bstr     = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secure)
$unsecure = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
$unsecure
8980042155jj@

PS C:\WINDOWS\system32>
```

Figure 11

Step 22

What is the content of the three variables above?

$secure: _____

$encrypted:_____

$unsecure: _____

How are they different? ___

Does SecureString date type contain encrypted information? _____

Explanation (answer guide):

- $secure → a SecureString object (not readable text).
- $encrypted → AES-encrypted string (portable if you use a key).
- $unsecure → plaintext you typed.
- **Difference:** memory-protected object vs ciphertext vs plaintext.
- **SecureString contain encrypted info?** It's protected in memory; true *portable* encryption is when you export with ConvertFrom-SecureString.

Step 23

To specify an encryption key, you can either assign a key with a fixed value or generate a random key (a better method).

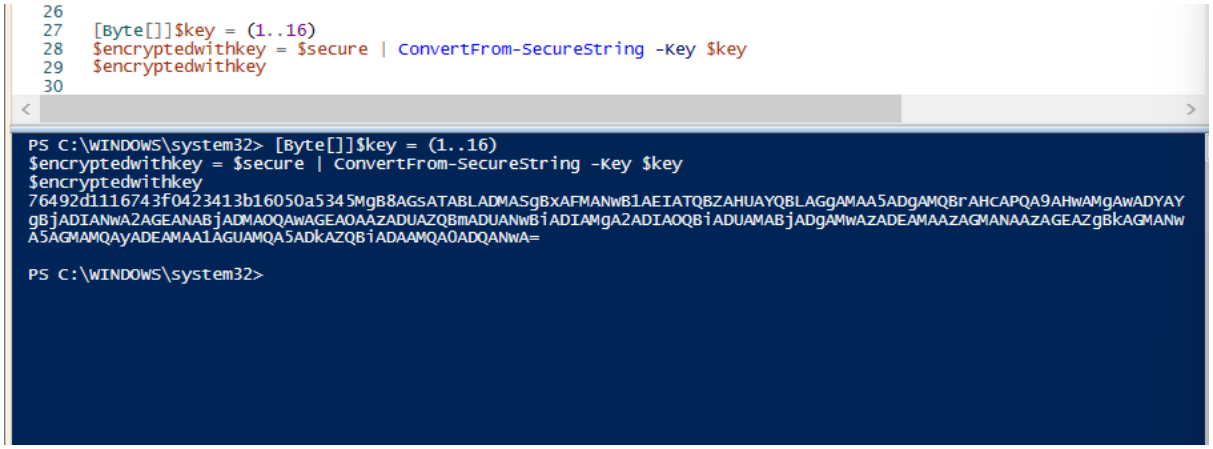


Figure 12

Step 24

Alternatively, you can generate a random 16-byte (128 bits) key. Write the key to a key file which can be stored in the machine where decryption is needed.

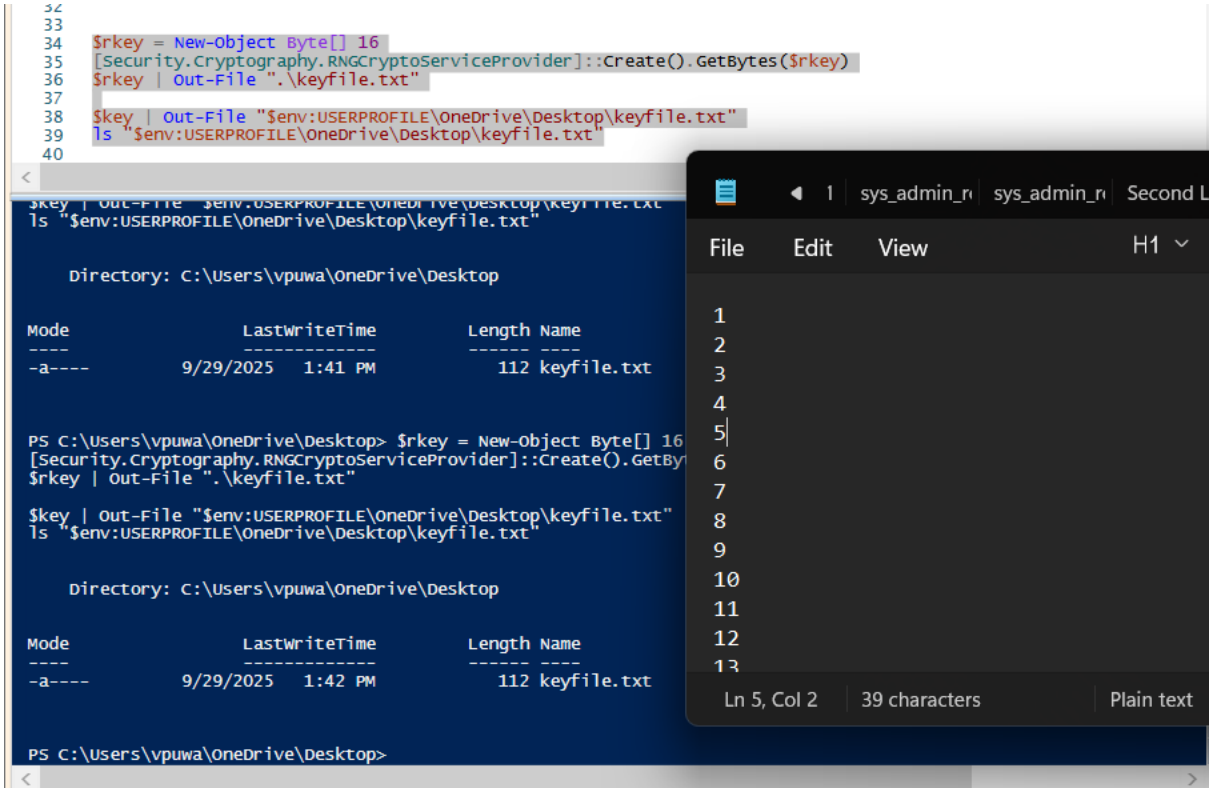


Figure 13

Step 25

Write your own script to read the key from the key file, encrypt the $secure variable from the previous step and then decryption it using the same key.

```
# === Generate a 16-byte AES key and save as raw bytes ===
$rkey = New-Object byte[] 16
[Security.Cryptography.RandomNumberGenerator]::Create().GetBytes($rkey)
Set-Content -Path .\keyfile.bin -Value $rkey -Encoding Byte   # <-- raw bytes


# === Read the key back as bytes (must be 16/24/32) ===
[byte[]]$keyIn = Get-Content -Path .\keyfile.bin -Encoding Byte
if ($keyIn.Length -notin 16,24,32) { throw "Key must be 16, 24, or 32 bytes." }


# === Encrypt your SecureString with the key ===
```

```
$cipher = ConvertFrom-SecureString -SecureString $secure -Key $keyIn
Set-Content -Path .\secret.enc -Value $cipher -NoNewline # avoid extra newline/BOM

# === Decrypt back to SecureString with the same key ===
$secure2 = Get-Content -Path .\secret.enc -Raw | ConvertTo-SecureString -Key $keyIn

# === (Optional) show plaintext to prove success ===
$bstr2     = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($secure2)
$plaintext = [Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr2)
Write-Host "Decrypted plaintext: $plaintext"
```

Step 26

Here is the result:

```
33
34  # === Generate a 16-byte AES key and save as raw bytes ===
35  $rkey = New-Object byte[] 16
36  [Security.Cryptography.RandomNumberGenerator]::Create().GetBytes($rkey)
37  Set-Content -Path .\keyfile.bin -Value $rkey -Encoding Byte   # <-- raw bytes
38
39  # === Read the key back as bytes (must be 16/24/32) ===
40  [byte[]]$keyIn = Get-Content -Path .\keyfile.bin -Encoding Byte
41  if ($keyIn.Length -notin 16,24,32) { throw "Key must be 16, 24, or 32 bytes." }
42
43  # === Encrypt your SecureString with the key ===
44  $cipher = ConvertFrom-SecureString -SecureString $secure -Key $keyIn
45  Set-Content -Path .\secret.enc -Value $cipher -NoNewline       # avoid extra newline/BOM
46
47  # === Decrypt back to SecureString with the same key ===
48  $secure2 = Get-Content -Path .\secret.enc -Raw | ConvertTo-SecureString -Key $keyIn
49
50  # === (Optional) show plaintext to prove success ===
51  $bstr2     = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($secure2)
52  $plaintext = [Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr2)
53  Write-Host "Decrypted plaintext: $plaintext"
54
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> # === Generate a 16-byte AES key and save as raw bytes ===
$rkey = New-Object byte[] 16
[Security.Cryptography.RandomNumberGenerator]::Create().GetBytes($rkey)
Set-Content -Path .\keyfile.bin -Value $rkey -Encoding Byte   # <-- raw bytes

