SkyPenguinLabs

SPL Computer Science Lessons - Steganography for Beginners

////////////////// SPL CSIxC

Powered by SkyPenguinLabs

SPL-CSIxC Introduction

> SPL Computer Science Lessons - Steganography for Beginners

Ever wondered how images can be malicious? Or ever wondered how DRM is implemented in secrecy? In this lesson, we are going to be conversing about some of the surface level basics of steganography to get you warmed up to the chase!

Additionally, this lesson does have a focus topic which is primarily reverse engineering. This means you may notice a few snippets bringing in scenarios related to reversing and more. Feel free to skip those sections if you are only here for the steganography part, though, it does not hurt to read.

Check the course outline on the next page to get an idea of how it is outlined.

Powered by SkyPenguinLabs

Table of Contents

> Course Outline and Course Content

As another free lesson, the content remains minimal, but expands as best as it can for the scope it aims to satisfy.

Section 0x00

> Prerequisites

For this course, you need at the bare minimum a basic understanding of what file formats are. This does not include their structures, programmatic or logical/theoretical technical implementations, or their standards, but instead, just the surface level understanding of a file, the different ones that exist, and how computers can differ between different file types.

Outside of that, if you want to follow along, feel free to set up a VM of Parrot to follow along.

(parrot is what system being used here, but any linux system really does, since the tools we are using are very basic and do not require anything fancy)

Powered by SkyPenguinLabs

Section 0x01

> Steganography From a Bird's Eye View

To kick this off directly, all bs aside, steganography is the act of taking information and attempting to conceal it inside of media formats, such as images, audio & video files, and various other forms of content for various purposes.

Now, there are so many questions that stem from that in of itself that we have to break this up into multiple sections, so I want to give you a good heads up of what the following sections discuss in order:

- **What steganography is technically speaking -** The shallow end introduction to what exactly steganography is and how it works behind the scenes.

- **Why steganography exists** - which breaks down why steganography was made originally, and why it is still used today / how it can be used. This also breaks down when

With them having been broken down, enjoy :)

Steganography Behind The Scenes

Steganography is widely discussed on the surface level and is often used as one of the most clickbaity things out there - because

"oohhhh spooky scary malware devs infested PNGs with malicious payloads and hacked a central server??????"
~ Every content creator

However, hardly anybody on that same surface cares to explain how it works on the backburner.

In order to understand how steganography works behind the scenes, we should at least make an attempt to understand its terminology directly from the dictionary.

By definition

> Steganography is the practice of concealing a message or information within another message or physical object in a way that its presence is not apparent to an observer

If the case is that the message is not concealed within physical observability, then we can imagine that the message and information is buried deep within the data that goes behind the image.

Anybody that has the basic introduction required, understands that images are filled with a bunch of data. The layers, or different parts of the image map to RGB values that get essentially packed into the files body.

Since every file is different, we can not cover '*how steganography works behind xyz media format*' as that is best suited for a book or an entire course haha. However, for a core format, I chose BMP because of how simple it is and it is a good OG for stego.

Behind the scenes, Stegonography takes advantage of the data inside of the image, specifically, unused data in the image by injecting those spaces with the *encoded/hidden* information. When we say, 'unused' data, we need to first get what we mean when we say 'data'. Directly, the data in the image is going to be anything from the file's metadata, to its actual content.

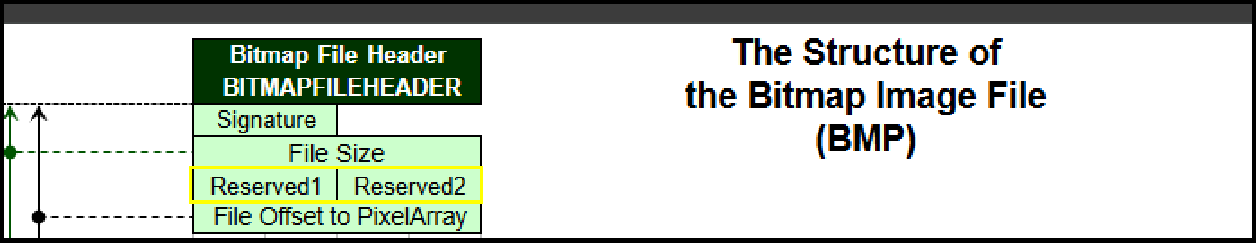
The reason data needs to be unused, or needs to be considered very insignificant to the image, is because if we change anything drastically, anything from humans to even AI is going to be able to tell its been tampered with as the result easily shows in the visual result. And the goal of steganography is to ***hide*** information within an image so we need to make sure nothing hiding in the image affects the way the image is rendered.

