Australian Government

Cyber Security Challenge Australia 2014
www.cyberchallenge.com.au

CySCA2014 Random Writeup

Background: Its super random!

Random 1 - Pulp Fiction

Question: RL Forensics Inc. has contracted Fortcerts to recover information stored in this bitmap. Both companies failed to recover the information but they identified that it was encrypted with AES-ECB. Can you recover the information in the encrypted bitmap?

Designed Solution: Players need to visualize the encrypted bitmap. They will then be able to see the flag.

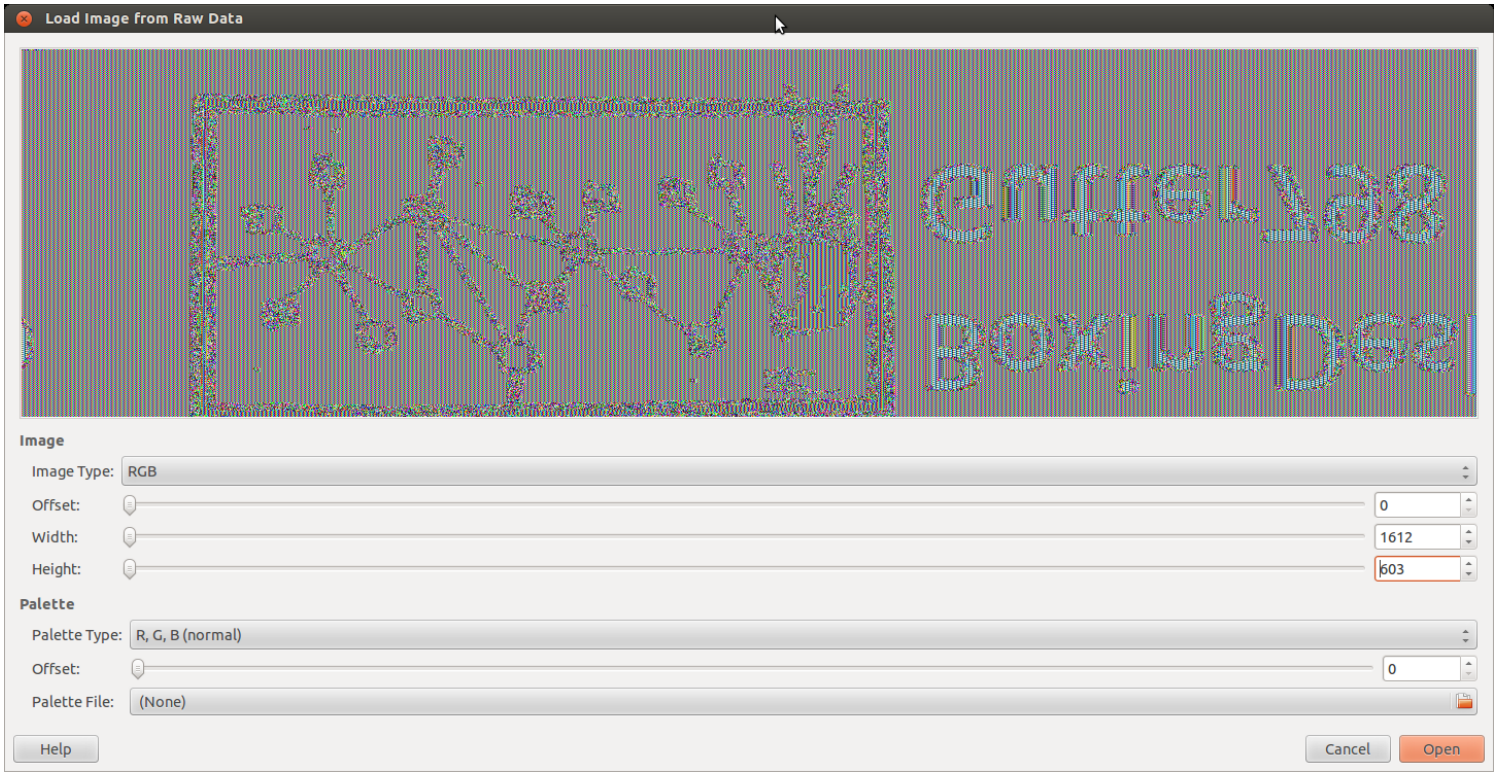
Write Up:
The file that is provided for this challenge is "438b8e5411a303d950d0cbe1bfe2230b-rand01". This is a 1.9mb file.

A quick Google for "bmp encryption" highlights a few articles comparing ECB vs CBC. The Wikipedia article has an example of bitmap encryption for ECB, illustrating the outline of Linux Tux despite the file being encrypted. This is due to blocks being encrypted without chaining causing patterns in the plaintext data to also be apparent within the encrypted data.

Unfortunately, in the file provided, the header has been encrypted and the height / width are unknown.

We can solve this two ways, first we can use a tool to visualize the raw bytes such as GIMP or you can rebuild the header. We will run though both methods below.

Using GIMP, you can load the provided file as a raw image. Then it's a matter of scrolling through the widths with the preview panel until an image appears. Eventually you come to a 1612 pixel wide image. Since a bitmap stores pixels from bottom row to the top row, you will need to flip the image vertically. Unfortunately, in Kali you need to first download the GIMP which can take a while.



Alternatively if we do not have GIMP, we can rebuild the image header and view the bitmap that way.