# === Read the key back as bytes (must be 16/24/32) ===
[byte[]]$keyIn = Get-Content -Path .\keyfile.bin -Encoding Byte
if ($keyIn.Length -notin 16,24,32) { throw "Key must be 16, 24, or 32 bytes." }

# === Encrypt your SecureString with the key ===
$cipher = ConvertFrom-SecureString -SecureString $secure -Key $keyIn
Set-Content -Path .\secret.enc -Value $cipher -NoNewline       # avoid extra newline/BOM

# === Decrypt back to SecureString with the same key ===
$secure2 = Get-Content -Path .\secret.enc -Raw | ConvertTo-SecureString -Key $keyIn

# === (Optional) show plaintext to prove success ===
$bstr2     = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($secure2)
$plaintext = [Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr2)
Write-Host "Decrypted plaintext: $plaintext"
Decrypted plaintext: 8980042155j@

PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 14

Step 26

The result of Exercise is in above screenshot.

Exercise IV: Secure Passwords

Step 28

You can obtain credential information from users using Get-Credential cmdlet, which returns an object called “PSCredential”.

$mycred = Get-Credential

$mycred

Step 29

The result is shown in the screenshot below

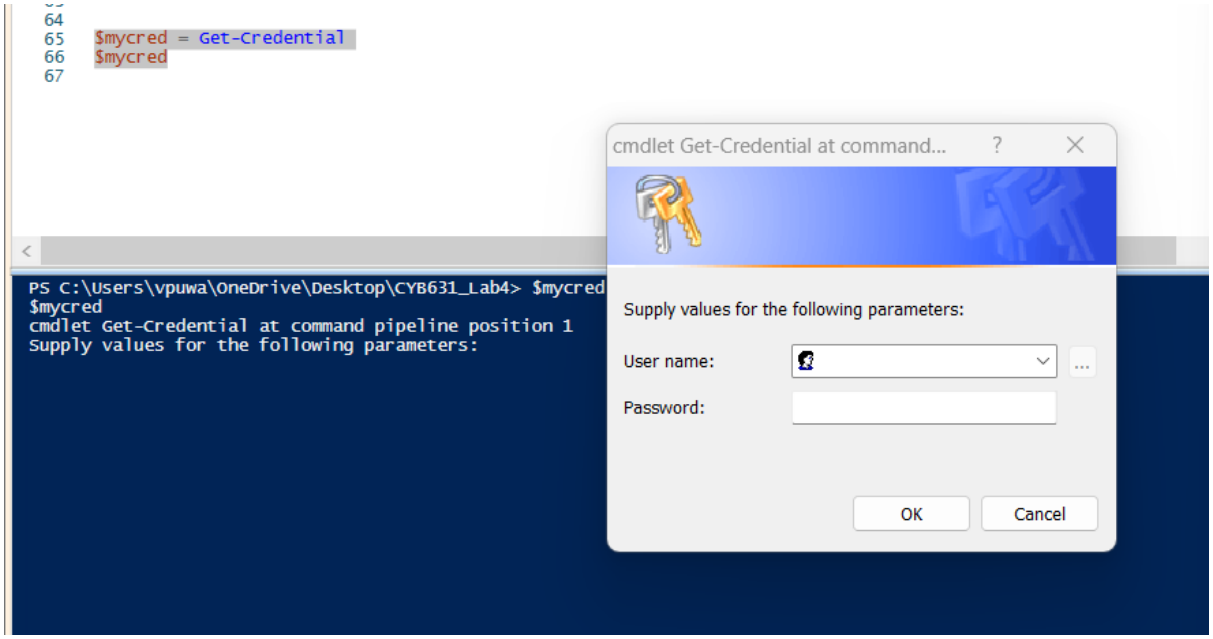


Figure 15

Step 30

You can obtain the key that you generated from the previous step in the key file to encrypt the password in the credential. The encrypted password can then be stored in a file.

$kfile = ".\keyfile.txt"

$mycred.Password | ConvertFrom-SecureString -Key (Get-Content $kfile) | Out-File ".\password.txt"

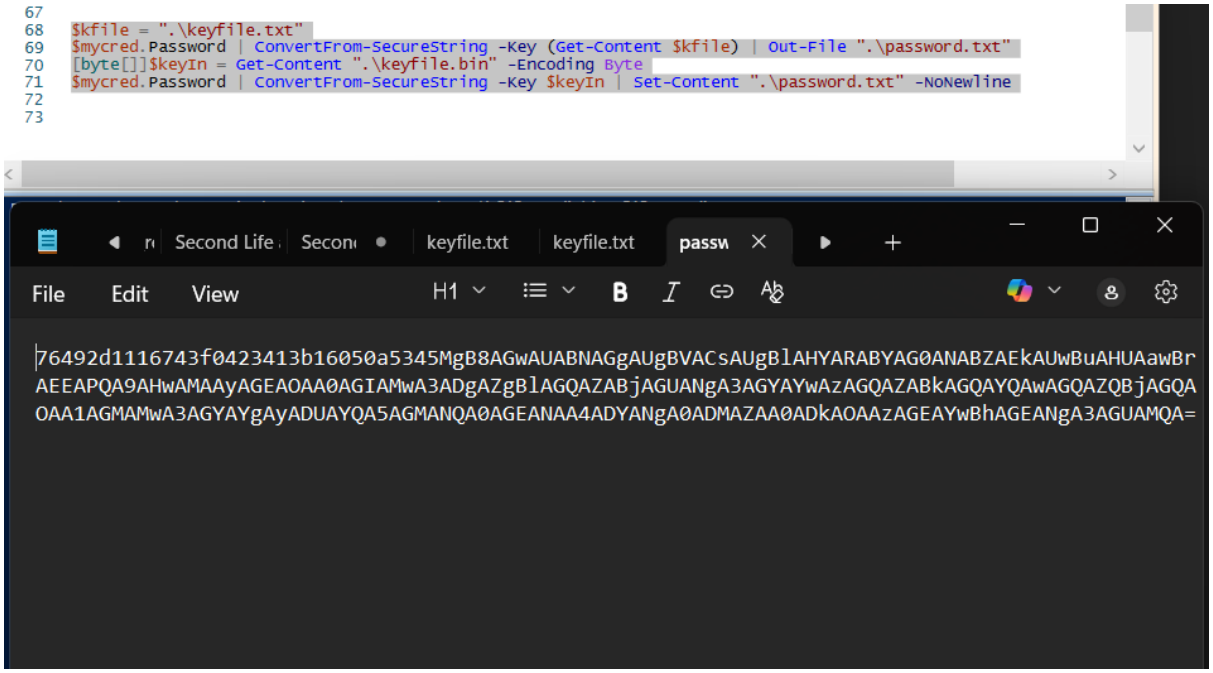


Figure 16

Step 31: What would happen if the key file in the above step is missing?

If the key file is missing, the encrypted password stored in password.txt cannot be decrypted. The ciphertext (like the Base64 string in your file) is useless without the exact same AES key. Any attempt to run ConvertTo-SecureString -Key … will fail, and the password cannot be recovered.

Step 32: Is the key file encrypted at this point? What would happen if an adversary obtained the key file?

- The key file itself is **not encrypted**; it is stored in plain form as raw bytes or text.
- If an adversary obtains the key file, they can use it to decrypt any password or SecureString data encrypted with that key. This would completely compromise confidentiality.

Best practice: Protect the key file with strict permissions (NTFS ACLs), or use DPAPI/Windows credential vault instead of storing raw keys in the filesystem.

Exercise V: Self-Signed Certificate and Script Signing

Step 33

PowerShell requires scripts to be signed as a security precaution so that others cannot run malicious scripts on a protected host. Create a self-signed certificate for the computer.

