PowerShell - szyfrowanie AES

Wdrożenie szyfrowania AES za pomocą PowerShell umożliwia ochronę ciągu hasłem.

Załaduj AesEncryption.psm1 dostępne w folderze z zadaniem

1. Wykonaj przykład szyfrowania

Utwórz hasło jako bezpieczny ciąg i zaszyfruj je.

Bezpieczne wczytanie hasła:

```
$password = Read-Host -AsSecureString
```

Ta linia kodu powoduje wyświetlenie użytkownikowi polecenia wprowadzenia hasła w sposób bezpieczny. Parametr -AsSecureString zapewnia, że wprowadzone hasło jest przechowywane jako bezpieczny ciąg znaków, co stanowi bardziej bezpieczny sposób przechowywania wrażliwych informacji, takich jak hasła, w PowerShell.

Szyfrowanie ciągu znaków:

```
$encrypted = Protect-AesString -String 'Zaszyfruj mnie!_!' -Password $password
```

Kod używa niestandardowej funkcji Protect-AesString, aby zaszyfrować ciąg znaków 'Zaszyfruj mnie!_!' za pomocą podanego hasła. Funkcja ta prawdopodobnie używa zaawansowanego standardu szyfrowania AES (Advanced Encryption Standard). Jednak konkretne szczegóły implementacji nie są dostarczone w tym fragmencie kodu.

Wyświetlanie tekstu szyfrogramu:

```
$encrypted.CipherText
```

```
PS C:\Windows\system32> $encrypted.CipherText
z9KWMO7L+jdOzFNY42Wmkv8jEnqjZmjbGh1dOW9nuQs=
```

Ta linia kodu wyświetla tekst szyfrogramu wygenerowany podczas procesu szyfrowania. W tym przypadku właściwy tekst szyfrogramu.

2. Wykonaj przykład deszyfrowania

Odszyfruj tekst za pomocą tego samego hasła.

Deszyfrowanie tekstu szyfrogramu:

```
$encrypted | Unprotect-AesString -Password $password
```

Ta linia kodu deszyfruje tekst szyfrogramu, który został wcześniej zaszyfrowany. Używana niestandardowa funkcja Unprotect-AesString, która dokonuje deszyfrowania przy użyciu algorytmu AES.

Zwracanie oryginalnego tekstu:

Po deszyfrowaniu tekst oryginalny zostaje zwrócony i może być używany dalej w skrypcie lub wyświetlany użytkownikowi. W tym konkretnym przypadku oryginalny tekst to

```
Zaszyfruj mnie!_!
```

Możesz przejść do kodu źródłowego (lub znaleźć go na Github) i przeanalizować go i zrozumieć.

Analiza

Musze wygenerować klucz do użycia do szyfrowania.

Jak wspomniano powyżej, chcę użyć hasła do ochrony mojego ciągu.

Musze wygenerować klucz za pomocą hasła.

3. Wydobądź (wyprowadź) klucz z hasła

W tym celu możemy wykorzystać klasę Rfc2898DeriveBytes w .Net. Do zbudowania tej klasy będziemy potrzebować ciągu, hasła i kilku innych rzeczy, jeśli potrzebna jest większa personalizacja.

Ciąg musi być reprezentowany w tablicy bajtów

```
# Musi mieć co najmniej 8 bajtów długości
```

Definiowanie:

```
$salt = "Klucz123"
```

Powyższa linia kodu przypisuje do zmiennej $salt ciąg znaków 'Klucz123', który będzie używany jako $salt. $salt to losowy lub pseudolosowy ciąg znaków, który jest dodawany do danych przed ich haszowaniem lub szyfrowaniem. Jest to używane do wprowadzenia dodatkowego elementu losowości, co utrudnia atakom typu "ataki słownikowe" lub "ataki na szyfrowanie o tym samym tekście".

Haszowanie z solą (ciąg zaburzający) to proces, w którym dodajemy losowy i niepowtarzalny ciąg znaków do hasła przed jego shaszowaniem. Dzięki temu każdy skrót hasło jest niepowtarzalny, nawet jeśli każdy użytkownik w systemie ma to samo hasło w formie tekstowej.

Konwertowanie soli na tablicę bajtów:

$saltBytes = [Text.Encoding]::UTF8.GetBytes($salt)

Ta linia kodu konwertuje ciąg znaków soli na tablicę bajtów. W tym przypadku używana jest metoda GetBytes z klasy [Text.Encoding]::UTF8, która przekształca znaki z kodowania UTF-8 na odpowiadające im bajty. Tablica bajtów $saltBytes będzie zawierała reprezentację binarną soli.

