R1 - Challenge 1

Theme: Steganography

Before covering how the audio steganography code works, it pays dividends to understand how audio is stored on disk.

Digital audio works by sampling audio many times per second. Each sample is a signed value that describes a normalised value between -1 and 1. Since this value is a signed integer, we can use steganography to store information in the least significant bit. Commonly, audio is stored as 16-bit "frames" at a framerate of 44.1KHz.

For audio steganography, the capacity of the stored data is given by the song duration multiplied by the sample rate, divided by eight. E.G. for a three-minute song at 44.1KHz, we could encode 992,250 bytes using a least-significant-bit method.

Solution code:

```python
#!/bin/python3
import sys
import numpy
import wave
import struct

fname = sys.argv[1]

waveform = []
waveformParams = None

with wave.open(fname, 'rb') as f:
    print("Width {}.format(f.getsampwidth()))
    print("Sampling Rate {}.format(f.getframerate()))
    print("Frames {}.format(f.getnframes()))
    print("Channels {}.format(f.getnchannels()))

waveformParams = f.getparams()

waveform = f.readframes(waveformParams.nframes)

waveformLength = len(waveform)

if waveformParams.sampwidth == 2:
    floatform = struct.unpack('h' * (waveformLength / waveformParams.sampwidth), waveform)
```

15
else:
    floatform = struct.unpack('b' * waveformLength, waveform)

stegLength = waveformParams.nframes / 8

stegData = numpy.zeros(stegLength, dtype=numpy.uint8)

for i in range(stegLength):
    byteVal = 0
    for shift in range(8):
        t = floatform[i * 8 + shift]
        byteVal += (t & 0b1) << (7 - shift)
    stegData[i] = byteVal

with open("stego.saurus", 'wb') as f:
    f.write(stegData.tobytes())
R1 - Challenge 2

Theme: Cryptography

We captured this suspicious outbound communication to a server. This might be of help to you. Click here for the file.

Hash.txt Contains a long string of decimals for participants to decode.

Test.py Python script which could be written by participants to solve the challenge. Solution: 1. Download the text file and analyse it 2. Copy the string of decimals in the text file and put it in a python script to decrypt the decimals. The script removes the repeating “837” number.

```python
MESSAGE = "6783711118371108371038371148379783711683711783710883797837116837105837111837110837"

PLAINTEXT = ""

DECIMALS_LIST = MESSAGE.split('837')

for DECIMAL in DECIMALS_LIST:
    DECIMAL = int(DECIMAL)
    PLAINTEXT = PLAINTEXT + chr(DECIMAL)

print ("Plain text:

PLAINTEXT = "flag:8ac5f87a2775"

3. Run the script to obtain the flag

```

```

Flag: flag:8ac5f87a2775

```
R1 - Challenge 3

Theme: Web-application

Solution:

1. Part of the flag is hidden in a cookie

2. The other part of the flag is the color of the blue strip in hex

Flag:
flag:2c3d4zb0e0e6
R1 - Challenge 4

**Theme:** Web-application

On access of Challenge 4, we are presented with the Flag Checker service.

By inspecting the source code of the website, we can discover the check flag() function is executed when we click submit.

We can see the check flag() function calls the check1tocheck4 function to see if each part of the flag is correct. It turned out that it was for every printable character, compute its corresponding hash with different hashing algorithms and put them into a dictionary for one to one matching. The script is as follows:

```python
import hashlib import hmac import string

forces every printable character, compute its corresponding hash with different hashing algorithms and put them into a dictionary for one to one matching. The script is as follows:

```

```python
l = string.printable

p1 = [quote; 8fa14cddf754f91ce6554c9e71929c8e7quot; , quot; 2db95e8e1a9267b7a1188556b2013b33quot; , quot; 0cc175b9c0f1b6a83quot; ]
p2 = [quote; 32096c20ef33d844ec6d675407ace18289357dquot; , quot; b6589fc6ab0dc82e91099d1c2d40ab994e8410equot; , quot; 59qu"t]
p3 = [quote; 4e07408562bed86660ce05c1dcefc3ad16b7230967dc01f640b7e4729b49fcequot; , quot; 6b866273ff34fcee19d6b804ef5bquot; ]
p4 = [quote; 01969a94bcf90f8aad43afecfc2046quot; , quot; f832eb995a8ecd24789c022d4e93913bquot; ]


print 39:39: join([ dict[x] for x in (p1 + p2 + p3 + p4) ])

Flag:
flag: C0DoU31rWVGus
R1 - Challenge 5

Theme: Cryptography

Solution: 1. Analyse and understand the JavaScript codes. 2. The first 5 characters of the ciphertext has to be flag: (format for a flag) 3. Write (flag:) in the plain text box and copy the first 5 characters of the ciphertext (àjjÈ4) to be set as the key, thereafter generate a new ciphertext

4. Use the newly generated ciphertext to set the new state by using the inspect element of the webpage. This is to set the state to the position of the flag.

5. Now the next 12 elements of the state will display the flag. 6. Copy the rest of the ciphertext (éç§[ÅÇ]) and click generate key and encrypt to obtain the flag

Flag: flag:49e3395f08eb
R1 - Challenge 6

Theme: Network Traffic Analysis

Tools Used:

https://www.wireshark.org/

Can you find the flag in the captured network traffic below?

Click [here](https://www.wireshark.org/) for the file.

This looks like a pretty normal packet trace... until we go to the preferences and enable IPv4 packet checksum validation!
Now we can see a lot of errors in packets with IP frames.

This looks like text.

It spells out "You're on the right track."

The rest of the packets with invalid checksums aren't text, but Wireshark does tell us what the valid checksum should be...
ASCII Table -- Printable Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>67</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
<td>68</td>
</tr>
<tr>
<td>E</td>
<td>45</td>
<td>69</td>
</tr>
<tr>
<td>F</td>
<td>46</td>
<td>70</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>71</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
<td>72</td>
</tr>
<tr>
<td>I</td>
<td>49</td>
<td>73</td>
</tr>
<tr>
<td>J</td>
<td>4A</td>
<td>74</td>
</tr>
<tr>
<td>K</td>
<td>4B</td>
<td>75</td>
</tr>
<tr>
<td>L</td>
<td>4C</td>
<td>76</td>
</tr>
<tr>
<td>M</td>
<td>4D</td>
<td>77</td>
</tr>
<tr>
<td>N</td>
<td>4E</td>
<td>78</td>
</tr>
<tr>
<td>O</td>
<td>4F</td>
<td>79</td>
</tr>
<tr>
<td>P</td>
<td>50</td>
<td>80</td>
</tr>
<tr>
<td>Q</td>
<td>51</td>
<td>81</td>
</tr>
<tr>
<td>R</td>
<td>52</td>
<td>82</td>
</tr>
<tr>
<td>S</td>
<td>53</td>
<td>83</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
<td>84</td>
</tr>
<tr>
<td>U</td>
<td>55</td>
<td>85</td>
</tr>
<tr>
<td>V</td>
<td>56</td>
<td>86</td>
</tr>
<tr>
<td>W</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>X</td>
<td>58</td>
<td>88</td>
</tr>
<tr>
<td>Y</td>
<td>59</td>
<td>89</td>
</tr>
<tr>
<td>Z</td>
<td>5A</td>
<td>90</td>
</tr>
</tbody>
</table>

Let's take the difference and compare it against our favourite ASCII table: 666C -> “fl”, which is the first two letters of flag!

ASCII Table -- Printable Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>67</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
<td>68</td>
</tr>
<tr>
<td>E</td>
<td>45</td>
<td>69</td>
</tr>
<tr>
<td>F</td>
<td>46</td>
<td>70</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>71</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
<td>72</td>
</tr>
<tr>
<td>I</td>
<td>49</td>
<td>73</td>
</tr>
<tr>
<td>J</td>
<td>4A</td>
<td>74</td>
</tr>
<tr>
<td>K</td>
<td>4B</td>
<td>75</td>
</tr>
<tr>
<td>L</td>
<td>4C</td>
<td>76</td>
</tr>
<tr>
<td>M</td>
<td>4D</td>
<td>77</td>
</tr>
<tr>
<td>N</td>
<td>4E</td>
<td>78</td>
</tr>
<tr>
<td>O</td>
<td>4F</td>
<td>79</td>
</tr>
<tr>
<td>P</td>
<td>50</td>
<td>80</td>
</tr>
<tr>
<td>Q</td>
<td>51</td>
<td>81</td>
</tr>
<tr>
<td>R</td>
<td>52</td>
<td>82</td>
</tr>
<tr>
<td>S</td>
<td>53</td>
<td>83</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
<td>84</td>
</tr>
<tr>
<td>U</td>
<td>55</td>
<td>85</td>
</tr>
<tr>
<td>V</td>
<td>56</td>
<td>86</td>
</tr>
<tr>
<td>W</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>X</td>
<td>58</td>
<td>88</td>
</tr>
</tbody>
</table>

Here's the next few characters decoded. It spells "flag:6" so far... the rest of the characters are the rest of the flag.

Congratulations :)
R1 - Challenge 7

Theme: Cryptography

Beware: What you see is not what it seems!

h2yv:94p6qrs7naeh

The flag is encrypted with a key.

What you are looking for is the answer to these ancient scripts.

Beware: What you see is not what it seems!

h2yv:94p6qrs7naeh

The flag is encrypted with a key.

What you are looking for is the answer to these ancient scripts.

Solution

flag:qwh493dof2c0

This question is not a straightforward question as warned in the puzzle “Beware! What you see is not what it seems!” Participants do not need to solve the hieroglyphs to get the flag. However, the hieroglyphs serve as a clue that the decoder used needs a key.


Answer: Khnumhotep II

Cipher used: Vinegère Cipher Key: cryptii (default key from cryptii.com)

You will need to append 0123456789 to the end of the alphabet since it is an alphanumeric cipher.
R1 - Challenge 8

Theme: Reverse Engineering

In the reversing challenge, the goal was to extract a 128-bit AES key from the binary and use it to communicate with the C2 server. The purpose of the challenge was to show that, alone, a secure cypher mode is not a sufficient authentication factor.

The intended solution was the use of Ghidra; however, other options of extracting the keys are equally as valid. To this end, the binary utilises a few anti-debugging techniques.

The key is constructed from three locations within the binary and loaded into memory. From here, OpenSSL talks to the C2 server transmitted an AES-128-GCM encrypted packet containing uint32_t(1). If a packet is sent containing uint32_t(0), the C2 server will return the flag. The demo codeskipstherecoveryofthekeybutshowsthemethodsendsendingdatatotheserver.

Solution Code: getFlag.py