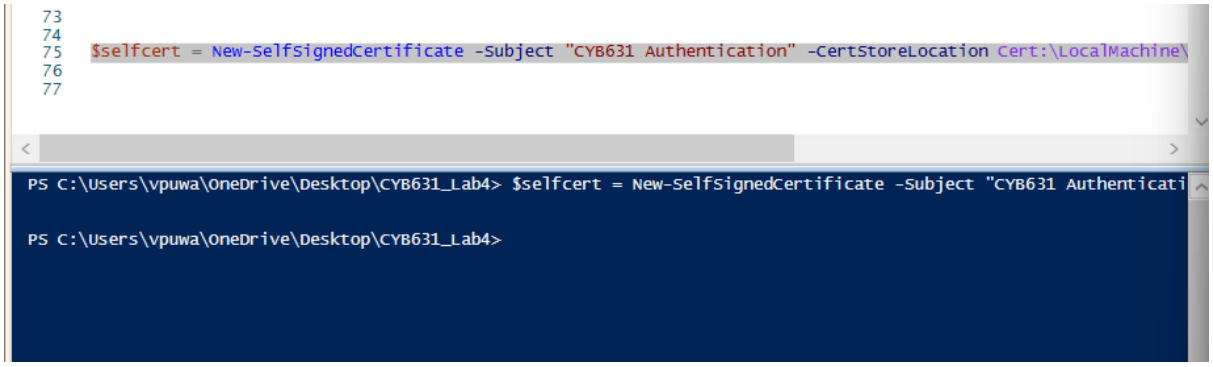


Figure 17

Step 34

Store the self-signed certificate, selcert, in the machine's Root Store and Trusted Publisher's Store.

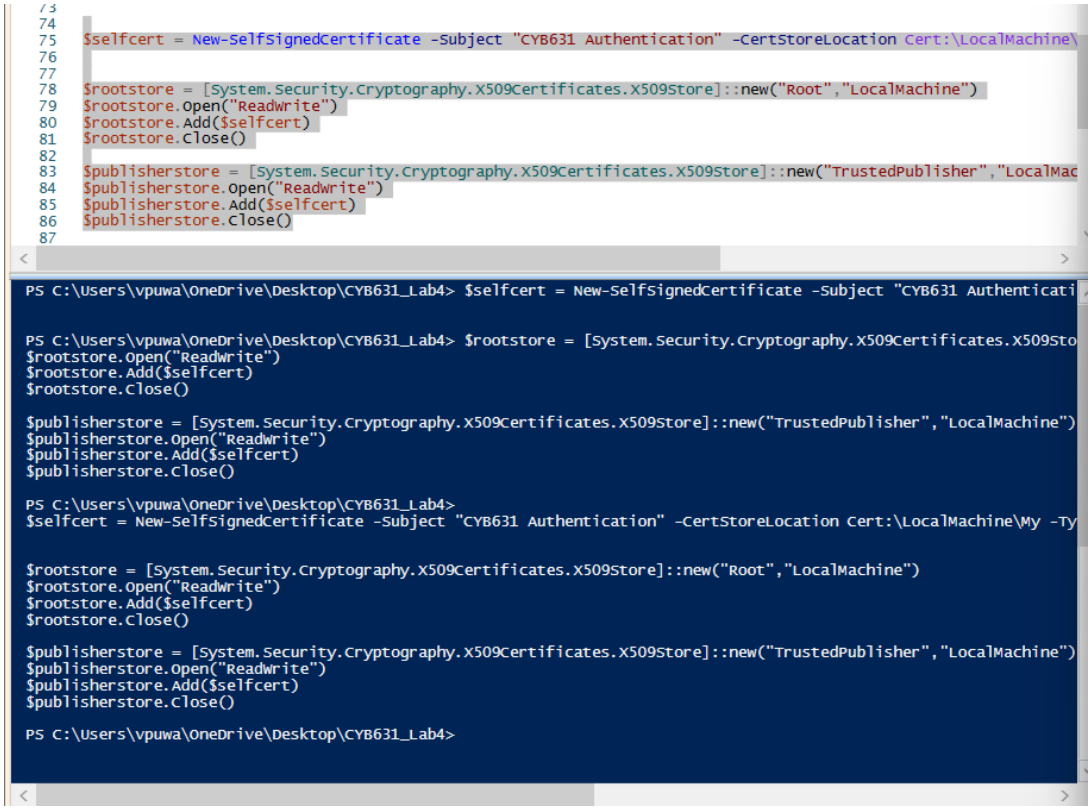


Figure 18

Step 35

Store the self-signed

certmgr.msc

Step 36

You should be able to see the certificate, like the window below. *Paste screenshot.*

Explanation: Show **CYB631 Authentication** under the appropriate store(s).

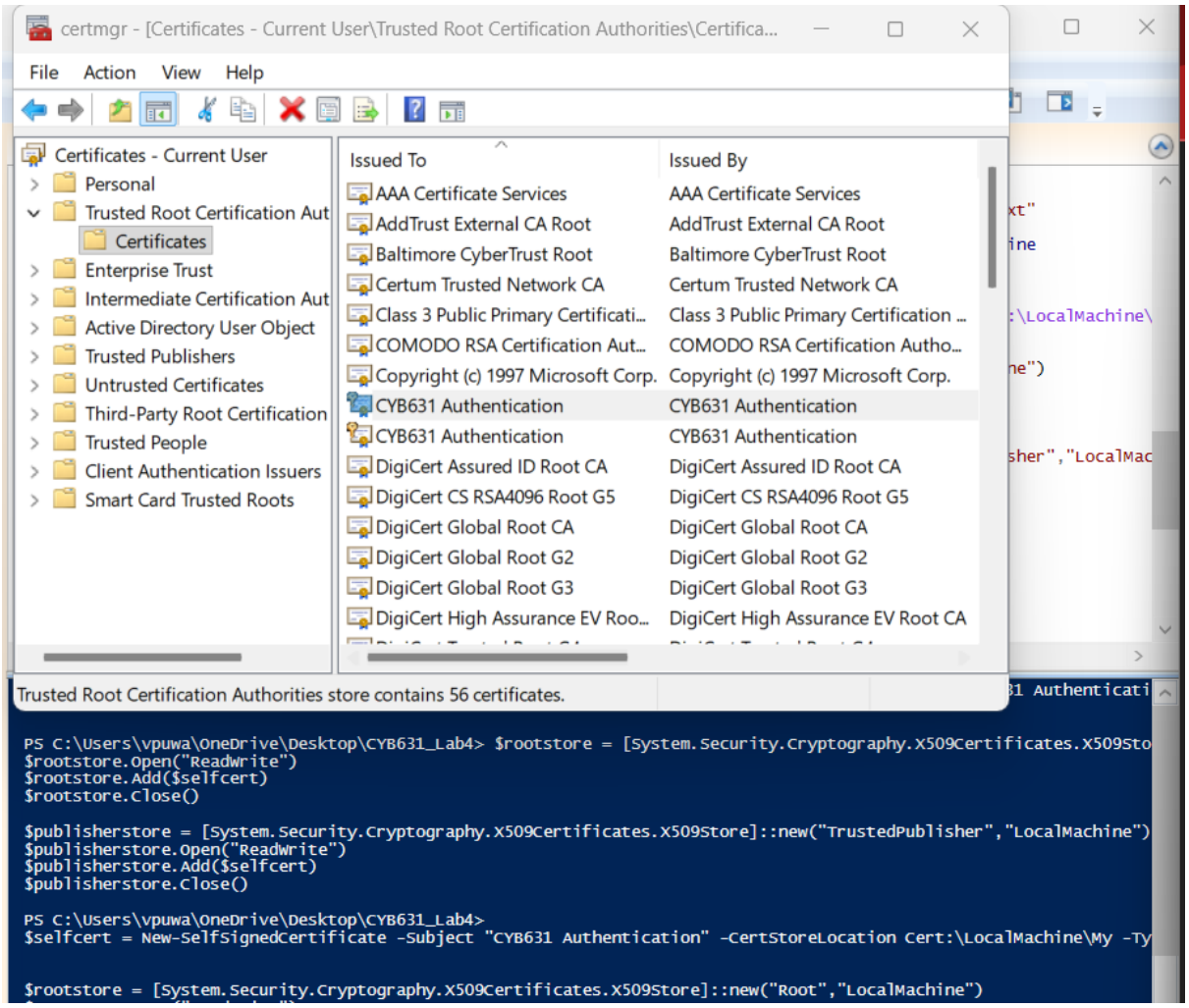


Figure 19

Step 37

Double-click on “CYB631 Authentication” to review the content of the certificate.