W praktyce, $salt jest używana w celu zabezpieczenia przed atakami, takimi jak ataki przy użyciu tęczowych tablic (ang. "rainbow tables") i ataki przy użyciu ogromnych zbiorów haseł. $salt dodawana do hasła przed jego haszowaniem sprawia, że nawet dla tych samych haseł otrzymuje się różne wyniki haszowania, co utrudnia odtworzenie oryginalnych haseł za pomocą gotowych tablic tęczowych.

Uzbrojeni w nasze bajty ciągu, możemy uzyskać nasze bajty hasła.

Definiowanie hasła:

$password = 'P@ssw0rd1'

Ta linia kodu przypisuje do zmiennej $password ciąg znaków 'P@ssw0rd1', który stanowi hasło, na podstawie którego będzie generowany klucz.

Tworzenie obiektu Rfc2898DeriveBytes:

$passDerive = New-Object Security.Cryptography.Rfc2898DeriveBytes -ArgumentList @($password, $saltBytes)

Ta linia kodu tworzy obiekt klasy Rfc2898DeriveBytes, która jest często używana do pochodzenia klucza z hasła. Konstruktor tej klasy przyjmuje dwa argumenty: hasło ($password) i $salt ($saltBytes).

Rfc2898DeriveBytes implementuje algorytm PBKDF2 (Password-Based Key Derivation Function 2), który jest powszechnie stosowany do bezpiecznego pochodzenia kluczy na podstawie haseł. Algorytm ten jest zalecany, ponieważ wykonuje wiele iteracji funkcji haszującej, co utrudnia atakom brute-force i atakom przy użyciu tęczowych tablic.

Po utworzeniu obiektu $passDerive, można go użyć do generowania kluczy pochodnych, na przykład za pomocą metody GetBytes().

Powyższy kod stanowi jeden ze sposobów na generowanie bezpiecznych kluczy pochodnych z hasła i soli w celu późniejszego ich wykorzystania, na przykład w kontekście haszowania haseł lub szyfrowania. Jeśli spojrzysz na zmienną $passDerive, zobaczysz, że domyślnym algorytmem mieszania jest SHA1, a liczba iteracji to 1000. Można to dostosować podczas konstruowania obiektu.

Określ algorytm skrótu SHA256 i liczbę interakcji w następujący sposób:

Definiowanie liczby iteracji:

$iterations = 1000

Ta linia kodu przypisuje do zmiennej $iterations liczbę iteracji. Liczba iteracji określa, jak wiele razy algorytm PBKDF2 będzie powtarzany, co wpływa na złożoność obliczeniową i bezpieczeństwo procesu pochodzenia klucza. Większa liczba iteracji zazwyczaj oznacza bardziej bezpieczny proces, ale wymaga więcej zasobów obliczeniowych.

Aktualizacja obiektu Rfc2898DeriveBytes z dodatkowymi parametrami:

$passDerive = New-Object Security.Cryptography.Rfc2898DeriveBytes -ArgumentList @($password, $saltBytes, $iterations, 'SHA256')

Ta linia kodu tworzy nowy obiekt klasy Rfc2898DeriveBytes, tym razem z trzema dodatkowymi parametrami:

$iterations: Liczba iteracji algorytmu PBKDF2.

'SHA256': Specyfikacja algorytmu haszowania używanego w procesie pochodzenia klucza. W tym przypadku używany jest algorytm SHA-256. PBKDF2 może być używane z różnymi algorytmami haszowania, a wybór SHA-256 to dobry standard z uwagi na bezpieczeństwo.

Dodanie liczby iteracji i określenie konkretnego algorytmu haszowania zwiększa bezpieczeństwo procesu pochodzenia klucza.
Warto dostosować te parametry w zależności od konkretnych wymagań bezpieczeństwa i zasobów dostępnych na danym systemie.

Wyciągnij klucz z tej klasy.

Definiowanie rozmiaru klucza:

$keySize = 256

Ta linia kodu przypisuje do zmiennej $keySize wartość 256, co oznacza, że oczekujemy klucza o długości 256 bitów.

$key = $passDerive.GetBytes($keySize / 8)

Ta linia kodu wywołuje metodę GetBytes() na obiekcie $passDerive, aby uzyskać bajty klucza o określonym rozmiarze.

Parametr $keySize / 8 konwertuje rozmiar klucza z bitów na bajty.

W rezultacie $key zawiera teraz klucz pochodzący z procesu pochodzenia klucza (w tym przypadku z algorytmu PBKDF2 z użyciem SHA-256, soli i liczby iteracji) o długości 256 bitów.

Ten klucz może być używany w różnych zastosowaniach, takich jak szyfrowanie danych, podpisywanie cyfrowe czy haszowanie. Warto pamiętać, że klucz ten powinien być przechowywany i zarządzany w sposób bezpieczny, zgodnie z zasadami bezpieczeństwa.

Mamy klucz wygenerowany z naszego hasła.