Given that a bitmap image is usually 24bits per pixel and the header is 54 bytes, the image size is X * Y * 3 + 54 = [filesize]

To start we will get the actual file size in bytes.

```
#> stat -c %s 438b8e5411a303d950d0cbe1bfe2230b-rand01
1977978
```

Now that we have the size we can create determine how many pixels are in the encrypted image.

```
(1977978 - 54) / 3 = 659308
```

We can use this value to determine which width and height value pairs give us a total pixel count matching the calculated pixel count.

```
dimensions = []

for i in range(5000):
	for j in range(5000):
		if (i*j) == 659308:
			dimensions.append((i,j))

> [(403, 1636), (409, 1612), (806, 818), (818, 806), (1612, 409), (1636, 403)]
```

We create a test bitmap to and use hexdump to extract the header bytes.

```
#> hd -n 256 test.bmp
00000000  42 4d 7a 2e 1e 00 00 00  00 00 36 00 00 00 28 00  |BMz.......6...(.|
00000010  00 00 4c 06 00 00 99 01  00 00 01 00 18 00 00 00  |..L.............|
00000020  00 00 44 2e 1e 00 00 00  00 00 00 00 00 00 00 00  |..D.............|
00000030  00 00 00 00 00 00 ff ff  ff ff ff ff ff ff ff ff  |................|
```

We know that in the bitmap header width is stored at offset 0x10 and is 4 bytes long. We also know that height is stored at offset 0x14 and is 4 bytes long. In our test file width and height are 0x0000064c and 0x00000199 respectively.

Finally, we create a script to create a bitmap for each combination of width/heights above using our test bitmap header and the bitmap data from the encrypted file.

```
from struct import pack

# precalculated dimensions
dimensions = [(403, 1636), (409, 1612), (806, 818), (818, 806), (1612, 409),
(1636, 403)]

# open the encrypted file for reading
f = open('438b8e5411a303d950d0cbe1bfe2230b-rand01', 'rb')
# skip over the header
f.seek(54)
# store the bitmap data for writing
data = f.read()
f.close()

# open the test file with the header
f = open('test.bmp', 'rb')
# seek to the width/height bytes
start_hdr = f.read(18)
# skip over width/header
f.seek(26)
# get the rest of the header
```

```
end_hdr = f.read(28)
f.close()

# write out a bitmap for all combinations of width/height
for i in dimensions:
  # create uniquely named file for width/height
  f = open('%s-%s.bmp'%(i[0], i[1]), 'wb')

  hdr = start_hdr
  hdr += pack('i', i[0])  # write the width
  hdr += pack('i', i[1])  # write the height
  hdr += end_hdr

  # write the header
  f.write(hdr)
  # write the bitmap data
  f.write(data)
  f.close()
```

Viewing the 1612-409.bmp image will reveal the xkcd image (350) and flag **BoxingDesktopGutter798**.

Random 2 - Spanish Trout

Question: Fortcerts has been hired by Mad Programming Skillz Pty. Ltd. to perform source code review to find vulnerabilities in the password checking algorithm that they use in many of their products. Fortcerts does not have the expertise to do c code auditing so they have asked you to take a look. Try find any vulnerabilities and capture the flag to demonstrate that an attacker could exploit the identified vulnerabilities. The test code is running at 172.16.1.20:12345

Designed Solution: Players need to identify that strncmp shouldn't be used to compare binary data. Once they have identified this, they need to find a hash collision and submit it to the server to capture the flag.

Write Up:
Looking at the code snippet we can derive the functionality of the test code. It receives the player supplied password string, it uses SHA1 to hash this supplied string and uses strncmp to compare it against the SHA1 hash of the secret key. If they do not match it prints both hashes to the player.

Understanding the strncmp function will help to identify the issue with this code.

"This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ, until a terminating null-character is reached, or until num characters match in both strings, whichever happens first."
http://www.cplusplus.com/reference/cstring/strncmp/

This means two strings may successfully compare even if they are different lengths. For example:

```
s1 = "AB\0CDEFG"
s2 = "AB\0"
strncmp(s1, s2, 8) == 0  # True
```

We connect to the server, and are prompted to enter a password. Once we enter a dummy password, we are given the SHA hash of the password and the SHA hash of the secret key.

```
#> nc 172.16.1.20 12345
Welcome to the SHA password oracle.

Enter your password:
test
Your hash A94A8FE5CCB19BA61C4C0873D391E987982FBBD3 does not match the secret hash
3BFC00848D4F990A3D3C3131A17B6092F89DDAE4
```

The SHA1 hash of the secret key has a \x00 character at the 3rd byte. To make strncmp think that the hashes match we need to find a string that has “\x3B\xFC\x00” as the first three bytes of its SHA1 hash.

This is quite easy and requires little CPU time to achieve.

```
from hashlib import *

for i in range(33,126):
  for j in range(33,126):
    for k in range(33,126):
      for l in range(33,126):
        if sha1("%s%s%s%s"%(chr(i), chr(j), chr(k), chr(l))).digest()[0:3] ==
"\x3b\xFC\x00":
          print "Password found: %s%s%s%s" % (chr(i), chr(j), chr(k), chr(l))
```

We run our python script for a few minutes and get a number of strings whose SHA1 hash starts with “\x3B\xFC\x00”.

```
#> python soln.py
Password found: .m7)
Password found: ES5U
Password found: S{o\
Password found: c9EX
Password found: nffx
```

We then connect to the server and enter the first password found by our script and we receive the flag **RetroFarmingAssault570**

```
#> nc 172.16.1.20 12345
Welcome to the SHA password oracle.

Enter your password:
.m7)
Congratulations: The key is RetroFarmingAssault570
```