Paste screenshot.

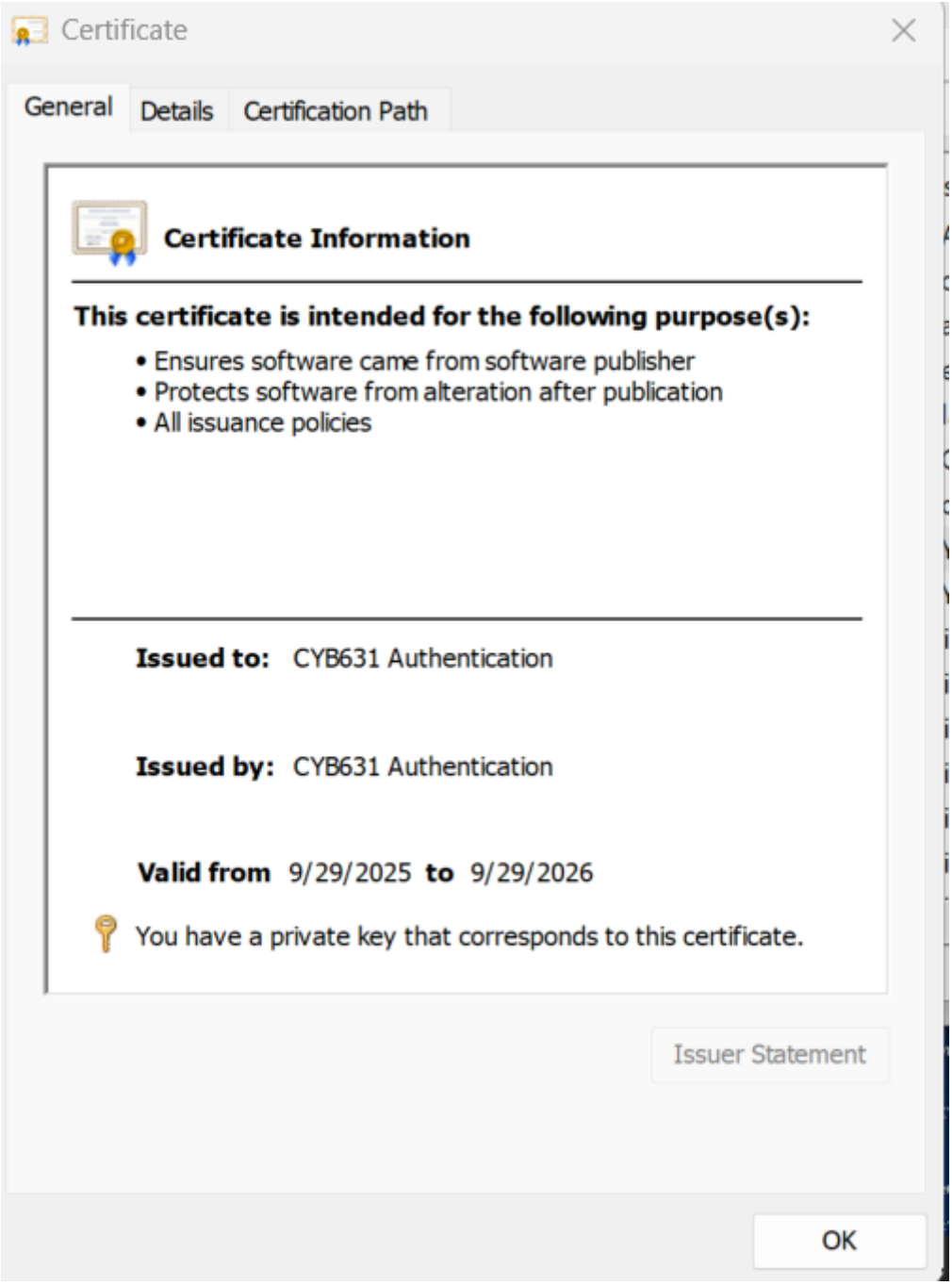


Figure 20

Step 38

You can also use PowerShell to review the certificate in your own path, in the root store and in the trusted publisher's store. Make sure that you have done the previous step correctly.

Get-ChildItem Cert:\LocalMachine\My | Where-Object { $_.Subject -eq "CN=CYB631 Authentication" }

Get-ChildItem Cert:\LocalMachine\Root | Where-Object { $_.Subject -eq "CN=CYB631 Authentication" }

Get-ChildItem Cert:\LocalMachine\TrustedPublisher | Where-Object { $_.Subject -eq "CN=CYB631 Authentication" }

Step 39

This is evidence you'll cite when signing.