Możesz wygenerować klucz, używając surowego hasła, ale nie jest to bezpieczny sposób na zrobienie tego. Używając tej klasy wprowadzamy entropię, aby proces był kryptograficznie poprawny.

4. Szyfruj

W .Net są łatwe w użyciu klasy do obsługi trudnych rzeczy. Możemy wykorzystać klasę SymmetricAlgorithm do stworzenia obiektu umożliwiającego szyfrowanie.

Użycie tej klasy pozwala również na tworzenie różnych obiektów do zarządzania różnymi algorytmami (np. TripleDES, RC2 itp.).

Tworzenie obiektu klasy AesManaged , która jest implementacją algorytmu szyfrowania symetrycznego AES (Advanced Encryption Standard).:

```
$cipher = [Security.Cryptography.SymmetricAlgorithm]::Create('AesManaged')
```

Ta linia kodu używa statycznej metody Create klasy SymmetricAlgorithm z przestrzeni nazw Security.Cryptography. W tym przypadku, jako parametr, przekazuje ciąg znaków 'AesManaged', co powoduje utworzenie obiektu klasy AesManaged.

Algorytm AES jest jednym z najbardziej powszechnie stosowanych algorytmów szyfrowania symetrycznego. Wersja zarządzana (AesManaged) jest jedną z implementacji algorytmu AES dostępnych w .NET Framework. Implementacje zarządzane są bardziej intuicyjne do użycia, ale mogą być wolniejsze niż ich odpowiedniki w wersji niezarządzanej (AesCryptoServiceProvider).

Wyświetlanie obiektu klasy AesManaged:

```
Write-Output $cipher
```

Ta linia kodu wyświetla informacje o utworzonym obiekcie klasy AesManaged. To obejście jest przydatne do zrozumienia domyślnych ustawień obiektu i parametrów algorytmu AES, które zostały ustawione podczas jego utworzenia.

W praktyce, obiekt klasy AesManaged jest używany do szyfrowania i deszyfrowania danych za pomocą algorytmu AES. Po utworzeniu obiektu, konieczne jest skonfigurowanie go, ustawienie klucza (który może pochodzić z poprzedniego procesu pochodzenia klucza) i określenie innych parametrów, zanim będzie można go użyć do operacji szyfrowania i deszyfrowania.

Zauważysz, że istnieje już utworzony klucz, wraz z domyślnymi wartościami rozmiaru klucza, trybu i dopełnienia. Dodatkowo zobaczysz właściwość IV (wektor inicjujący). Korzystając z trybu szyfrowania CBC (którego używamy), najlepiej jest generować unikalne IV za każdym

razem, gdy coś szyfrujemy. Na szczęście ta klasa generuje jeden za każdym razem, gdy jest inicjowana, ale będziemy musieli wiedzieć, czym jest IV, aby ją odszyfrować.

Teraz użyję tej rzeczy do zaszyfrowania niektórych danych. Stworzymy obiekt szyfrujący za pomocą naszego klucza z góry i wstępnie wygenerowanego IV. Kod PowerShell przygotowuje obiekty potrzebne do szyfrowania danych za pomocą algorytmu AES w trybie CBC (Cipher Block Chaining).

Pobieranie wektora inicjalizacyjnego (IV):

```
$iv = $cipher.IV
```

Ta linia kodu pobiera wektor inicjalizacyjny (IV) z obiektu klasy AesManaged. IV jest używany w trybie CBC do inicjalizacji pierwszego bloku danych przed szyfrowaniem.

Tworzenie obiektu ICryptoTransform (Encryptor):

```
$encryptor = $cipher.CreateEncryptor($key, $iv)
```

Ta linia kodu tworzy obiekt implementujący interfejs ICryptoTransform, który służy do przekształcania danych (w tym przypadku do szyfrowania). CreateEncryptor jest metodą obiektu klasy AesManaged, która tworzy odpowiedni obiekt do przeprowadzania operacji szyfrowania na podstawie dostarczonego klucza i wektora inicjalizacyjnego.

Tworzenie obiektu MemoryStream:

```
$memoryStream = New-Object -TypeName IO.MemoryStream
```

Ta linia kodu tworzy obiekt strumienia pamięci (MemoryStream), który będzie używany do przechowywania wyniku operacji szyfrowania.

Tworzenie obiektu CryptoStream:

```
$cryptoStream = New-Object -TypeName Security.Cryptography.CryptoStream -ArgumentList @( $memoryStream, $encryptor, 'Write' )
```

Ta linia kodu tworzy obiekt strumienia kryptograficznego (CryptoStream), który jest używany do przekazywania danych do algorytmu szyfrowania. Parametry konstruktora to: strumień docelowy ($memoryStream), obiekt implementujący ICryptoTransform ($encryptor), oraz tryb operacji (w tym przypadku 'Write', co oznacza, że strumień będzie używany do zapisu danych).