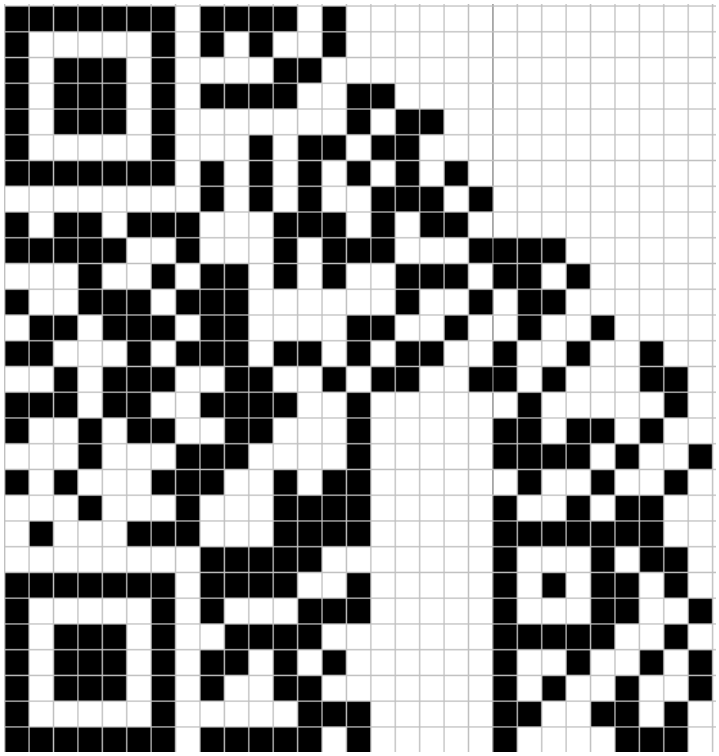
Random 3 - Reed Between The Lines

Question: In an RL Forensics Inc. case contracted out to Fortcerts, an image has been found on a suspects computer. Analysts believe there is corrupted information secretly hidden in the image. As a "computer expert" they want you to recover the data from this image.

Designed Solution: Players need to identify that the message in the challenge was stored in a QR code partially obscured on the right side of the supplied image. Players then needed to identify that QR codes have redundancy and error correction but too much data was missing for this to be automatically corrected. They had to manually decode the data that was present, predict enough data so that the automatic error correction can take place and reinsert it back into the QR code. This will allow automatic error correction to recover the flag.

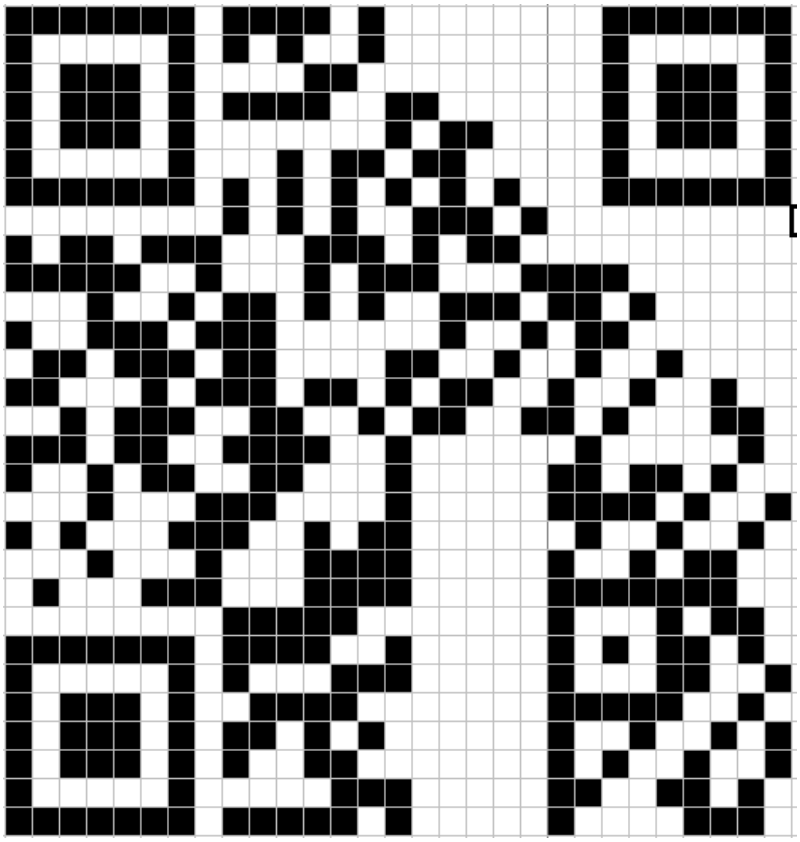
Write Up:
In this challenge we were given an image of taken in Canberra. It showed old and new Parliament House, the High Court and Questacon across Lake Burley Griffin. On the right hand side of this image was a QR code duct taped to a bin. Unfortunately, the QR code was at a slight angle and duct tape was concealing two portions of the QR code. The image with the QR code highlighted is shown below.

The first step to this was to extract the QR code from the image. Photoshop skills help here, but it can easily be done by manually copying the data onto a 29 x 29 grid. For this write up, we used Excel to create a 29 x 29 grid. Below is the QR code in Excel after it was manually extracted from the image.



As you can see there is not enough information for a QR code reader to decode this message and too much data is missing for the error correction to calculate what is missing. So we have to reinsert enough of the missing information so that a QR code reader can use the QR codes error correction to determine the missing information.

To do this we can start by adding the required sections of a QR code that are missing back into our QR code. Firstly and the most important is the positioning cube in the top right corner. If there are not three positioning cubes on the QR code then a QR code reader will not read the QR code.



Next we will add the destroyed timing and metadata bits. Fortunately the timing bits are always the same and the metadata bits are duplicated in two places on the QR code. The diagram on the next page highlights the sections that these are.