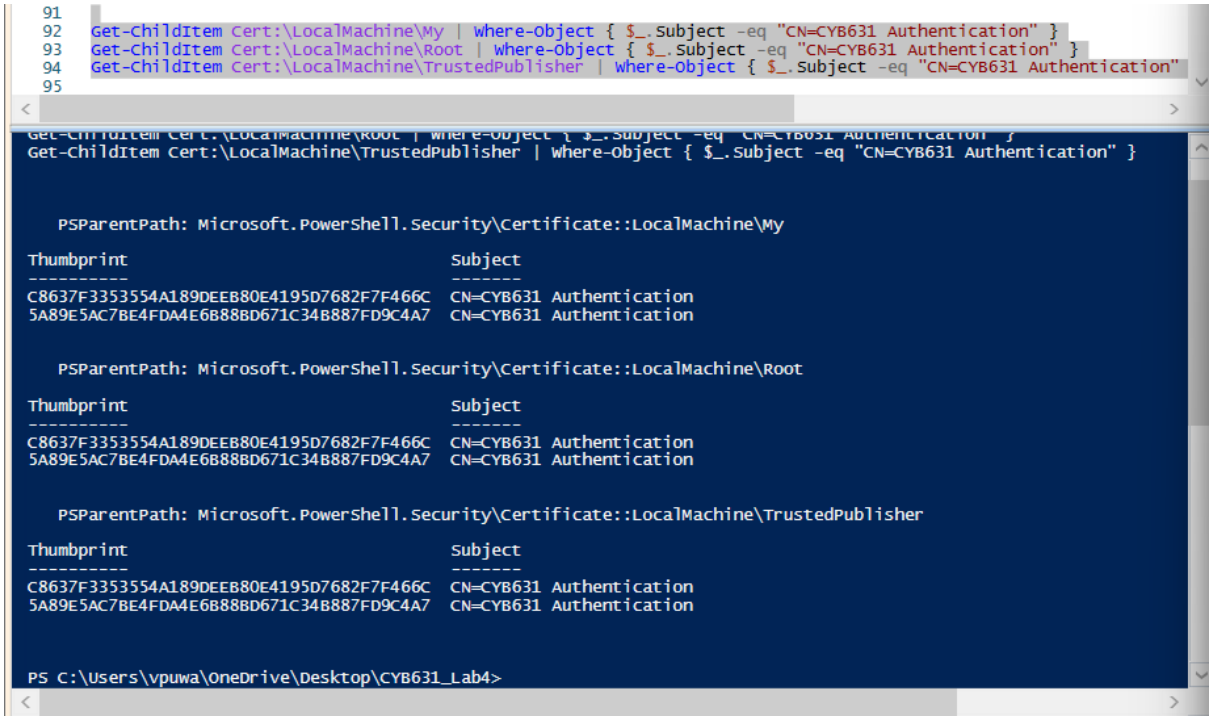


Figure 21

Step 40

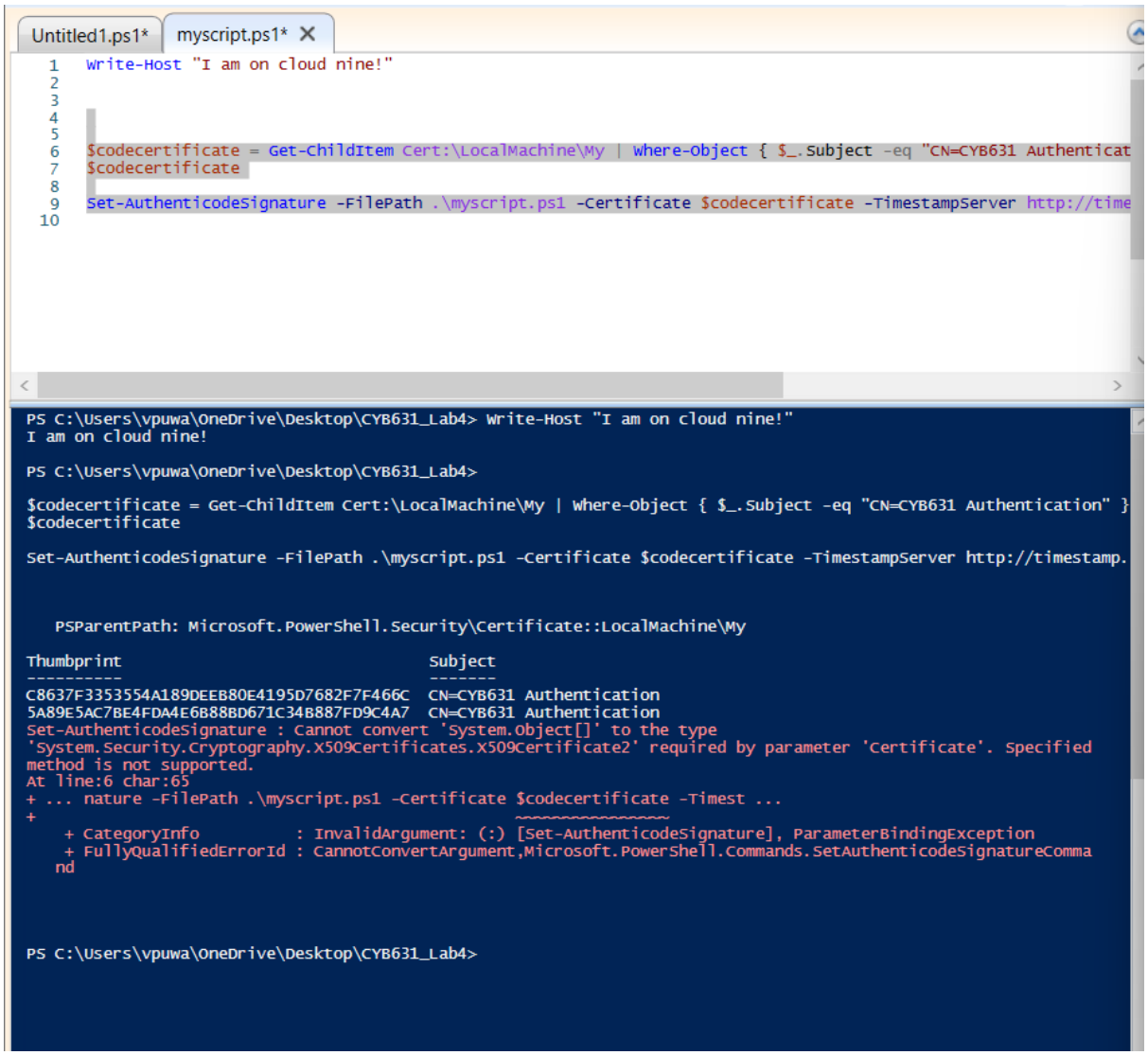
Create a simple PowerShell script for testing certificate signing. *Paste screenshot.* We will sign one of them using the self-signed certificate. For example, creating a file called "myscript.ps1" containing the following codes:

Write-Host "I am on cloud nine!"

Step 41

By default, any new PowerShell scripts have to be signed unless the Execution Policy is changed for that script. You should encounter an error when running myscript.ps1.

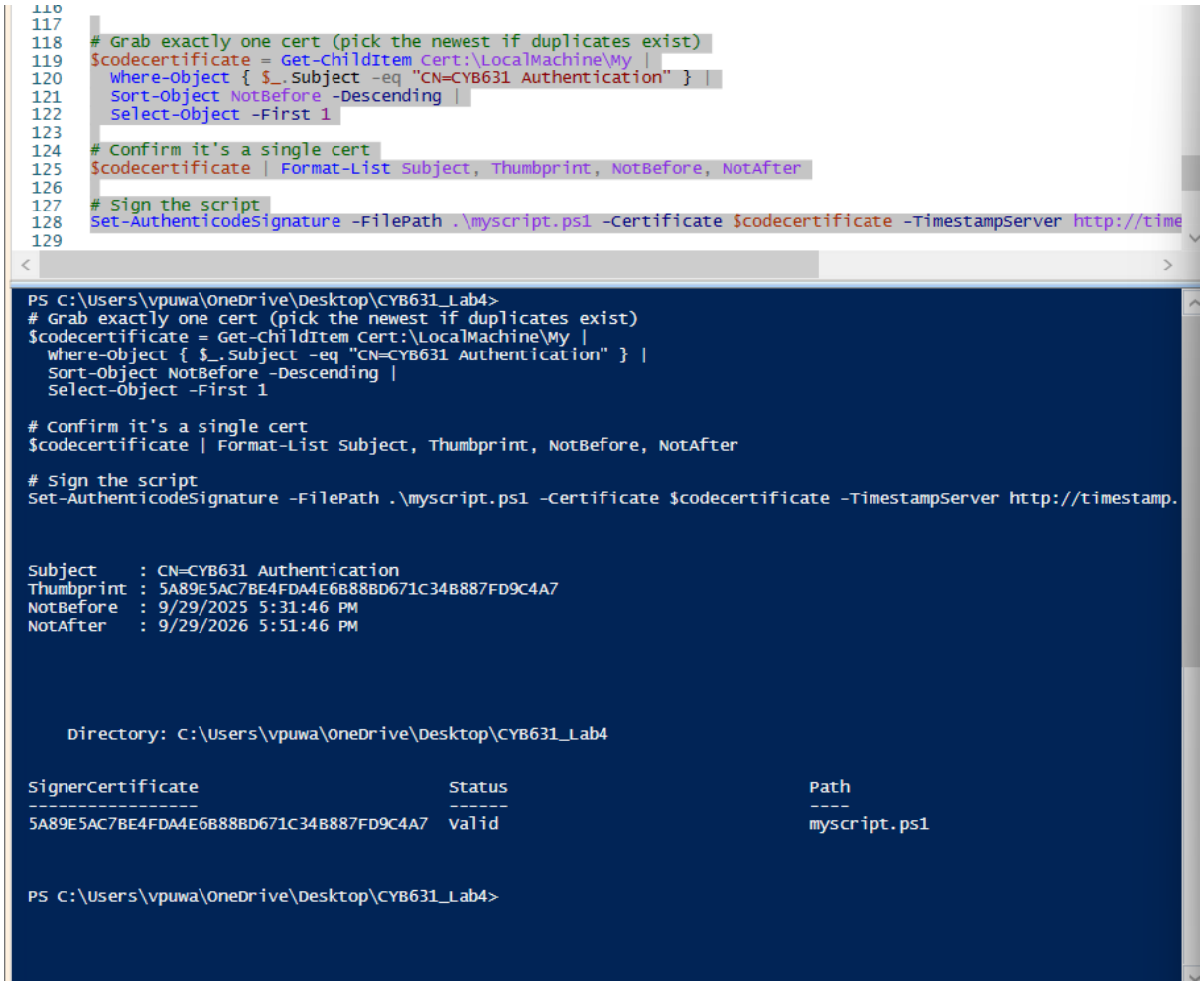
.\myscript.ps1



Step 42

We will sign myscript.ps1 with the certificate that we just created. You will need to retrieve it from your own path and then use it to sign. We are using DigiCert's time server for