Po przygotowaniu tych obiektów, można użyć $cryptoStream do przesyłania danych do algorytmu AES w trybie CBC, co skutkuje zaszyfrowaniem tych danych. Ostatecznie wynikowa postać zaszyfrowanych danych będzie przechowywana w strumieniu pamięci ($memoryStream).

Mamy teraz obiekt strumienia kryptograficznego, gotowy z naszym kluczem do zapisania w naszym strumieniu pamięci.

Szyfrujemy.

Poniższy kod PowerShell służy do zaszyfrowania ciągu znaków 'Hello, World!' za pomocą wcześniej przygotowanego obiektu CryptoStream i algorytmu AES w trybie CBC.

Konwersja ciągu znaków na tablicę bajtów w kodowaniu UTF-8:

```
$string = 'Hello, World!'

$strBytes = [Text.Encoding]::UTF8.GetBytes($string)
```

Te linie kodu konwertują ciąg znaków 'Hello, World!' na tablicę bajtów, używając kodowania UTF-8. Wynikowy byte[] przechowywany jest w zmiennej $strBytes.

Zapisywanie bajtów do strumienia kryptograficznego (CryptoStream):

```
$cryptoStream.Write($strBytes, 0, $strBytes.Length)
```

Ta linia kodu zapisuje tablicę bajtów ($strBytes) do strumienia kryptograficznego ($cryptoStream). Oznacza to, że dane są przesyłane do algorytmu AES w trybie CBC w celu zaszyfrowania.

Przesyłanie bloku końcowego (FlushFinalBlock):

```
$cryptoStream.FlushFinalBlock()
```

Ta linia kodu przesyła blok końcowy do strumienia kryptograficznego, co jest istotne w przypadku trybów szyfrowania blokowego, takich jak CBC. Blok końcowy zawiera dodatkowe informacje potrzebne do poprawnego zakończenia operacji szyfrowania.

Pobieranie zaszyfrowanych bajtów z strumienia pamięci (MemoryStream):

```
$encryptedBytes = $memoryStream.ToArray()
```

Ta linia kodu pobiera zaszyfrowane bajty z strumienia pamięci ($memoryStream) i przypisuje je do zmiennej $encryptedBytes.
Teraz zaszyfrowane dane są dostępne jako tablica bajtów i mogą być używane lub przechowywane w zależności od potrzeb.

W rezultacie po wykonaniu tego kodu, zmienna $encryptedBytes zawiera zaszyfrowaną postać ciągu znaków 'Hello, World!' przy użyciu algorytmu AES w trybie CBC z wcześniej przygotowanym kluczem i wektorem inicjalizacyjnym.

Base64 Zakoduj zaszyfrowane bajty, aby uzyskać ciąg

Poniższy kod PowerShell koduje zaszyfrowane bajty w formie ciągu znaków, używając kodowania Base64.

```
$encryptedString = [Convert]::ToBase64String($encryptedBytes)
```

Ta linia kodu używa statycznej metody ToBase64String z klasy [Convert], aby zakodować zaszyfrowane bajty (przechowywane w zmiennej $encryptedBytes) w ciąg znaków Base64. Kodowanie Base64 przekształca binarną reprezentację danych na ciąg znaków, co jest przydatne do przechowywania danych binarnych w formie tekstowej.

```
Write-Output $encryptedString
```

Ta linia kodu wyświetla zakodowany ciąg Base64 ($encryptedString). Zakodowany ciąg może być teraz przechowywany, przesyłany lub używany w inny sposób w zależności od wymagań aplikacji.

W rezultacie po wykonaniu tego kodu, zmienna $encryptedString zawiera zakodowany ciąg Base64, który reprezentuje zaszyfrowane dane uzyskane wcześniej. Zakodowany ciąg Base64 jest często używany do przechowywania lub przesyłania danych binarnych w formie tekstowej, na przykład w przypadku zaszyfrowanych danych.

```
PS C:\Windows\system32> Write-Output $encryptedString
wcHRd9c/k7SQsXvPpIGppA==
```

Możesz ponownie uruchomić kod (zaczynając od wygenerowania nowego szyfru), a otrzymasz zupełnie inny zaszyfrowany ciąg.

5. Odszyfruj

Ponownie użyjemy szyfru, który wygenerowaliśmy powyżej, ponieważ ma już potrzebne nam właściwości. Proces odszyfrowywania jest podobny do tego używanego do szyfrowania. Aby utworzyć deszyfrator, będziemy również potrzebować klucza, który wygenerowaliśmy powyżej. Poniższy kod PowerShell służy do deszyfrowania danych, które zostały wcześniej zaszyfrowane przy użyciu algorytmu AES w trybie CBC.