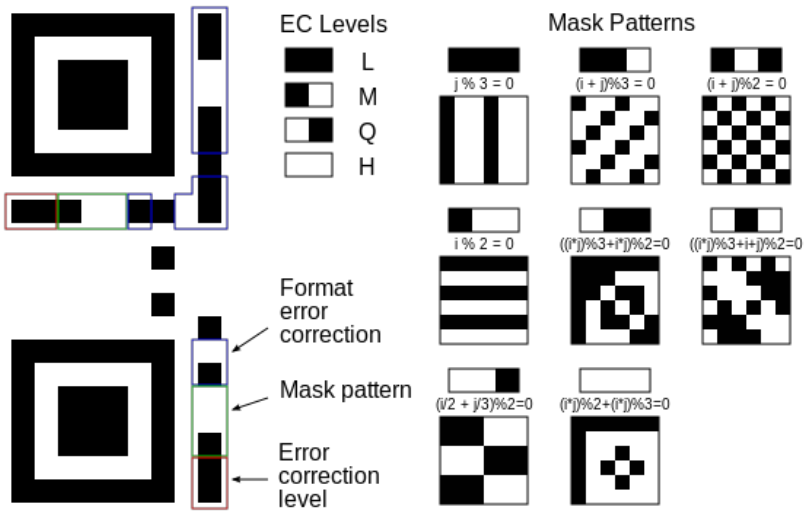
In this diagram each bordered color represents a different section of the metadata.

- Orange is the timing bits. These always alternate on/off between the edges of the positioning cubes.
- Blue are format error correction bits.
- Green is the mask pattern. These tell us what mask was used on the QR code.
- Red is the error correction level. These tell us what level of error correction was used when this QR code was generated.
- Purple is always set on every QR code.
- Yellow is an alignment cube.

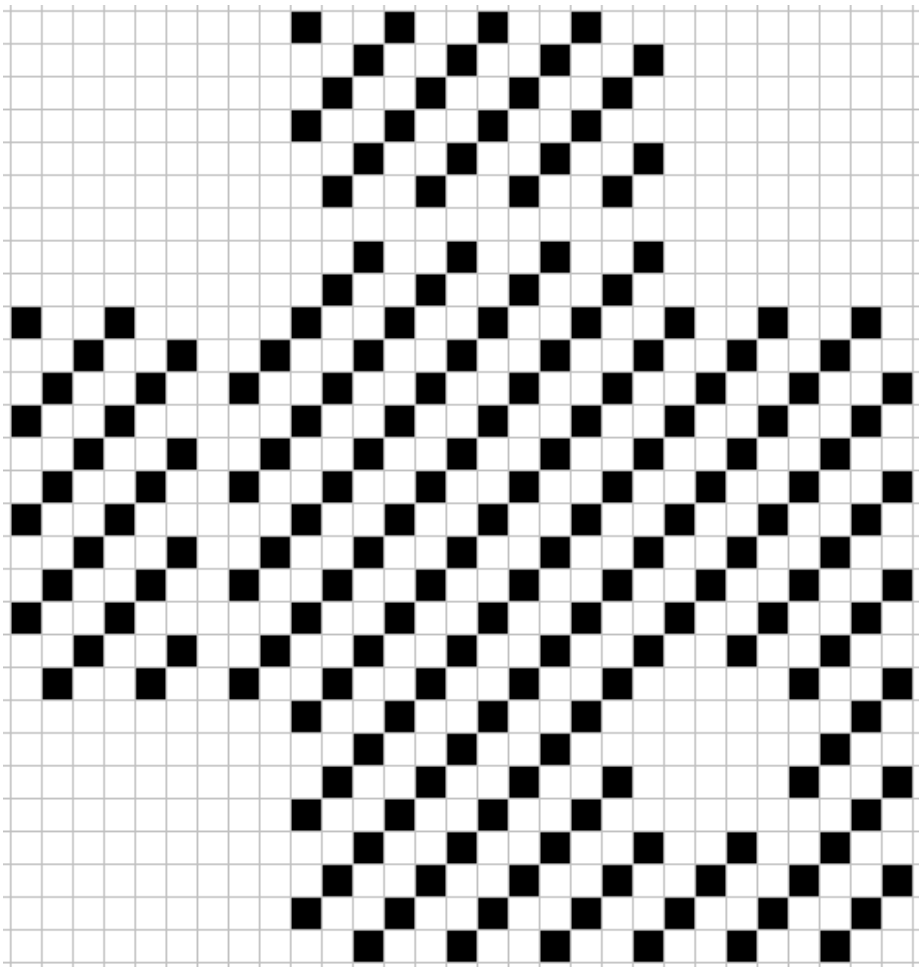
As you can see, the metadata sections are mirrored across the two different locations of each colored section. This is how we determine what bits to set in the sections that were missing.

At this point you could attempt to read the QR code with a QR code reader, however there is not enough error correction in the QR code to make up for the missing parts. So we need to attempt to read it manually.