```python
from cryptography.hazmat.primitives.ciphers.aead import AESGCM import os import struct import requests

# Read the key from secret.key
key = open("secret.key", "rb").read()

# The request for a flag is uint32_t(0). AEAD is used, so flipping a cyphertext bit won't work.
context = AESGCM(key)

# Payload and IV
payload = struct.pack("I", 0)
iv = os.urandom(16)

# Encrypt and append cyphertext
creptertext = iv + context.encrypt(iv, payload, None)

# Get the flag

data = decrypt(flag)

# Decrypt it!
print(data)
```

25
R1 - Challenge 10

Theme: Shredded File

Tools Used: To solve the challenge, we first run the binary file.

The above image shows the result of running shred.bin. The text is “Ooops, your important files are shredded. To obtain the shredded parts, you need to pay xxxx NZD. The price can be negotiated :D “ Of course, we will not be paying to retrieve the file back, but we could try running “strings” on the file.

After running the “strings” command, you will find that the file contains .zip file. We could extract the .zip file by using online tools like CyberChef. Extracting .zip file will result in 10 files of secret where each of the files contains some form of string. The string is encoded Base64.

Since the original binary file’s name is shred, we could expect that an original secret file is shredded into these 10 parts of secret. However, concatenating these 10 parts of shredded will not work. The result of running shred.bin is a hint to solve this challenge. As can be seen that the result of running shred.bin contains 10 vertical lines. This hints that the first character in the original file will be in the first shredded part, the second character of the original file will be in the second shredded part, so on and so forth. We can combine the files manually or write a script to work for us. The below python script can be used to combine the files.