Tworzenie obiektu ICryptoTransform (Decryptor):

```
$decryptor = $cipher.CreateDecryptor($key, $iv)
```

Ta linia kodu tworzy obiekt implementujący interfejs ICryptoTransform, który jest używany do przeprowadzania operacji deszyfrowania. Metoda CreateDecryptor obiektu klasy AesManaged tworzy odpowiedni obiekt do deszyfrowania na podstawie dostarczonego klucza i wektora inicjalizacyjnego.

```
# Pobierz bajty z zaszyfrowanego ciągu
```

Konwersja zakodowanego ciągu Base64 na bajty:

```
$bytes = [Convert]::FromBase64String($encryptedString)
```

Ta linia kodu przekształca zakodowany ciąg Base64 ($encryptedString) z powrotem na bajty. Otrzymane bajty są przechowywane w zmiennej $bytes.

```
# Utwórz strumień pamięci i krypto
```

Utworzenie strumienia pamięci i strumienia kryptograficznego do odczytu:

```
$stream = New-Object -TypeName IO.MemoryStream -ArgumentList @(, $encryptedBytes)

$cryptoReader = New-Object -TypeName Security.Cryptography.CryptoStream -ArgumentList @( $stream, $decryptor, 'Read' )
```

Te linie kodu tworzą obiekt MemoryStream ($stream), który jest używany do przechowywania zaszyfrowanych danych, a także obiekt CryptoStream ($cryptoReader), który jest używany do odczytu danych z obiektu MemoryStream przy użyciu deszyfratora.

Jesteśmy teraz skonfigurowani do odczytu ze strumienia krypto za pomocą naszego deszyfratora.

Odczytywanie danych z strumienia kryptograficznego:

```
$decrypted = New-Object -TypeName Byte[] -ArgumentList $bytes.Length

$decryptedByteCount = $cryptoReader.Read($decrypted, 0, $decrypted.Length)
```

Te linie kodu odczytują zaszyfrowane dane z obiektu CryptoStream i umieszczają je w tablicy bajtów $decrypted. Liczba odczytanych bajtów jest przechowywana w zmiennej $decryptedByteCount.

Konwersja bajtów na deszyfrowany ciąg znaków:

```
$decryptedString = [Text.Encoding]::UTF8.GetString($decrypted, 0, $decryptedByteCount)
```

Ta linia kodu konwertuje odczytane bajty na ciąg znaków przy użyciu kodowania UTF-8. Wynikowy ciąg znaków ($decryptedString) reprezentuje deszyfrowane dane.

Wyświetlanie deszyfrowanego ciągu znaków:

```
Write-Output $decryptedString
```

Ta linia kodu wyświetla deszyfrowany ciąg znaków. Teraz otrzymujemy oryginalne dane po procesie deszyfrowania.

W rezultacie po wykonaniu tego kodu, zmienna $decryptedString zawiera oryginalny ciąg znaków 'Hello, World!', który został pierwotnie zaszyfrowany i potem pomyślnie deszyfrowany.

```
PS C:\Windows\system32> Write-Output $decryptedString
Hello, World!
```

Masz, szyfrowanie AES i deszyfrowanie w PowerShell. Aby zapoznać się z gotowymi funkcjami, które dokładnie to robią, zobacz poniżej.

Kod

Kod źródłowy jest dostępny na Github i poniżej. Kryptografia jest skomplikowana i zgodna z najlepszymi praktykami.

Powyższy kod PowerShell definiuje dwie funkcje: Protect-AesString i Unprotect-AesString. Te funkcje są przeznaczone do szyfrowania i deszyfrowania ciągów znaków przy użyciu algorytmu AES w trybie CBC. Poniżej znajdziesz wyjaśnienie poszczególnych fragmentów kodu:

Funkcja DecryptSecureString:

```
Function DecryptSecureString
{
  Param(
    [SecureString]$SecureString
  )
  $bstr = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($SecureString)
  $plain = [Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

```
  return $plain
}
```

Ta funkcja przyjmuje obiekt typu SecureString i zwraca odpowiadający mu ciąg znaków. Jest to używane do uzyskania dostępu do wartości przechowywanej w obiekcie SecureString, który zazwyczaj jest używany do przechowywania haseł w sposób bardziej bezpieczny niż zwykłe ciągi znaków.

Funkcja NewPasswordKey:

```
Function NewPasswordKey
{
  [CmdletBinding()]
  Param(
    [SecureString]$Password,
    [String]$Salt
  )
  # ...
}
```

Ta funkcja generuje klucz na podstawie hasła i soli. Wykorzystuje do tego funkcję DecryptSecureString do uzyskania wartości hasła z obiektu SecureString. Następnie używa algorytmu PBKDF2 (Rfc2898DeriveBytes) do pochodzenia klucza na podstawie hasła, soli, liczby iteracji i algorytmu haszowania.

Klasa CipherInfo:

```
Class CipherInfo
{
  [String]$CipherText
```

```
  [Byte[]]$IV
  [String]$Salt


  CipherInfo([String]$CipherText, [Byte[]]$IV, [String]$Salt)
  {
    $this.CipherText = $CipherText
    $this.IV = $IV
    $this.Salt = $Salt
  }
}
```

Ta klasa służy do przechowywania informacji o zaszyfrowanych danych, takich jak zaszyfrowany tekst (CipherText), wektor inicjalizacyjny (IV) i $salt (Salt).

Funkcja Protect-AesString:

```
Function Protect-AesString
{
  [CmdletBinding()]
  Param(
    [String]$String,
    [SecureString]$Password,
    [String]$Salt = 'qtsbp6j643ah8e0omygzwlv9u75xcfrk4j63fdane78w1zgxhucsytkirol0v25q',
    [Security.Cryptography.PaddingMode]$Padding = 'PKCS7'
```

```
  )
  # ...
}
```

Ta funkcja przyjmuje ciąg znaków, hasło, opcjonalną $salt i tryb dopełniania (padding). Wykorzystuje podaną funkcję NewPasswordKey do uzyskania klucza na podstawie hasła i soli. Następnie używa algorytmu AES w trybie CBC do zaszyfrowania tekstu.

Funkcja Unprotect-AesString:

```
Function Unprotect-AesString
{
  [CmdletBinding(DefaultParameterSetName='String')]
  Param(
    # ...
  )
  Process
  {
    Try
    {
      # ...
      if ($PSCmdlet.ParameterSetName -eq 'CipherInfo')
      {
        # ...
      }
```

```
    # ...
    return $decryptedValue
  }
  Catch
  {
    Write-Error $_
  }
 }
}
```

Ta funkcja przyjmuje zaszyfrowany ciąg znaków, hasło, opcjonalną $salt, wektor inicjalizacyjny (IV) lub obiekt CipherInfo, a także tryb dopełniania (padding). Wykorzystuje podaną funkcję NewPasswordKey do uzyskania klucza na podstawie hasła i soli (lub używa danych z obiektu CipherInfo). Następnie używa algorytmu AES w trybie CBC do deszyfrowania tekstu.

Eksportowanie funkcji:

```
Export-ModuleMember -Function Protect-AesString, Unprotect-AesString
```

Ta linia kodu eksportuje funkcje Protect-AesString i Unprotect-AesString jako część modułu PowerShell, co umożliwia ich użycie w innych skryptach lub modułach.

W skrócie, kod ten dostarcza funkcje do bezpiecznego szyfrowania i deszyfrowania danych za pomocą algorytmu AES w trybie CBC, korzystając z bezpiecznego przechowywania haseł w obiektach SecureString oraz dodając obsługę sol

Pełny kod:

```
Function DecryptSecureString
{
 Param(
```

```
    [SecureString]$SecureString
  )
  $bstr = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($SecureString)
  $plain = [Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
  return $plain
}

Function NewPasswordKey
{
  [CmdletBinding()]
  Param(
    [SecureString]$Password,

    [String]$Salt
  )
  $saltBytes = [Text.Encoding]::ASCII.GetBytes($Salt)
  $iterations = 1000
  $keySize = 256

  $clearPass = DecryptSecureString -SecureString $Password
  $passwordType = 'Security.Cryptography.Rfc2898DeriveBytes'
```

```
  $passwordDerive = New-Object -TypeName $passwordType `
    -ArgumentList @(
      $clearPass,
      $saltBytes,
      $iterations,
      'SHA256'
    )

  $keyBytes = $passwordDerive.GetBytes($keySize / 8)
  return $keyBytes
}

Class CipherInfo
{
  [String]$CipherText
  [Byte[]]$IV
  [String]$Salt

  CipherInfo([String]$CipherText, [Byte[]]$IV, [String]$Salt)
  {
    $this.CipherText = $CipherText
```

```
    $this.IV = $IV
    $this.Salt = $Salt
  }
}

<#
.SYNOPSIS
Encrypt a string using the Advanced Encryption Standard (AES).

.DESCRIPTION
Encrypt a string using the Advanced Encryption Standard (AES).

.PARAMETER String
The string to encrypt.

.PARAMETER Password
The password to use to encrypt your string.