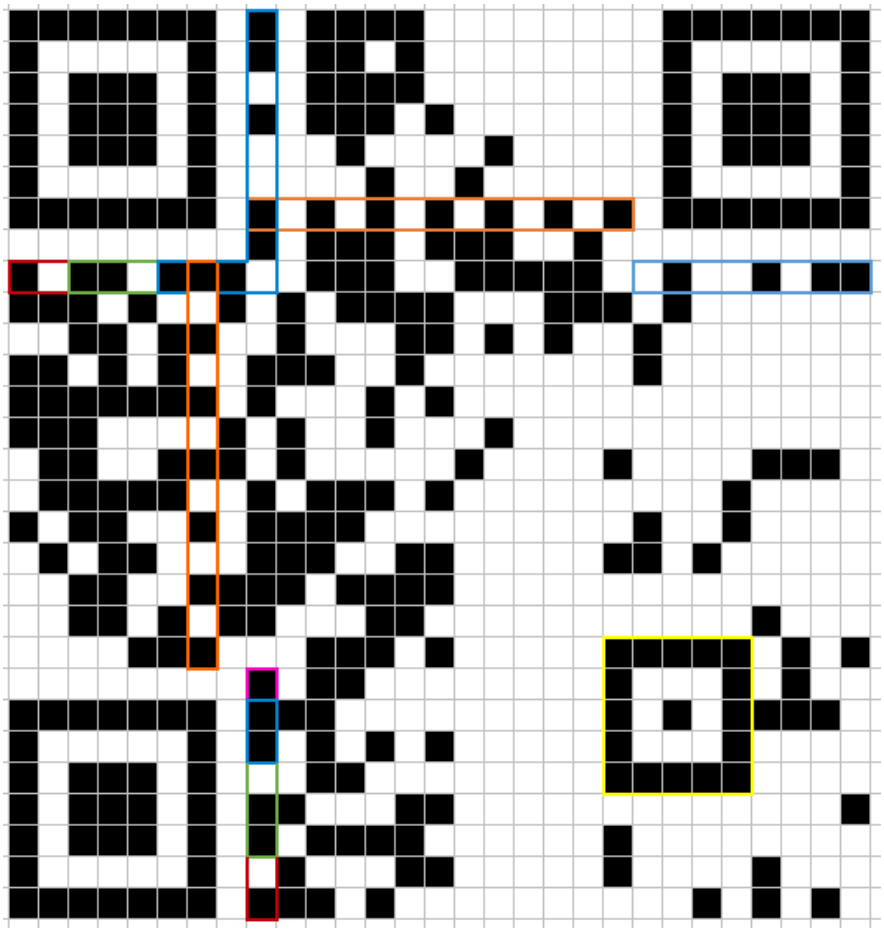
When a QR code is generated it has a mask applied over the data bits, the metadata bits are not masked. So to read the data we need to unmask it. To determine what mask was used we need to look at the mask bits. In this case it is [black][black][white] or 110. Wikipedia provides this handy image on the next page to help decode the mask.



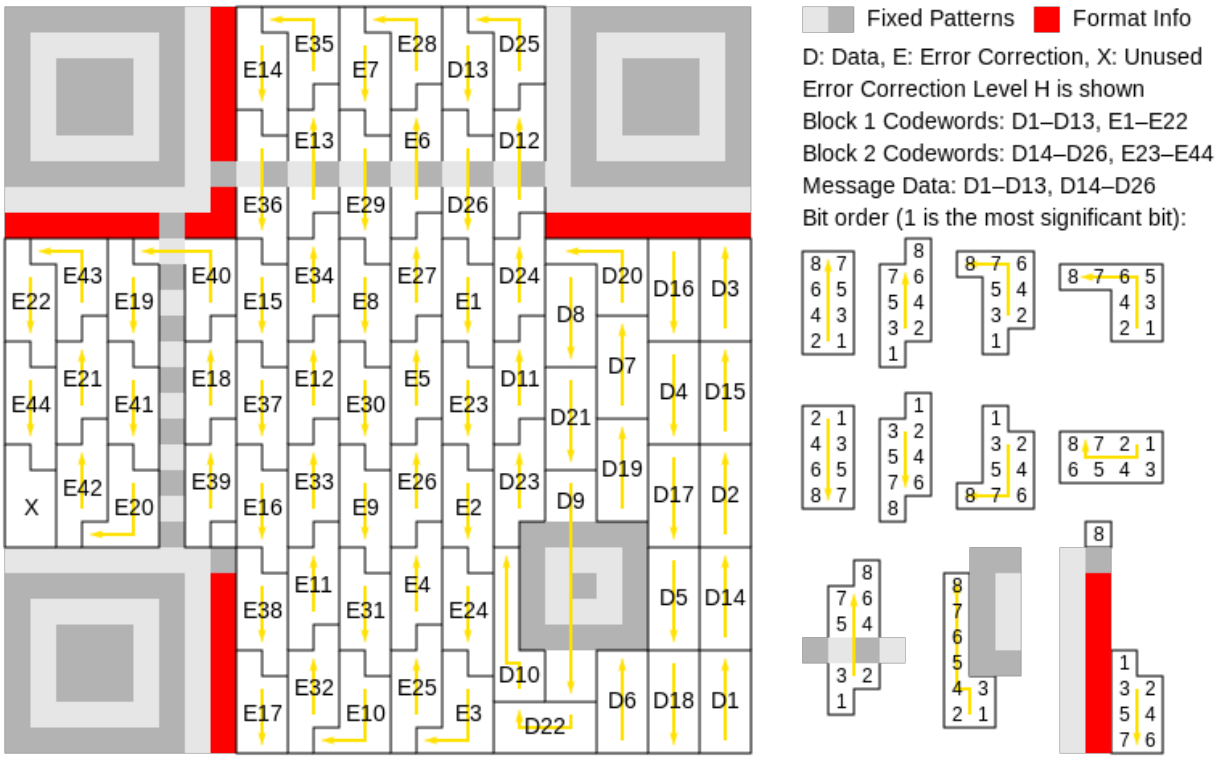
As we can see the mask pattern used on this QR code was (row + column)%3 = 0. This is shown below.