BMP is a much simpler format to understand because its data can be easily broken down into the 3 part structure which is absolutely necessary for the format *(depending on the version of BMP, this changes but this is the older structure, the basic one, which contained the DIB header)*:

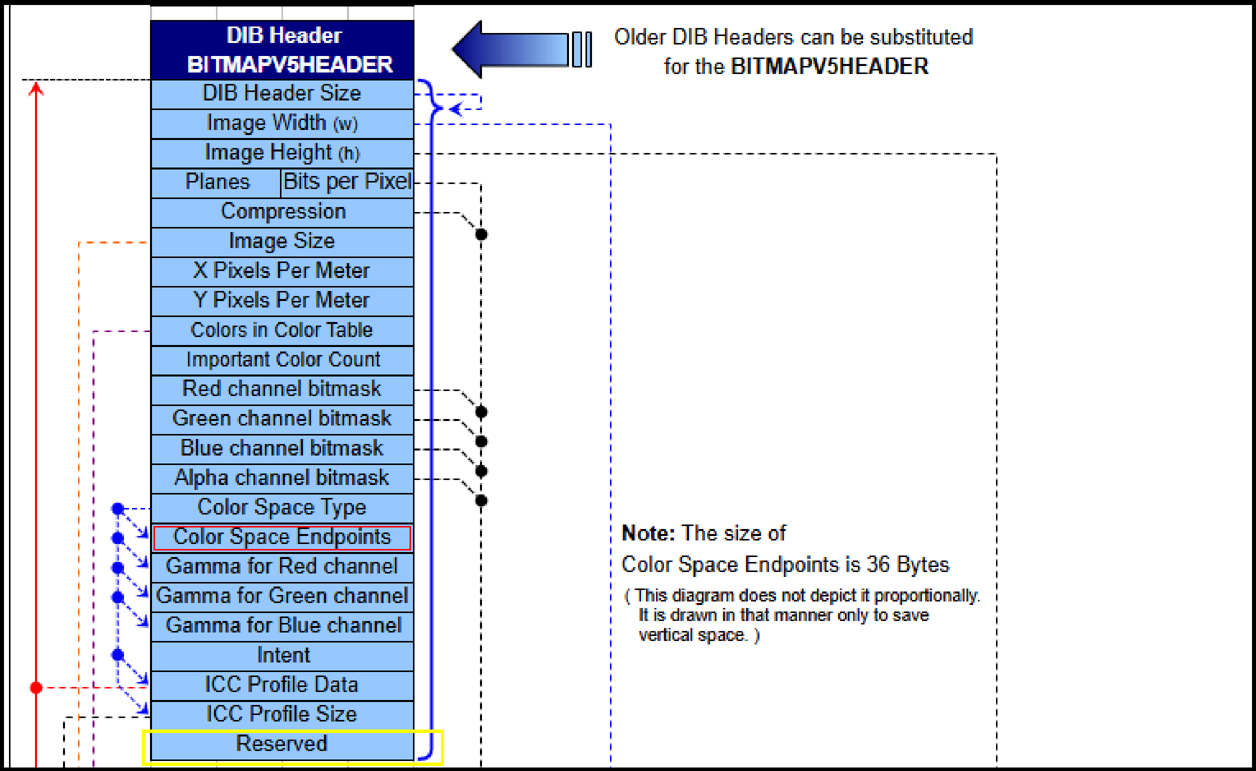
- BITMAP_FILE_HEADER - This is the file header of the bitmap file, containing the magic bytes [0x42, 0x4D] ('BM'), which is how programs identify the start of a bitmap file.
- BITMAP_INFO_HEADER - This is a secondary portion of the heder which includes the specific metadata of the BMP file. This is typically 40 bytes long and contains the height and color information about the file.
- BITMAP_PIXEL_ARRAY - This is the final structure which composes of the raw image data stored as BRG (*Blue Green Red*)

These components get broken down by this graph.