.PARAMETER Salt
The salt used for generating the password key.
You must use the same salt for encryption / decryption.
```

```
.PARAMETER Padding
The padding to use for encryption / decryption.
Default is PKCS7.
#>
Function Protect-AesString
{
  [CmdletBinding()]
  Param(
    [Parameter(Position=0, Mandatory=$true, ValueFromPipeline=$true)]
    [String]$String,

    [Parameter(Position=1, Mandatory=$true)]
    [SecureString]$Password,

    [Parameter(Position=2)]
    [String]$Salt = 'qtsbp6j643ah8e0omygzwlv9u75xcfrk4j63fdane78w1zgxhucsytkirol0v25q',

    [Parameter(Position=3)]
    [Security.Cryptography.PaddingMode]$Padding = 'PKCS7'
  )
```

```
  Try
  {
    $valueBytes = [Text.Encoding]::UTF8.GetBytes($String)
    [byte[]]$keyBytes = NewPasswordKey -Password $Password -Salt $Salt

    $cipher = [Security.Cryptography.SymmetricAlgorithm]::Create('AesManaged')
    $cipher.Mode = [Security.Cryptography.CipherMode]::CBC
    $cipher.Padding = $Padding
    $vectorBytes = $cipher.IV

    $encryptor = $cipher.CreateEncryptor($keyBytes, $vectorBytes)
    $stream = New-Object -TypeName IO.MemoryStream
    $writer = New-Object -TypeName Security.Cryptography.CryptoStream `
      -ArgumentList @(
        $stream,
        $encryptor,
        [Security.Cryptography.CryptoStreamMode]::Write
      )

    $writer.Write($valueBytes, 0, $valueBytes.Length)
    $writer.FlushFinalBlock()
```

```
    $encrypted = $stream.ToArray()

    $cipher.Clear()
    $stream.SetLength(0)
    $stream.Close()
    $writer.Clear()
    $writer.Close()
    $encryptedValue = [Convert]::ToBase64String($encrypted)
    New-Object -TypeName CipherInfo `
      -ArgumentList @($encryptedValue, $vectorBytes, $Salt)
  }
  Catch
  {
    Write-Error $_
  }
}

<#
.SYNOPSIS
Decrypt a protected string using the Advanced Encryption Standard (AES).
```

.DESCRIPTION

Decrypt a protected string using the Advanced Encryption Standard (AES).

.PARAMETER String

The string to decrypt.

.PARAMETER Password

The password to use to decrypt your string.

.PARAMETER Salt

The salt used for generating the password key.

You must use the same salt for encryption / decryption.

.PARAMETER InitializationVector

The initialization vector (IV) used for encryption, to use for decryption.

.PARAMETER CipherInfo

The CipherInfo object to use for decryption.

This object is obtained from Protect-AesString.

.PARAMETER Padding

```
The padding to use for encryption / decryption.
Default is PKCS7.
#>
Function Unprotect-AesString
{
  [CmdletBinding(DefaultParameterSetName='String')]
  Param(
    [Parameter(Position=0, Mandatory=$true, ParameterSetName='String')]
    [Alias('EncryptedString')]
    [String]$String,

    [Parameter(Position=1, Mandatory=$true)]
    [SecureString]$Password,

    [Parameter(Position=2, ParameterSetName='String')]
    [String]$Salt = 'qtsbp6j643ah8e0omygzwlv9u75xcfrk4j63fdane78w1zgxhucsytkirol0v25q',

    [Parameter(Position=3, Mandatory=$true, ParameterSetName='String')]
    [Alias('Vector')]
    [Byte[]]$InitializationVector,
```

```
    [Parameter(Position=0, Mandatory=$true, ParameterSetName='CipherInfo', ValueFromPipeline=$true)]
    [CipherInfo]$CipherInfo,

    [Parameter(Position=3, ParameterSetName='String')]
    [Parameter(Position=2, ParameterSetName='CipherInfo')]
    [Security.Cryptography.PaddingMode]$Padding = 'PKCS7'
  )
  Process
  {
    Try
    {
      if ($PSCmdlet.ParameterSetName -eq 'CipherInfo')
      {
        $Salt = $CipherInfo.Salt
        $InitializationVector = $CipherInfo.IV
        $String = $CipherInfo.CipherText
      }
      $iv = $InitializationVector

      $valueBytes = [Convert]::FromBase64String($String)
      $keyBytes = NewPasswordKey -Password $Password -Salt $Salt
```

```
    $cipher = [Security.Cryptography.SymmetricAlgorithm]::Create('AesManaged')
    $cipher.Mode = [Security.Cryptography.CipherMode]::CBC
    $cipher.Padding = $Padding

    $decryptor = $cipher.CreateDecryptor($keyBytes, $iv)
    $stream = New-Object -TypeName IO.MemoryStream `
      -ArgumentList @(, $valueBytes)
    $reader = New-Object -TypeName Security.Cryptography.CryptoStream `
      -ArgumentList @(
        $stream,
        $decryptor,
        [Security.Cryptography.CryptoStreamMode]::Read
      )

    $decrypted = New-Object -TypeName Byte[] -ArgumentList $valueBytes.Length
    $decryptedByteCount = $reader.Read($decrypted, 0, $decrypted.Length)
    $decryptedValue = [Text.Encoding]::UTF8.GetString(
      $decrypted,
      0,
      $decryptedByteCount
```