The mask application is a simple XOR operation. So if a square is the filled in the mask and in the QR code it is inverted otherwise it is not changed.We apply the mask to our QR code not including the parts that were hidden by tape in the original image. This gives us the following QR code.

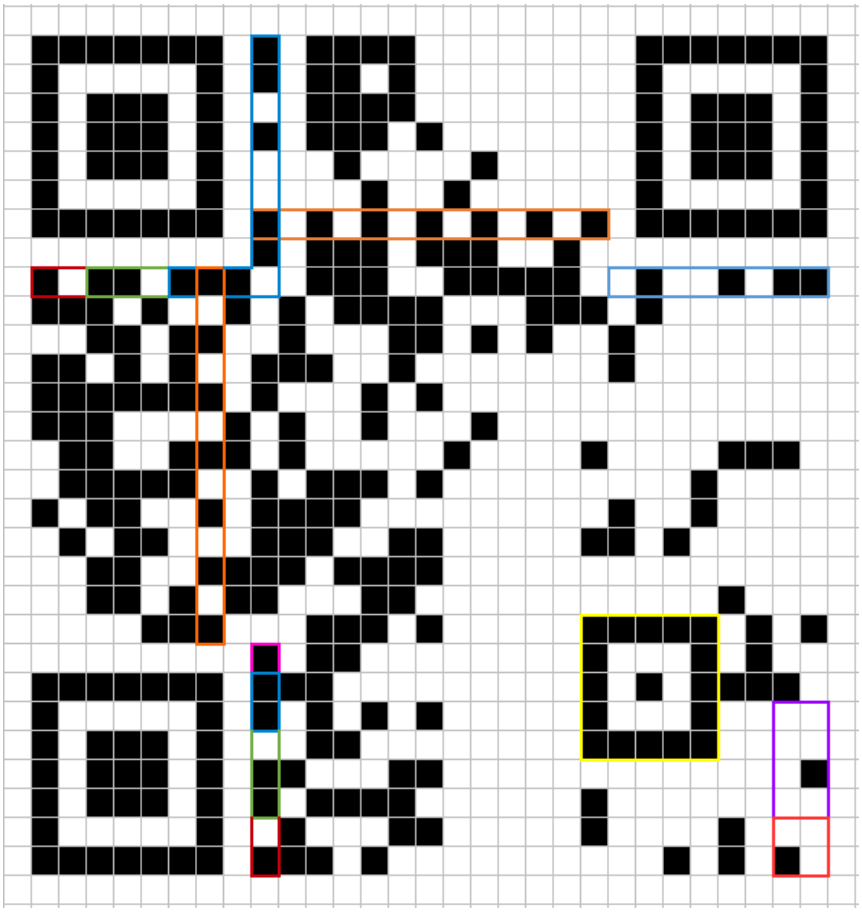


In a QR code data is stored starting from the bottom right corner. It then snakes up and down from right to left in two column rows. The image below is from Wikipedia and shows the way that the data is stored in a QR code.



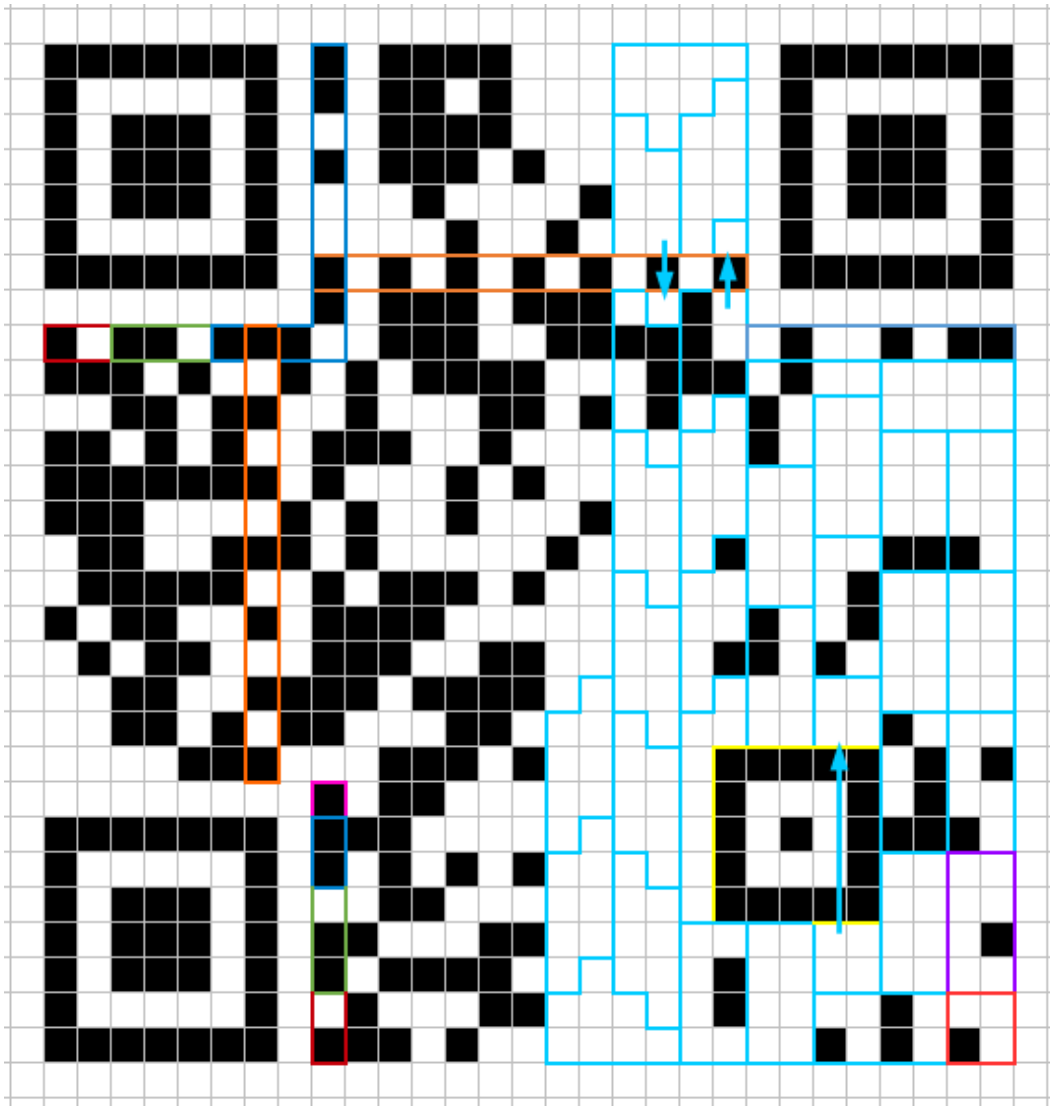
The right hand side of this image indicates the order that bits are read in. However this image is slightly misleading as it does not take into account the four encoding bits at the start or the next eight bits for length.