- BMP File Header



- BMP Informational Header



Finally, the actual image data…

- BMP Image Data Section

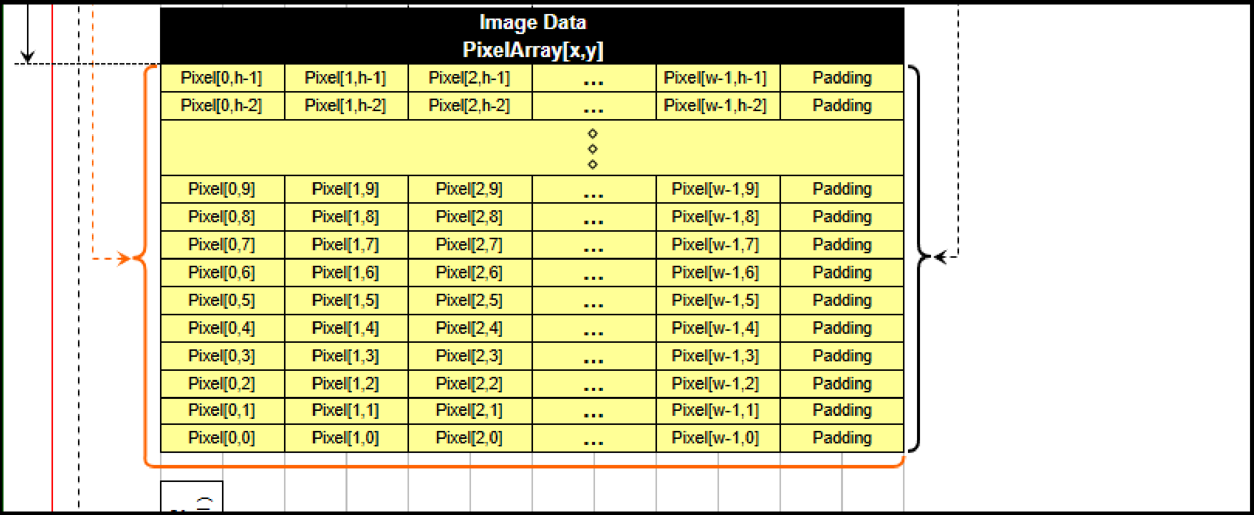


Image Data PixelArray[x,y]					
Pixel[0,h-1]	Pixel[1,h-1]	Pixel[2,h-1]	…	Pixel[w-1,h-1]	Padding
Pixel[0,h-2]	Pixel[1,h-2]	Pixel[2,h-2]	…	Pixel[w-1,h-2]	Padding
			⋮		
Pixel[0,9]	Pixel[1,9]	Pixel[2,9]	…	Pixel[w-1,9]	Padding
Pixel[0,8]	Pixel[1,8]	Pixel[2,8]	…	Pixel[w-1,8]	Padding
Pixel[0,7]	Pixel[1,7]	Pixel[2,7]	…	Pixel[w-1,7]	Padding
Pixel[0,6]	Pixel[1,6]	Pixel[2,6]	…	Pixel[w-1,6]	Padding
Pixel[0,5]	Pixel[1,5]	Pixel[2,5]	…	Pixel[w-1,5]	Padding
Pixel[0,4]	Pixel[1,4]	Pixel[2,4]	…	Pixel[w-1,4]	Padding
Pixel[0,3]	Pixel[1,3]	Pixel[2,3]	…	Pixel[w-1,3]	Padding
Pixel[0,2]	Pixel[1,2]	Pixel[2,2]	…	Pixel[w-1,2]	Padding
Pixel[0,1]	Pixel[1,1]	Pixel[2,1]	…	Pixel[w-1,1]	Padding
Pixel[0,0]	Pixel[1,0]	Pixel[2,0]	…	Pixel[w-1,0]	Padding

Based on the basic comprehension of how Stegonography is executed - that is by injecting and stuffing data into points that are unused by the file that results in absolutely no visual or programmatically detectable change in the files data which results in an error, we can use some of these pads and reserved portions of the file to stuff our data.

Usually, when data is stuffed in the image, or gets hidden, people who employ steganographic techniques will encode it. In other cases, they encrypt it and hide the key in the data, which a specific type of software known as a decryption tool knows how to find.

Woah woah woah, did I just say, they can encrypt data and hide it in other segments?

Well yeah, that's true as well. A much more advanced scenario of steganography being used involves malicious attackers, such as malware developers, engineering programs which are designed to look at and parse very specific segments of the sections in the BMP file's structure.

The program then takes the data it gets from scanning the segments, and knows how to assemble it in such a way that creates an encryption key, which can be used to decrypt portions of the data that were scanned and classified as ‘encrypted’ to the program.

This is very hypothetical, and is very possible but its actual technical implementation varies drastically depending on many different factors involving the type of information trying to be hidden, how and what it's trying to be hidden inside of, and even miniscule details such as the images properties.

We will discuss this further in the practical section, but I wanted to at least make sure this concept was hammered down.

Now, while we did cover chunks of the image that are reserved or unused, such as padding, we did not cover any other portions of the file which can be used and manipulated, and there is one common favorite amongst the public.

This method of steganography is known as LSB Steganography. This requires an understanding how colors translate to binary which is not that complicated. Some people get stuck but let's simplify it.

Basically.

Each pixel in a 24-bit BMP uses three bytes, one each for Blue, Green, and Red (BGR, not RGB) intensity values from 0-255.

If we take this into consideration, this means for every single pixel in the image, we can take the least significant bit (*LSB, which is considered to be the rightmost bit in a binary number*) and modify it without causing any visual distortion to the image. If every set of BGR values has 3 LSB placements (*1 for every color code, or 1 for every set of decimal values converted to binary*), we can take advantage of this and modify the binary bits accordingly.

Let's put this into a scenario where ‘actual’ data that we can interpret. I chose the letter ‘H’ for this, the beginning of ‘Hello world’.

If this character wanted to be embedded, we would have to take its ASCII decimal value, convert it to binary, whose ASCII decimal is 72 which is 8 bits of binary as `1001000`.