```
    )
    $cipher.Clear()
    $stream.SetLength(0)
    $stream.Close()
    $reader.Clear()
    $reader.Close()
    return $decryptedValue
  }
  Catch
  {
    Write-Error $_
  }
 }
}

Export-ModuleMember -Function Protect-AesString, Unprotect-AesString
```

Kilka przykładów.

Przykład 1:

Poniższy kod PowerShell demonstruje proces szyfrowania i deszyfrowania tekstu przy użyciu funkcji Protect-AesString i Unprotect-AesString. Wczytywanie hasła:

```
$pass = Read-Host -AsSecureString
```

Użytkownik jest proszony o wprowadzenie hasła, które jest następnie wczytywane jako obiekt SecureString przy użyciu Read-Host -AsSecureString.

Szyfrowanie tekstu:

$info = Protect-AesString -String "Keep this super safe!!" -Password $pass

Funkcja Protect-AesString jest używana do zaszyfrowania tekstu "Keep this super safe!!" za pomocą wczytanego hasła. Zaszyfrowane dane są przechowywane w obiekcie $info, który jest instancją klasy CipherInfo.

Decrypt

Deszyfrowanie tekstu:

Unprotect-AesString -CipherInfo $info -Password $pass

Funkcja Unprotect-AesString jest używana do deszyfrowania danych. Jako parametry przekazywane są:

CipherInfo zawierający zaszyfrowany tekst, wektor inicjalizacyjny (IV) i $salt,

Wcześniej wczytane hasło ($pass).

Funkcja ta deszyfruje zaszyfrowany tekst, korzystając z podanych informacji i hasła. Ostatecznie deszyfrowany tekst jest wyświetlany na ekranie.

W ten sposób kod demonstruje podstawowe zastosowanie funkcji szyfrowania i deszyfrowania za pomocą algorytmu AES w trybie CBC, a także korzystanie z bezpiecznego wczytywania hasła i przechowywania go w formie SecureString.

Ten przykład szyfruje ciąg przy użyciu domyślnej treści i określonego hasła.

Odszyfrowuje zaszyfrowany ciąg przy użyciu domyślnej treści i tego samego hasła.

Przykład 2:

Powyższy kod PowerShell ilustruje proces zapisu i odczytu informacji o zaszyfrowanych danych do/z pliku za pomocą formatu JSON.

Wczytywanie hasła:

$password = Read-Host -AsSecureString

Użytkownik jest proszony o wprowadzenie hasła, które jest następnie wczytywane jako obiekt SecureString przy użyciu Read-Host -AsSecureString.

Szyfrowanie tekstu i zapisywanie informacji do pliku JSON:

```
$secret = 'Super secret info...'

$cipherInfo = Protect-AesString -String $secret -Password $password -Salt 'MoreSalt'

$cipherInfo | ConvertTo-Json -Compress > ./nothingtosee.json
```

Tekst "Super secret info..." jest szyfrowany za pomocą funkcji Protect-AesString, a następnie informacje o zaszyfrowanych danych (obiekt $cipherInfo) są przekształcane do formatu JSON za pomocą ConvertTo-Json z opcją -Compress. Ostatecznie dane są zapisywane do pliku nothingtosee.json.

Odczytywanie informacji z pliku JSON i deszyfrowywanie tekstu:

```
$info = Get-Content ./nothingtosee.json | ConvertFrom-Json

Unprotect-AesString -String $info.CipherText -Salt $info.Salt -InitializationVector $info.IV -Password $password
```

Informacje o zaszyfrowanych danych są odczytywane z pliku JSON (./nothingtosee.json) za pomocą Get-Content i ConvertFrom-Json. Następnie dane są przekazywane do funkcji Unprotect-AesString w celu ich deszyfrowania. Parametry deszyfrowania, takie jak zaszyfrowany tekst ($info.CipherText), $salt ($info.Salt), wektor inicjalizacyjny ($info.IV) oraz hasło ($password), są dostarczane do funkcji.

W efekcie końcowym tekst jest deszyfrowywany i wyświetlany na ekranie, umożliwiając odzyskanie oryginalnej informacji.
Warto zauważyć, że ten proces jest bezpieczny, ponieważ hasło jest wczytywane jako obiekt SecureString, co zwiększa bezpieczeństwo przetwarzania haseł.

W tym przykładzie zastosowano niestandardową treść do szyfrowania.

Przechowuje dane wyjściowe w pliku json w celu bezpiecznego przechowywania i ponownego wykorzystania później.

Następnie odszyfrowuje przechowywany ciąg znaków, używając przechowywanych informacji i hasła użytego do jego zaszyfrowania.