```
#!/usr/bin/python3
files = []
for i in range(10): with open(quot;secret.partquot; + str(i+1), quot;rquot;) as file: files.append(file.read())
output = 39;39;
for char in range(len(files[0])): for i in range(10): if char == len(files[i]): continue else: output += files[i][char]
print(output)
```

This results in

```
QSBwYXBlciBzaHJlZGRlciBpcyBhIG1lY2hhbmljYWwgZGV2aWNlIHVzZWQgdG8gYiB0IHBhcGVyIGIudG8gZW1lbnltbGllcyB0byB0aGluZzBhcHBpbmdsZSByZXN0aW9uc29sZCByZXRhcmRlbGVzZGVzIHN0cmlwcyBvciBmaW5lIHNvdW50YWJsZSByZXR0aXItc2hyb2ZrIHRoZSBzdWUgaXMgY2FyaWluZyB0byBtb3ZlbnRzIHRoZSBzaGF0ZSBhIHZhbiByZXRlZ2F0ZSByZXR0aXItbG9jaA==
```

Finally, online tool such as Cryptii, can be used to decode the Base64 string. The decoded string contains flag, flag:6sg4s1ax0n2, at the end.
R1 - Challenge 11

Bob's computer has been pwned and some of his important files were encrypted by a ransomware. Can you help him retrieve the data from the memory dump?
Hint: Bob loves Notepad

Click here for the memory file.

In challenge 11 we are presented with a memory file. The unintended solution is to run either strings or grep on it and the flag is shown in plaintext. The intended solution is described as follows. We first download the memory file and used volatility to identify the profile using the imageinfo option.

We can then perform more analysis using this profile option. For example, we can observe the various commands that were executed using the cmdscan utility.

From here, we can observe several interesting things. 1. There is a powershell command that was executed 2. We know the content of key.txt We can base64 decode the powershell command to see what it’s doing.

It is downloading a ransomware.exe from a url. We can then proceed to download that ransomware. Once we obtain the executable, we can do a simple strings analysis on it. We could observe several python libraries in the strings output.

We assume it was compiled with pyinstaller and proceed to decompiling it. We could use a Pyinstaller Extractor (https://github.com/extremecoders-re/pyinstxtractor) for this.

We have now obtained the compiled bytecode file ransomware.pyc. To decompile this back to source file, we can use the tool decompyle3 for this (https://github.com/rocky/python-decompile3).

We can see from the source code that it’s encrypting files in AES CBC mode with an IV of ‘abcdefghijkmnop’ hardcoded in the source file. It also writes the sentence “Encrypted Data” before the encrypted data. Now we have the key and the IV to decrypt the encrypted files. As the hint suggested, we should have a look at the Notepad memory. We can use volatility for this.

We first list the processes using pslist. And then find the PID of notepad and dump the memory of it. We can do a strings on the memory dump because the encrypted data was written to the file in base64.

Next we can open up strings.dmp and search for the text “Encrypted Data”. We will see the encrypted data in base64.

Finally, we use CyberChef to decrypt the data.

Flag: flag:RUpF6X0dntqV
This is a Server-Side Template Injection (SSTI) challenge. There are various ways to solve this, but all solutions were based on the insecure usage of the `render_template_string()` function. Upon visiting the website, we are presented with a text saying the website is under construction.

However, robots.txt showed two hidden directories.

Navigating to the `/secretagent` directory, we can observe the webapp. When clicking the button, it shows our User Agent string.

The other directory `/secretsource` shows the source code of this webapp. It shows that it’s a simple Flask app. During the analysis, we can observe several interesting things.

1. The flag is read from `app/flag.txt` and stored in the `secret` variable in the app’s config. 2. The User Agent string is passed to an initial filter which rejects any strings with unwanted characters or strings. 3. A length check of 70 is then performed on the User Agent string. 4. If the string “s3cr3tAg3nt” is present in the User Agent string, it returns a fake flag.

The vulnerability lies in the usage of `render_template_string`, which allowed code injection from user-supplied inputs. We could observe this by capturing the request in Burp and sending it to Repeater.

We can see that the supplied input `7*7` was evaluated to 49. This confirms the vulnerability and we can proceed to exploitation.

Normally, we could just read the config variable by the payload `config`. However, since it is disallowed, we could not use it.

However, we could reference the config variable from the `url_for` function’s `_globals_` attribute. So an ideal payload would be the following.

However, we still need to bypass the WAF. Fortunately we can use the disallowed strings and characters by referencing from the `request.args` variables.

Now all that is left is to bypass the length restriction. We can overcome this by setting variables in Jinja. For example, rest of the payload. We can also add the `secret` variable for displaying it more clearly.

Flag: `flag:cf11msfpo2b`