We are going to be stuffing the first 3 initial bits in a set of 3 BGR values as BGR(255, 0, 68) meaning we have to convert every single decimal value [B, G, R] to their binary representation.

- 68 - 01000100
- 0   - 00000000
- 255 - 11111111

Then take our value's (H) binary representation ([1][0][0]1000) and stuff it into the BGR values by changing the last bit to follow in this sequence. So since our value is 010 we get

- 68 - 01000100 -> '1' = 0100010[1] (Rightmost 0 gets flipped to a 1)
- 0   - 00000000 -> '0' = 00000000 (No bit flip here)
- 255 - 11111111 -> '0' = 1111111[0] (flip the last bit to a 1)

Now our BGR values, while visually unchanged, were changed to represent the first 3 bytes of the letter 'H'. In order to implement the full character, we need to do this 2 more times to satisfy stuffing the word 'h' in there. Which, you can probably guess, requires automation.

As you can imagine, while this concept seems simple on the surface, an individual automating this would require a decent amount of knowledge in software engineering to do properly. This is because, if we do this on a scale where said individual is hiding entire sets of payloads using this method, they need to tell the program how to properly track and calculate every single location which usually involves storing some metadata which a decoding program is specifically designed to look for prior to attempting the decode of the content inside of the image.

Concluding

Steganography is widely discussed on the surface level, yet not really explained much in depth and can often be hard for newcomers to understand. However, while people also make it out to be easy, or overly complex, it sits somewhere in-between, not being the most unique thing in the world, but being a tool that can be utilized by attackers in unique and niche scenarios which allows them to gain a massive foothold on their target. Additionally, attackers and really anybody can take the same information stored inside of an image and encrypt it, making it extremely hard for anybody to understand without a large amount of anomalies in the encryption, or the program that actually decodes it. The latter is more likely.

In those scenarios, we may need to apply much more offensive and proactive techniques to understand how the data in the image was stored. Oftentimes, one of the most called for skills is reverse engineering. However, before a reverse engineer can even start with the application at hand, if they even obtain it, they need to understand what file format is being used here.

The section on the next page, discusses file formats & their implementations in steganography.

Section 0x02

> File Formats and Stegonography

As we mentioned in the first section of this lesson, initially, we had to stick with one file format to demonstrate one of the concepts known as LSB steganography. The target filetype ended up being chosen as a BMP file purposefully for simplicity.

Now, you may already be wondering - *why exactly does the file format matter more than just the use with it?*

Truth being told, steganography gets applied in many different scenarios, from actual exploitation to simply just hiding messages for sake of storage. However, when steganography is applied in a specific scenario, the type of scenario often determines the type of file that the person in question has to use. Additionally, it also depends on the type of steganography that is going to be used.

For example, consider an attacker which wants to stuff some malware in an image, so that when a user opens it through a specific image rendering software, the software scans over that code segment in the image and executes it (*most AV catch this without a shitload of modifications to the data*).

The attacker needs to know multiple things

1. **What File Format the Software Accepts**
   In this case, the attacker needs to have a firm understanding of what file format the software takes depending on the scenario. If we are applying the one above, since this is specifically a program for rendering media files, we can presume it renders most common image formats. Spanning from JPG, to PNG and BMP.

   This allows the attacker to understand how they are going to approach the malware portion of this task.

2. **What File Format Triggers The Flaw**

   Some cases may require the attacker to understand exactly what file format in the chosen software results in the flaw or state which can be taken advantage of. For example, sanitization exists on specific types of image formats such as raster file formats, but not vector files such as SVG.

   Knowing what format triggers the flaw is important because you do not want to be wasting your time stuffing payloads into images if there is nothing that's going to work.

   Additionally, attackers may find a format which triggers a flaw and is an entrypoint, however, this entrypoint may be temporary as other, later stages of the exploitation phase of the flaw may introduce new roadblocks such as configured firewalls or final parsing rules and systems prior to action completion.

   For example, social media apps sometimes have multiple phases of both sanitization, image compression, reconstruction, and etc which involve making absolutely PERFECT copies of image content to fit their platforms frontend AND backend standards while also involving process in the middle that validate the contents prior to passing to other components. This is quite a pipeline, and exists in complex scenarios where people have to go ALL out mapping every single scenario to media formats, and outside of steganography alone, there are so many more areas where attackers can take advantage of the way file formats make a program react that changes the way an exploitation scenario comes out.

   The next explanation explores how the file structure is enough to change the outcome.

3. **Steganography Method Changes on File Change**
   As we were exploring above, we used LSB steganography with the BMP file. However, if you tried to implement the same technique on a JPEG file, the result would change.