To interpret our QR code we need to know the encoding of the data and it's length. The encoding is determined by the first four bits starting from the bottom right of the QR code. These are bordered red in the image on the next page. The length is determined by the next eight bits which are bordered in purple in the image on the next page.



Wikipedia has a table that tells us what each encoding scheme is. When reading the bits, the bottom right corner is most significant and we read right to left when going upwards. This gives us 0100 which looking at the table is byte encoding.

Now to read the length we again use the bottom right bit of these eight as the most significant bit. This gives us 0010 0000, which is 32. This is in bytes, so our byte encoded string is 32 bytes long. Knowing this we can map out the 32 characters from our QR code following the pattern in the map from wikipedia. This is shown for our QR code below, each character is bordered in pale blue.



Note: The bits to the left of the characters we have mapped out are the error correction bits. We don't need these to read the characters.

Now we need to read the characters. The first and second character were not obscured so we can read these. The first character is 01001000 or 72 as a decimal number. 72 in ascii is a "H", so the first character is "H".

The second character is 00000000 which in null in ascii. As we can quickly see, in the characters highlighted every second character is a null. This tells us that this string is likely unicode. This is important, because we now know that every second byte should be null so we can start predicting data that should be in the obscured areas.

The next character we know that is not obscured and is not null is the 7th. This character is in a down row, so the top right corner becomes the most significant bit as shown in the diagram earlier in this write up. Decoding this character gives us 01101011 or 107. In ascii this is a “k”.

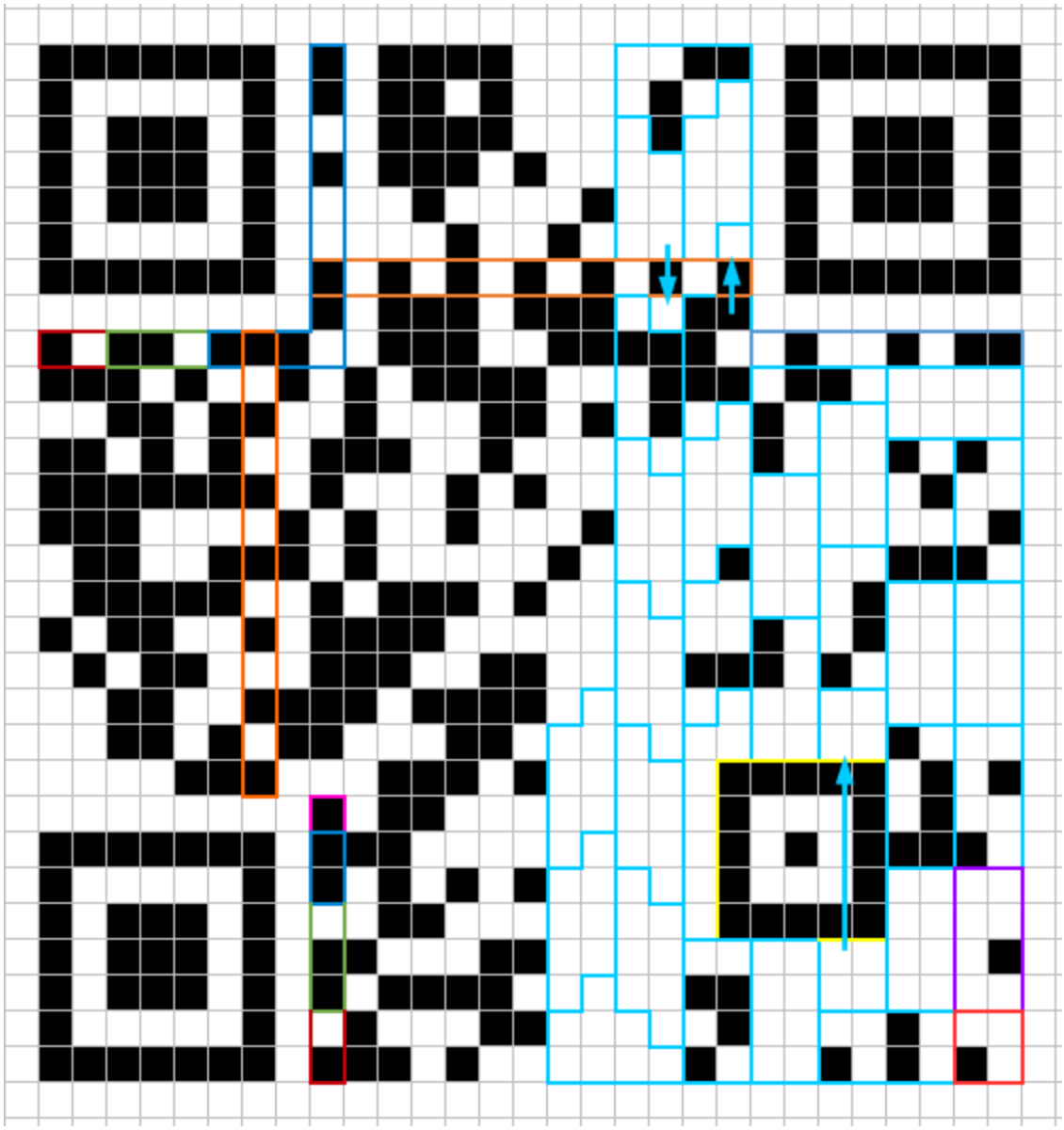
We continue this process to decode the characters that were not obscured in the original image. After completing this we have a unicode string that has the format.

```
“H__kTh_P____t___”
```