stamping the certificate. Since DigiCert's time server is on the Internet, you will need Internet access for this script to work.



```
116
117
118 # Grab exactly one cert (pick the newest if duplicates exist)
119 $codecertificate = Get-ChildItem Cert:\LocalMachine\My |
120   Where-Object { $_.Subject -eq "CN=CYB631 Authentication" } |
121   Sort-Object NotBefore -Descending |
122   Select-Object -First 1
123
124 # Confirm it's a single cert
125 $codecertificate | Format-List Subject, Thumbprint, NotBefore, NotAfter
126
127 # Sign the script
128 Set-AuthenticodeSignature -FilePath .\myscript.ps1 -Certificate $codecertificate -TimestampServer http://time
129
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
# Grab exactly one cert (pick the newest if duplicates exist)
$codecertificate = Get-ChildItem Cert:\LocalMachine\My |
  Where-Object { $_.Subject -eq "CN=CYB631 Authentication" } |
  Sort-Object NotBefore -Descending |
  Select-Object -First 1

# Confirm it's a single cert
$codecertificate | Format-List Subject, Thumbprint, NotBefore, NotAfter

# Sign the script
Set-AuthenticodeSignature -FilePath .\myscript.ps1 -Certificate $codecertificate -TimestampServer http://timestamp.



Subject    : CN=CYB631 Authentication
Thumbprint : 5A89E5AC7BE4FDA4E6B88BD671C34B887FD9C4A7
NotBefore  : 9/29/2025 5:31:46 PM
NotAfter   : 9/29/2026 5:51:46 PM




    Directory: C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4


SignerCertificate                         Status                         Path
-----------------                         ------                         ----
5A89E5AC7BE4FDA4E6B88BD671C34B887FD9C4A7  Valid                          myscript.ps1


PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 22

Step 43

View the signature block

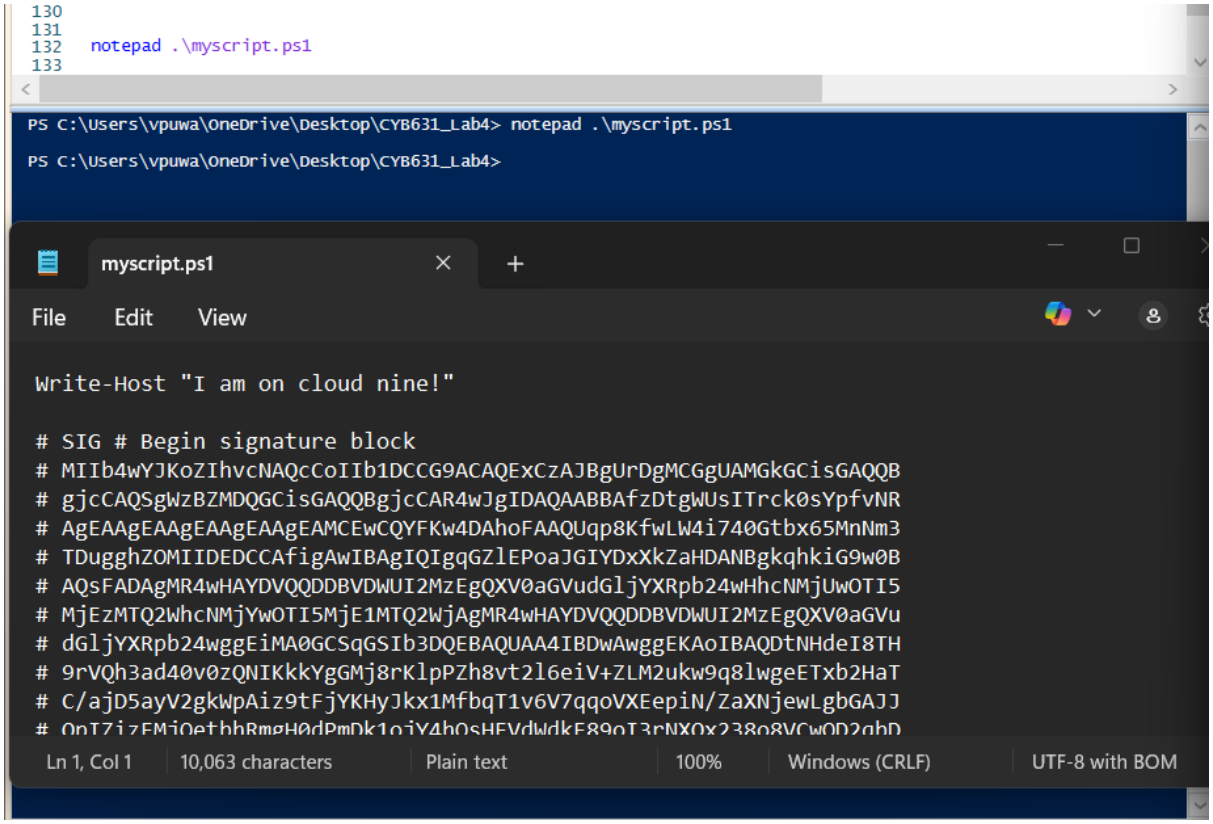


Figure 23

Step 43

Screenshot available in the above figure.

Step 44

Paste a screenshot of the script including its original script (Write-Host "I am on cloud nine!") and the first five lines of the signature block.

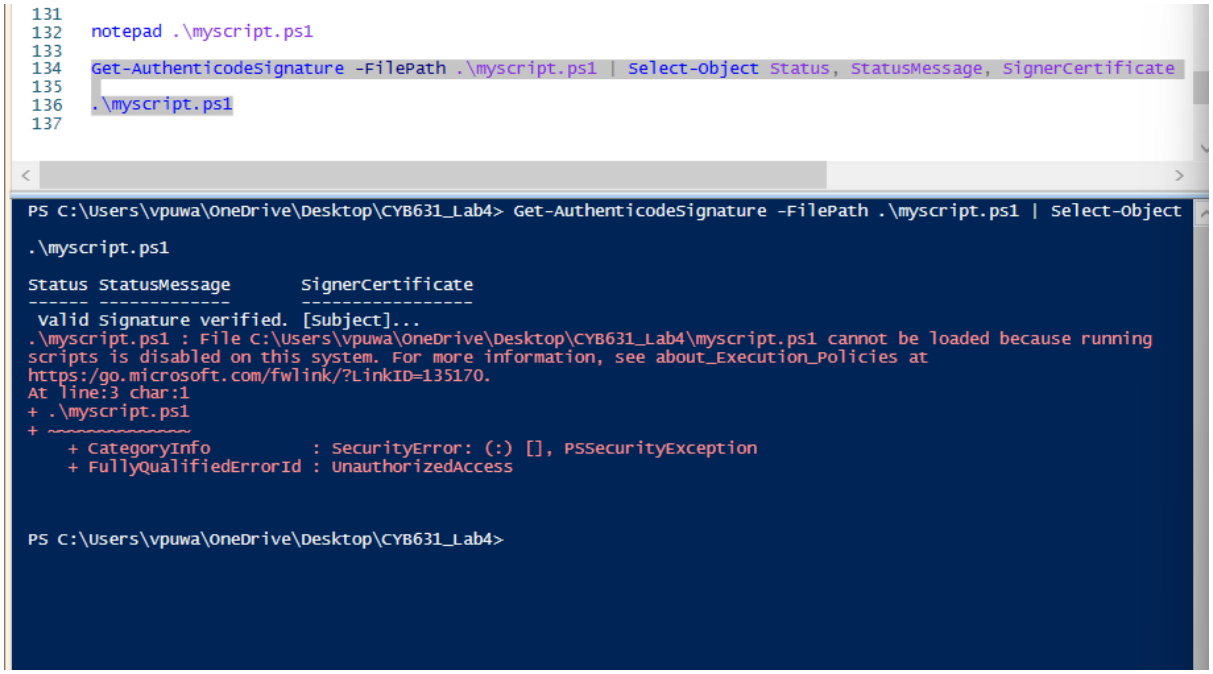


Figure 24

Step 44

Validate the signature of the file.