   This is because different image formats carry entirely different structures and internal metadata from other image formats, even if they build off of existing ones and JPEG specifically carries a compression algorithm known as Lossy. JPEG's lossy compression algorithm discards "unimportant" visual information which is exactly where hidden payloads and data gets stuffed unlike BMP files that maintain their original pixel relationships and statistical properties making embedded data harder to distinguish Image variations.

   Additionally, specific versions of images carry other features which change the structure of the image and the way structures get interpreted. This also changes the methods of steganography the person employs on the format but also ***how*** it gets employed. .

The modern world of steganography hardly falls in the middle of the fence as its often scenarios are filled with the most basic scenarios where steganography is leveraged by an attacker or person and on the other side it falls extremely difficult, requiring a lot more engineering & understanding of the components at hand.

Now of course, we gotta give the people defending a good chance to. As we all know, attackers leave footprints. Sometimes, when steganography is used for malicious purposes, such as malware, developers may leave portions of the decoding components in their droppers (*a type of software used to drop or deliver and execute a malicious payload, which in some cases may be stuffed inside of an image as we explored*) which can be used by reverse engineers, and defensive researchers to understand what an attackers goal is when implementing steganography.

Section 0x03

> Practical Examples of Steganography

Now that we understand what steganography is at the surface level, and understand a basic example of how it works, I would like to give you a good practical example before we move onto the conclusion and finish this short lesson off.

In this scenario, we are going to still stay within the use of BMP files. But instead of working with LSB steganography, it will use a series of basic CLI utilities to embed a base64 encoded message which says 'SPL' inside of a basic BMP file's header reserved portions.

Now, as it may seem, the data is small. However, this is a requirement because the reserved header space only allows for about 4 bytes of room.

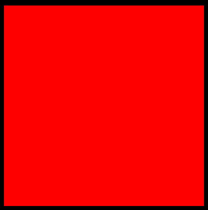
- **Note**: For more *'intermediate'* people reading this, you may also understand that these 4 bytes do offer significant room for bit switches, or storage of offsets in regards to other information encoded inside of the file. In some cases, the offset references the location of say a table of payloads to reference or encoded keys and signatures. Whatever the use may be, headers are usually used for storing lightweight but symbolistic information for applications relying on the asset.

The following commands below walk you through a basic scenario of encoding the phrase 'SPL' into base64 and then placing it inside the header of a very limited BMP image.

1. **> First, we need to create the BMP image, so, run the command below.**

```
convert -size 100x100 xc:red output.bmp
```

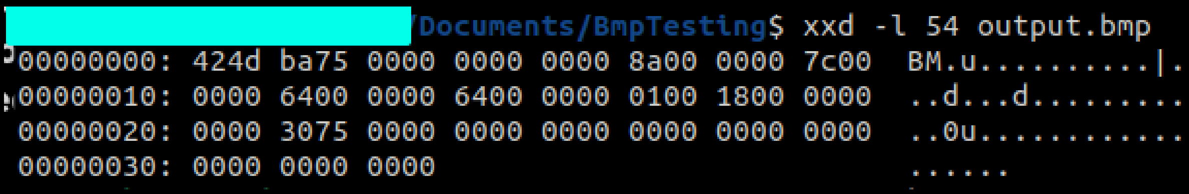
This results in the image shown on the right



2. **> Now we need to make sure the BMP image is suitable, this includes verifying there is room in the header. Since a header is 54 bytes long, we need to read 54 bytes with xxd**