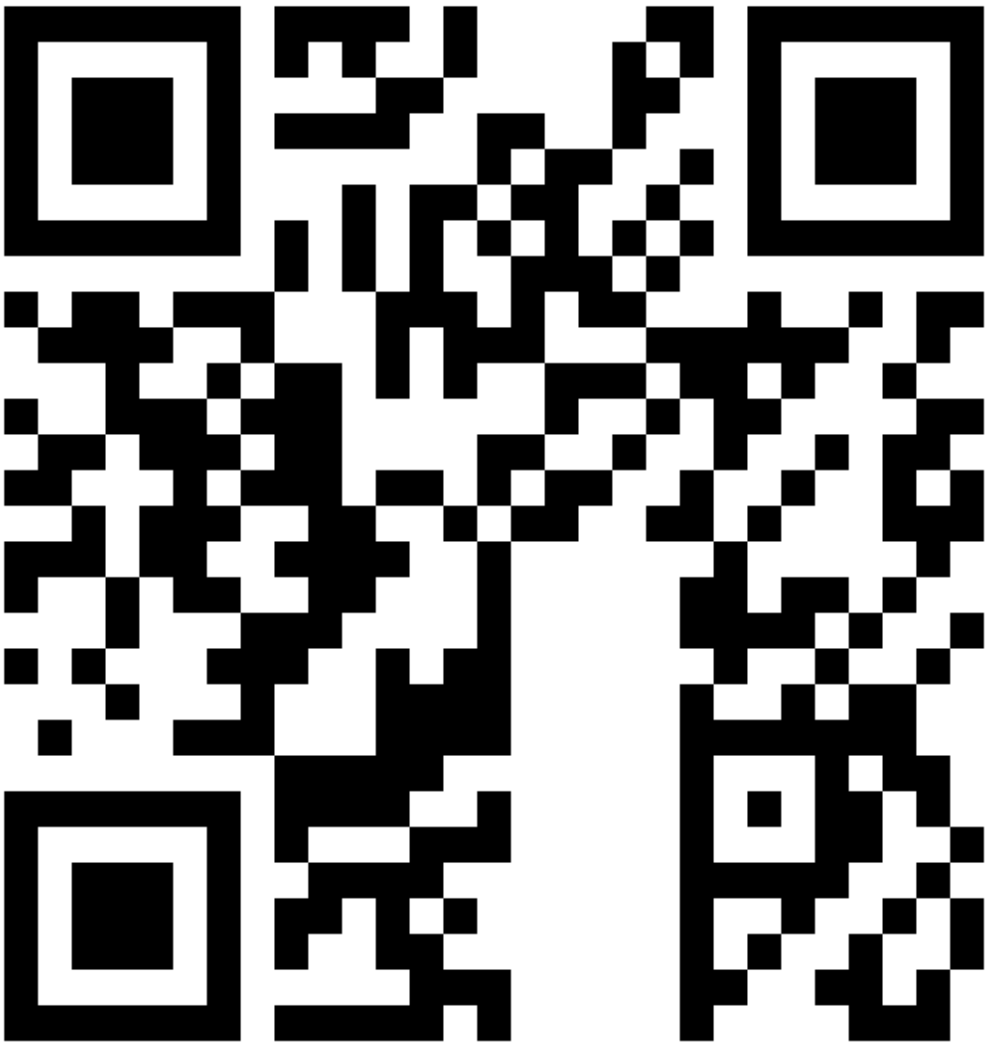
The underscores in this string represent characters that were obscured in the original QR code.

Knowing that the string is unicode, the format of flags in CySCA and that this is a CTF we can infer that the start of the key is most likely “HackThePlanet___”.

We now encode the extra predicted characters and place them back into our QR code. After this process we have the QR code shown in the image below.



However, we still don’t know the three numbers that are at the end of the key. Hopefully we now have predicted enough of the data to allow the error correction to complete the key for us. So we re-apply the mask to the QR code. The image below shows our QR code after this.



Now we can load this into a QR code reader and if you were lucky enough to have a good QR code reader it may have been able to read this and give you the key (We found one that would).

However, if you didn't there are still a little more data we can add in. Remember earlier we said that this was a unicode string. Well as you can see above we still don't have any on data in the missing area where the numbers should be. If every second byte is a null we know that after applying the mask it will look just like the mask. After this process, we are left with the following QR code.

Now there are enough correct bits that most QR code readers can do the error correction and get the key. Reading this with a QR code reader we get the key “**HackThePlanet790**”.