```
138
139  # See what's enforcing the block
140  Get-ExecutionPolicy -List
141
142  # Allow only signed scripts for THIS PowerShell session (safest for the lab)
143  Set-ExecutionPolicy -Scope Process -ExecutionPolicy AllSigned -Force
144
145  # (If the file was downloaded or copied from the internet, clear Zone.Identifier)
146  Unblock-File .\myscript.ps1
147
148  # Run again
149  .\myscript.ps1
150
151
152  powershell.exe -NoProfile -ExecutionPolicy Bypass -File .\myscript.ps1
153
154
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> # See what's enforcing the block
Get-ExecutionPolicy -List

# Allow only signed scripts for THIS PowerShell session (safest for the lab)
Set-ExecutionPolicy -Scope Process -ExecutionPolicy AllSigned -Force

# (If the file was downloaded or copied from the internet, clear Zone.Identifier)
Unblock-File .\myscript.ps1

# Run again
.\myscript.ps1


powershell.exe -NoProfile -ExecutionPolicy Bypass -File .\myscript.ps1



I am on cloud nine!
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser      Restricted
 LocalMachine       Undefined
I am on cloud nine!


PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 25

Step 46

un myscript.ps1 again. It should work this time since it is now signed.

.\myscript.ps1

➡️ *Explanation:* If your Execution Policy is strict, temporarily allow signed scripts only:

Set-ExecutionPolicy -Scope Process -ExecutionPolicy AllSigned -Force

Unblock-File .\myscript.ps1

.\myscript.ps1

Or, as a fallback for testing:

powershell.exe -NoProfile -ExecutionPolicy Bypass -File .\myscript.ps1

Step 47

The result for running is here.

```
138
139  # See what's enforcing the block
140  Get-ExecutionPolicy -List
141
142  # Allow only signed scripts for THIS PowerShell session (safest for the lab)
143  Set-ExecutionPolicy -Scope Process -ExecutionPolicy AllSigned -Force
144
145  # (If the file was downloaded or copied from the internet, clear Zone.Identifier)
146  Unblock-File .\myscript.ps1
147
148  # Run again
149  .\myscript.ps1
150
151
152  powershell.exe -NoProfile -ExecutionPolicy Bypass -File .\myscript.ps1
153
154
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> # See what's enforcing the block
Get-ExecutionPolicy -List

# Allow only signed scripts for THIS PowerShell session (safest for the lab)
Set-ExecutionPolicy -Scope Process -ExecutionPolicy AllSigned -Force

# (If the file was downloaded or copied from the internet, clear Zone.Identifier)
Unblock-File .\myscript.ps1

# Run again
.\myscript.ps1


powershell.exe -NoProfile -ExecutionPolicy Bypass -File .\myscript.ps1



I am on cloud nine!
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser      Restricted
 LocalMachine       Undefined
I am on cloud nine!


PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 26

Exercise VI: Encryption and Decryption using a Certificate

Step 48

Digital certificates facilitate public key encryption. Protect-CmsMessage uses public key encryption. The public key is contained in the certificate.

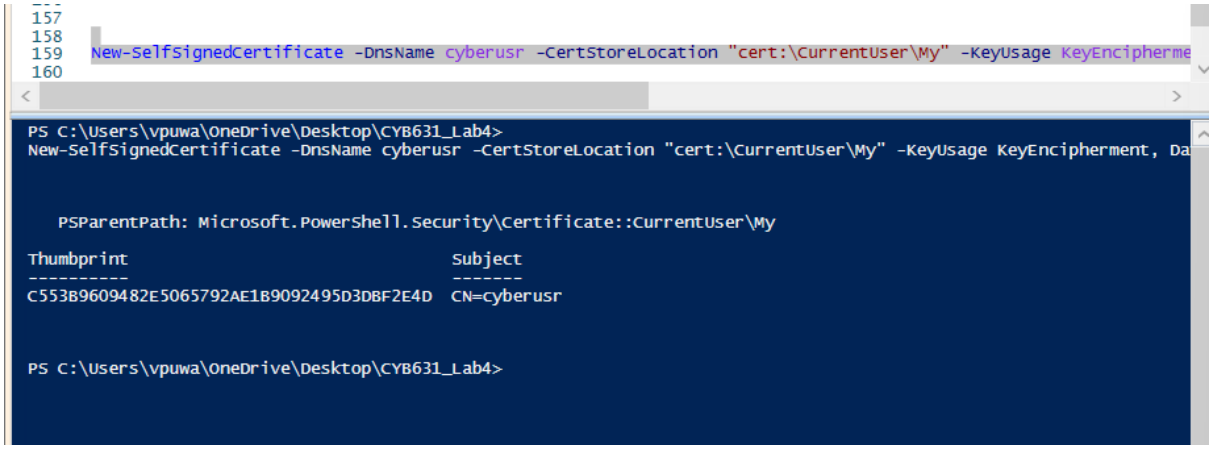


Figure 27

Step 49

You can view the certificate by the following.

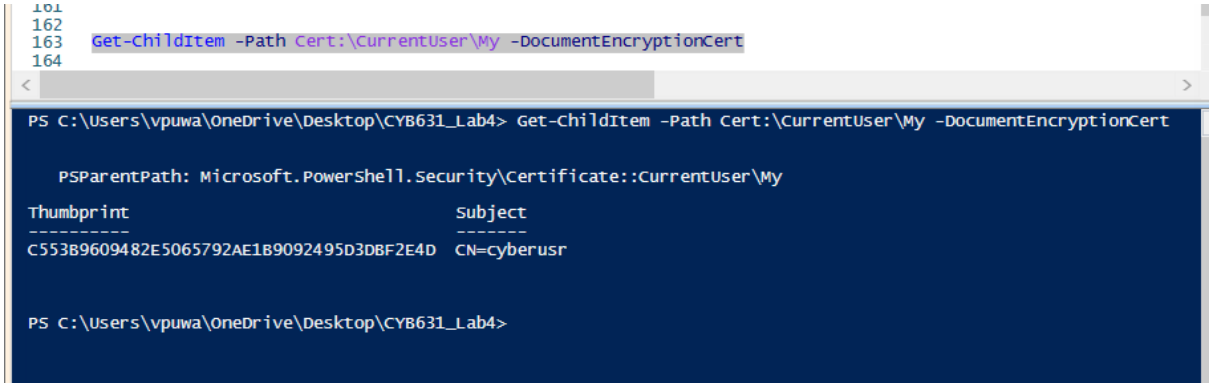


Figure 28

Step 50

Encrypt a message and store it in a file p1.txt. The message is encrypted using the public key stored in the certificate.

```
164
165  "This is a secret!" | Protect-CmsMessage -To CN=cyberusr -OutFile p1.txt
166
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> "This is a secret!" | Protect-CmsMessage -To CN=cyberusr -OutFile p

PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> LS


    Directory: C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         9/29/2025   1:54 PM             16 keyfile.bin
-a----         9/29/2025   1:45 PM            144 keyfile.txt
-a----         9/29/2025   5:59 PM          10219 myscript.ps1
-a----         9/29/2025   6:41 PM            630 p1.txt
-a----         9/29/2025   5:17 PM            276 password.txt
-a----         9/29/2025   1:54 PM            276 secret.enc


PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 29

Step 51

Examine p1.txt. *Paste a screenshot of the file to show the first 5 lines.*

```
-----BEGIN CMS-----
MIIBqwYJKoZIhvcNAQcDoIIBnDCCAZgCAQAxggFDMIIBPwIBADAnMBMxETAPBgNVBAMMCGN5YmVy
dXNyAhBnGlnmIYU5rUu68TR9b+1gMA0GCSqGSIb3DQEBBzAABIIBALSj8vjyY0NsmPq9AKXIOx3u
GPnBEE52oe+0kJITtLmePjTVPBL6DGZdHiWZo/TuKsmtM1weI7GfssRfaZQMSCtMwUwMdAcy0UPl
i2sl4jWUpHqeB5Dcsu+KoY5CLavKvw9Cp3osDu+gx0imw2MOE8osozGeHTtnWVBJ/tmLjelAiY0w
uCXcMEBVUX6JuTAEy7GFBIjYb5drRrtJy5SFBLEXg/BG83TrxgKRDNoDl48MYh83B+mgxJyLjEZp
/+22q6W91ZCtjHRJgqw1hCUF1z4xusM5Jq4Q+slG/5MNGqchbSz9uN0LVBDjj64Ihok1VGvV3hXx
Dxn4j9HsnK8zzGowTAYJKoZIhvcNAQcBMB0GCWCGSAFlAwQBKgQQh+cgEpHFpdld+XKe4e7oLYAg
zxAHhz8Wgau9lse+/IWMYgfke1xwWD+B5WFE2mcFGso=
-----END CMS-----
```