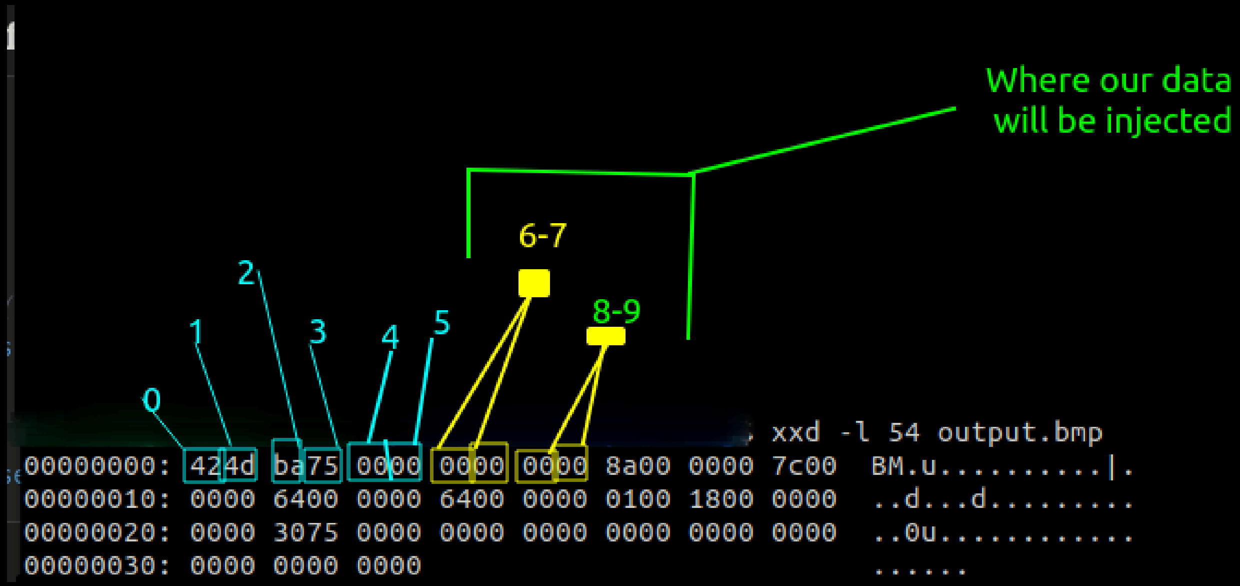
   ```
   xxd -l 54 output.bmp
   ```

   This results in the following.



```
[redacted]/Documents/BmpTesting$ xxd -l 54 output.bmp
00000000: 424d ba75 0000 0000 0000 8a00 0000 7c00  BM.u..........|.
00000010: 0000 6400 0000 6400 0000 0100 1800 0000  ..d...d.........
00000020: 0000 3075 0000 0000 0000 0000 0000 0000  ..0u............
00000030: 0000 0000 0000                           ......
```

Since a BMP header's reserved structure is defined to contain two different sections, where reserved section 1 starts at offset 0x06, and reserve section 2 starts at offset 0x08, we can see that the reserved portions are null. Shown below.



```
xxd -l 54 output.bmp
00000000: 424d ba75 0000 0000 0000 8a00 0000 7c00  BM.u..........|.
00000010: 0000 6400 0000 6400 0000 0100 1800 0000  ..d...d.........
00000020: 0000 3075 0000 0000 0000 0000 0000 0000  ..0u............
00000030: 0000 0000 0000                           ......
```

As you can see, by the highlighted sections on the image in the next page, we are going to need to read 6 bytes into the file (*this is known as 'seeking' into a file conceptually, the more technical terminology*) and count 2 spaces in to write our content.

3. **> Create the base58 encoded string and write it to the file's segments**

   Now we need to create the content and place it into those reserved segments. So we can do this simply by running the following command with base64.

   ```
   echo -n 'SPL' | base64
   ```

   And this gives us the string U1BM which needs to get converted to hexadecimal before we can write it into the file.

   The hexadecimal value of the 4 bytes are the following (*ASCII characters are 1 byte [8-bits] in size, so our result is 4 hexadecimal values*)

   ```
   55 31 42 4D
   ```

   Now we need to format it a little bit for the program to understand what we are writing.

   ```
   RESERVED_SECTION1 = '\x55\x31'  - | Str part 1
   RESERVED_SECTION2 = '\x42\x4d'  - \     part 2
   ```

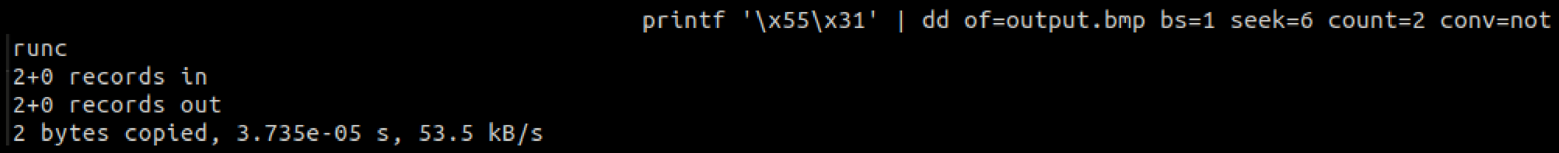
   Finally we can pipe the result of the characters when displayed toa program on linux known as 'dd' (*disk duplicator, aka disk destroyer because of how dangerous it can be to users who do not know what they are doing*).

   Since we have two segments that we need to write to, both at different segments, we call the tool twice.

   This tool when called with the following command will open our BMP file, seek 6 bytes into it, count twice and write the output from the pipe to the offsets in the image.

```
printf '\x55\x31' | dd of=output.bmp bs=1 seek=6 count=2 conv=notrunc
```

Finally we can pipe the result of the characters when displayed toa program on linux known as ‘dd’ (*disk duplicator, aka disk destroyer because of how dangerous it can be to users who do not know what they are doing*).



```
                                                printf '\x55\x31' | dd of=output.bmp bs=1 seek=6 count=2 conv=not
runc
2+0 records in
2+0 records out
2 bytes copied, 3.735e-05 s, 53.5 kB/s
```

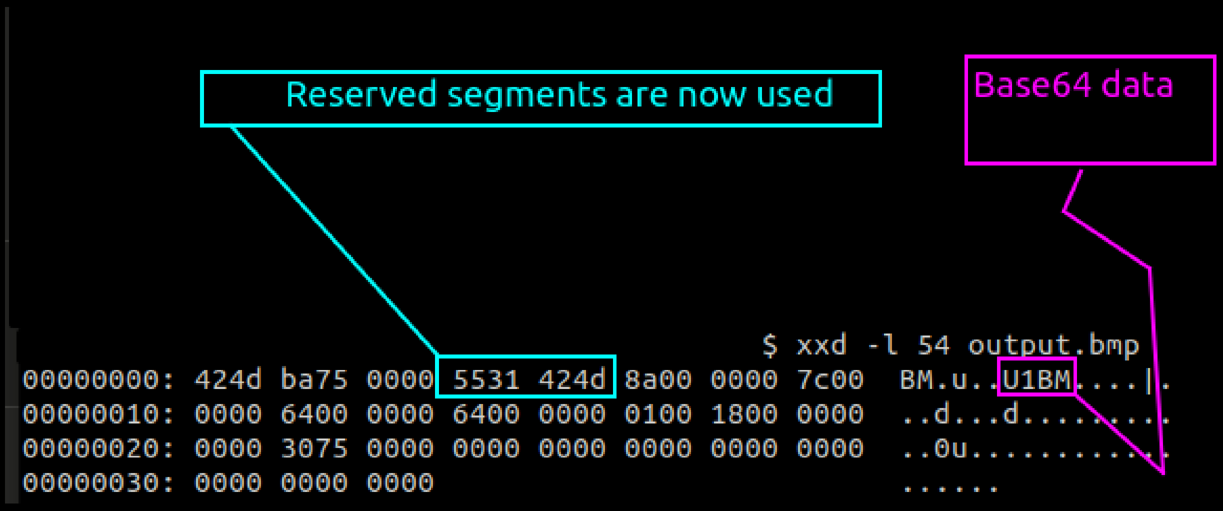
We do this again for positions 8-9

```
printf '\x42\x4d' | dd of=output.bmp bs=1 seek=8 count=2 conv=notrunc
```

Which also results in a similar output.

4. **> Finally, let's validate that our data placement was successful and tracked.**

   To validate, all we need to do is rerun the xxd command we did above to view the files header, and make sure the string we wanted to write exists in the reserved header spots.



```
                                                    $ xxd -l 54 output.bmp
00000000: 424d ba75 0000 5531 424d 8a00 0000 7c00  BM.u..U1BM....|.
00000010: 0000 6400 0000 6400 0000 0100 1800 0000  ..d...d.........
00000020: 0000 3075 0000 0000 0000 0000 0000 0000  ..0u............
00000030: 0000 0000 0000                           ......
```

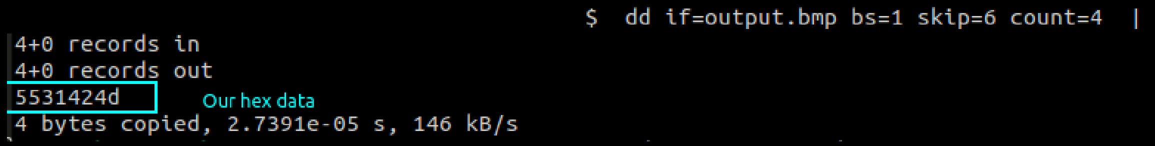
And as you can see, we got a successful hit with this! Now, all we need to do is tell a program to read the file and about any program which reads the files reserved segments, and parses it will parse the text contents.

To recover the content with dd we can run the following command. The command reruns dd to go back to the exact positions we had before, but instead counting 4 bytes into the 6th skipped position once the skip finishes (*we could have also done this during writing the payload to the file to save time*) and we send a pipe '|' to 'xxd' which takes the output of dd and tells xxd to take that binary data from dd and output it as a continuous stream of lowercase hex digits, with no addresses, no ascii translation, and no formatting.

We do this because we want the raw data, so, that being said, the command is:

```
dd if=output.bmp bs=1 skip=6 count=4 | xxd -p
```

Which gives us



```
$  dd if=output.bmp bs=1 skip=6 count=4  |
4+0 records in
4+0 records out
5531424d
4 bytes copied, 2.7391e-05 s, 146 kB/s
```

And this allows us to take this and convert it to ascii using xxd as well. So we can pipe the output of xxd into the output of xxd again with the following command.

```
echo -n 5531424d | xxd -r -p
```

Which gives us `U1BM` and with that, we can confirm that the placement of data was not only correct but was also not tampered with or corrupted during the process. This is extremely important to learn how to do the more you get into this, so make sure you never leave that out of the skillshed.

While this was one of the most basic applications of steganography, there are so many other avenues and things to consider when working with images that makes for so many opportunities for leverage when using them in realistic scenarios. Whether it is to hide information, or the goal is to use steganography in exploitation scenarios.

However, now that we understand what it looks like, we know exactly how to look for it logically. So let's bring this all together and bridge the other side of the coin here.