Figure 30

Step 52

To decrypt the file, you will need to log in with the user credential that created the certificate and has the private key (the password for the account).

```
171
172   Unprotect-CmsMessage -Path p1.txt
173
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> Unprotect-CmsMessage -Path p1.txt
This is a secret!

PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 31

Step 53

For others (other users or other hosts) to send you an encrypted file, you will need to export your certificate for others to import. There is a set of PKI cmdlets that can be used to manage digital certificates. *(Reference: Microsoft Learn PKI cmdlets)*

Step 54

```
166
167
168   $encCert = Get-ChildItem Cert:\CurrentUser\My -DocumentEncryptionCert | Where-Object { $_.Subject -eq "CN=cyb
169   Export-Certificate -Cert $encCert -FilePath .\cyberusr.cer
170
```

```
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
$encCert = Get-ChildItem Cert:\CurrentUser\My -DocumentEncryptionCert | Where-Object { $_.Subject -eq "CN=cyberusr"
Export-Certificate -Cert $encCert -FilePath .\cyberusr.cer


    Directory: C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         9/29/2025   6:45 PM            784 cyberusr.cer


PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
```

Figure 32

Step 55

Paste a snapshot of the content in the certificate file exported above to show that you completed it.

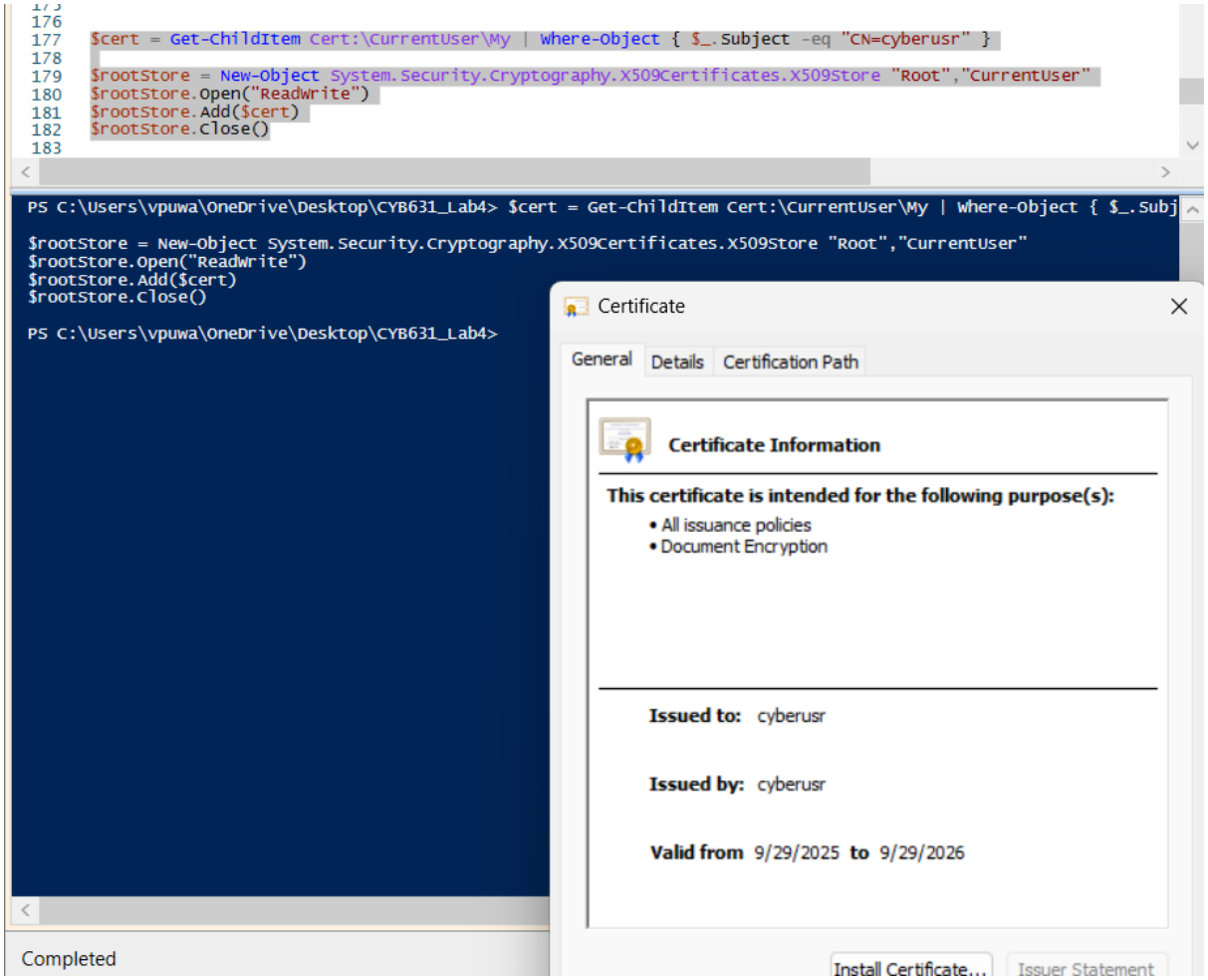

$cert = Get-ChildItem Cert:\CurrentUser\My | Where-Object { $_.Subject -eq "CN=cyberusr" }
$rootStore = New-Object System.Security.Cryptography.X509Certificates.X509Store "Root","CurrentUser"
$rootStore.Open("ReadWrite")
$rootStore.Add($cert)
$rootStore.Close()
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4> $cert = Get-ChildItem Cert:\CurrentUser\My | Where-Object { $_.Subj
$rootStore = New-Object System.Security.Cryptography.X509Certificates.X509Store "Root","CurrentUser"
$rootStore.Open("ReadWrite")
$rootStore.Add($cert)
$rootStore.Close()
PS C:\Users\vpuwa\OneDrive\Desktop\CYB631_Lab4>
Completed
Certificate
General
Details
Certification Path
Certificate Information
This certificate is intended for the following purpose(s):
• All issuance policies
• Document Encryption
Issued to: cyberusr
Issued by: cyberusr
Valid from 9/29/2025 to 9/29/2026
Install Certificate...
Issuer Statement

Figure 33

Exercise VII: Lab and Class Reflection

What did you like about this lab?

I liked that it connected core confidentiality concepts to real, hands-on PowerShell tasks. Hashing files with Get-FileHash, verifying vendor downloads, working with SecureString and explicit AES keys, signing scripts with a self-signed **Code Signing** cert, and using CMS (Protect-CmsMessage / Unprotect-CmsMessage) made the theory tangible. The evidence-driven workflow (commands → screenshot → one-line takeaway) also mirrors how we'd document in a SOC or audit.

What were the challenges?

- Managing certificates across the correct stores (CurrentUser\My vs Root) and choosing the right cert **type** (CodeSigning vs DocumentEncryptionCert).
- Execution policy and tooling on Windows Server 2022 (PowerShell ISE not installed by default; needing VS Code or adding the ISE feature).
- Consistency of key material (binary vs text) for ConvertFrom-SecureString -Key/ConvertTo-SecureString -Key.
- CMS recipient selection (matching CN exactly) and understanding that CMS encrypts to the **public key**, so decryption must use the original user profile/private key.
- Small but time-consuming issues: path/permission/UAC prompts, and aligning screenshots with clear captions.

Suggestions to help me learn better:

- Provide a short "pre-flight checklist" (tooling, execution policy, cert store notes, timestamp server URL).

- Include one end-to-end example per section (hashing → compare, CMS → encrypt/decrypt, signing → verify) with a model screenshot + caption so expectations are crystal clear.
- Ship a tiny sample dataset (3–5 files) for hashing and a known safe download link + published hash to avoid web hunting.
- Recommend VS Code as the default editor on Server 2022 and note how to install ISE if required.
- Add a troubleshooting appendix (common CMS errors, cert selection tips, key length rules for AES, “signature status: Unknown” fixes).
- Optional: a cleanup script to remove generated keys/certs/files after submission.

VM power-down

Acknowledged. I’ve powered off my Windows VM on the server after completing and verifying all screenshots.