How the Learning Steganography aids Reverse Engineers

Before we end this entire lesson I would like to dedicate a small section to the reverse engineering side of things. More so, because you may find yourself reverse engineering the way software, and even malware, utilize image media and assets to store specific information which other programs rely on.

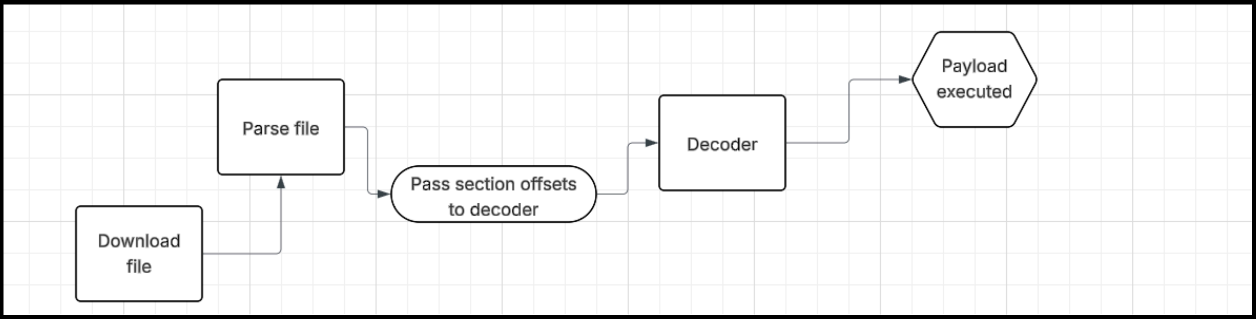
In one of the scenarios above, an attacker was stuffing some malicious code inside of their images, and a dropper, would fetch the malware via image from a remote C2, and using specific techniques programmatically, would locate and decode the payload in the image so it can be executed on the host.

At some point, a reverse engineer may stumble across the dropper, and try to rip it apart to understand exactly what is happening. In order to successfully do this, the reverse engineer will apply sets of knowledge such as the ones used in this lesson to try and figure out how the dropper may be attempting to parse through the image’s data.

Like the way we validated the image contents and successful injection, a reverse engineer would attempt to catch the resource in transit before the dropper deletes it and dump its data to view what's inside. (*typically files may be deleted or shredded after being used to avoid footprints, but deleting files depending on the method in of itself can be noisy as well*)

If there is anything obvious, such as a large amount of text in the center body of the files metadata sections base32 encoded, then that is usually a red flag and something that is paid attention to. Additionally, the information of it being a base32 string in of itself gives the reverse engineer just enough incentive to start searching the program for modules, software or even custom implementations of base32 encoders/decoders. When the reverse engineer finds the reference to the decoder in the program, they can cross reference this up the call-chain, going from the decoder, to the component which called the decoder originally parsing the images content finally to the code which called the image parser fetching and prepping the image for parse before being decoded.

Just from that, the reverse engineer may be able to reveal the entire chain of logistics within the program, being able to break down the process like below.



While this is hypothetical, and depending on the programs complexity it may not always be this simple of a chain nor process, the simple patterns in the image, such as a base32 encoded string, if a reverse engineer is trained enough to know about how steganography works, may be enough to figure things out.

If there is NOT any obvious information, than the idea would be to find specific information relating to the downloading and usage of resources, as well as watching how the software behaves in an active (*dynamic*) environment by intercepting the data it downloads, looking at the way it writes to files by inspecting environment calls, and so much more.

Knowing steganography as a reverse engineer is extremely important and can be also really cool to deep dive into when involving most of those scenarios.

Now, let's finish this lesson off, and send ya back 127.0.01

Section 0x04

> Lesson Conclusion

Steganography is a LARGE world, it has so many different methods of being implemented into digital technologies and so many possibilities to be abused. One of the most delicate art forms of payload crafting as well that I personally see being used even with as much validation exists.

Additionally, while it may not be something you want to spend all of your focus on, it is a really important skill to grasp technically even to a level of identification for applications such as reverse engineering and additionally, it never hurts to expand it.

A large amount of steganography has started to become ported to software, and automated technologies that can account for things that humans can't, such as locating algorithms encoded in data, splitting and reassembling fragmented text across hundreds of chunks in a resource encrypted through multiple layers. Additionally, the idea of using steganography maliciously in today's world also includes a large understanding of the technologies being exploited and the specific component loading the images content. This is to make sure you account for bypassing sanitization, header and section validation / integrity if the system includes it and various other systems built to prevent abuse.

I am super excited to talk about this topic more if people are actually interested in it, and make some full fledged technical courses over it because like specific portions of RE, only the popular fancy and click baity parts of steganography are brought to the surface until some random and obscure method becomes popular once from the one time it was used in a niche scenario.

With that being said, I also hope you learned a lot in this lesson and can not wait to see you in more!