Theme: Steganography

Find	the flag in t	he audio fi	le:		
Click	here to dov	wnload the	audio.		
•	0:00 / 3:31			:	

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:

```
#!/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)
```

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())</pre>
```

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.

PLAINTEXT = ""

```
DECIMALS_LIST = MESSAGE. split('837')
```

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

```
print ("Plain_text:\n" + PLAINTEXT)
```

3. Run the script to obtain the flag



Flag:

flag:8ac5f87a2775

Theme: Web-application

Cookies are not cream!	
More info here! >>	

Solution:

1. Part of the flag is hidden in a cookie

\square				<u>_</u>			
🕞 🗘 Inspector	> Console	Debugger	↑↓ Network	{ } Style Edi	tor 📿 F	erformance	(): M
E Cache Storage		🗑 Filter Iter	ms				
- E Cookies		Name	Value	Domain	Path	Expires / Max	-Age
https://r1.nzcs	sc.org.nz		firstpart:2c3d4z	r1.nzcsc.org	/challen	Session	
Indexed DB							
🕨 🗄 Local Storage							
Session Storage	1						

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

		\sim	
Inspector Console Debugger ↑↓ Network	<pre>{} Style Editor</pre>	Performance	D: Memory
Inspector Console Debugger ↑↓ Network Q Search HTML + ✓	Style Editor Filter Styles	Performance	:hov .cls +
Image: Console Debugger ↑↓ Network Q Search HTML + ✔ <1DOCTYPE	<pre>{} Style Editor ♥ Filter Styles element ↓ { }</pre>	Performance	C Memory (E :hov .cls +
Console Debugger ↑↓ Network Q Search HTML + ★ html + ★ <html lang="en"> event scroll ▶ <head> = ▶ <div class="navbar"> = mavbar"> = // div< flex</div></head></html>	<pre>{} Style Editor \$\vee Filter Styles element \$\vee { } .page_content background-c }</pre>	Performance container { color: #b0e0e6; }	:hov.cls + inccss1.
Console Debugger ↑↓ Network Q Search HTML + // html + // <html lang='ten"'> event scroll + ead> / + ead> <!--</td--><td><pre>{} Style Editor \$\[\frac{Filter Styles element \$\[\frac{1}{2}\] ,page_content background-c } .page_content_ background-c }</pre></td><td><pre> Performance _container \$</pre></td><td><pre> * Memory * hov .cls + i css1. web.cs </pre></td></html>	<pre>{} Style Editor \$\[\frac{Filter Styles element \$\[\frac{1}{2}\] ,page_content background-c } .page_content_ background-c }</pre>	<pre> Performance _container \$</pre>	<pre> * Memory * hov .cls + i css1. web.cs </pre>
Image: Console Debugger ↓ Network Q Search HTML + ✓ html + ✓ <html lang="en"> event scroll + ✓ > <head> @ </head> *<ebody> > * <dbody> > > <div class="navbar">@ * <div class="page_content">flex * <td< td=""><td><pre>{} Style Editor \$\vee Filter Styles element \$\phi { } .page_content background-c } .page_content_ background-c margin-top: overflow: > } }</pre></td><td><pre> Performance _container</pre></td><td>Memory E thov .cls + i ccssl. web.cs</td></td<></div></div></dbody></ebody></html>	<pre>{} Style Editor \$\vee Filter Styles element \$\phi { } .page_content background-c } .page_content_ background-c margin-top: overflow: > } }</pre>	<pre> Performance _container</pre>	Memory E thov .cls + i ccssl. web.cs

Flag:

flag:2c3d4zb0e0e6

Theme: Web-application

Bob has developed a flag checker. Click here to access the service.

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 $check_f lag() function is executed when we click submit.$

We can see the check $_{f}lag()$ function calls the check $_{1}$ to check $_{4}$ functions to see if each part of the flag is correct. It turned out that it was force severy printable character, compute its corresponding has hwith different has hing algorithms and put the minto adjust in a set of the flag is correct. The set of the flag is correct is the set of the flag is correct. The set of the flag is correct is correct in the set of the flag is correct. It is consistent of the flag is correct. The set of the flag is correct is correct in the set of the flag is correct. The set of the flag is correct is correct in the set of the flag is correct. The set of the flag is correct is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the flag is correct in the set of the flag is correct. The set of the flag is correct in the set of the fl

import hashlib import hmac import string

l = string.printable

 $\mathbf{p}_1 = [quot; 8fa14cdd754f91cc6554c9e71929cce7quot;, quot; 2db95e8e1a9267b7a1188556b2013b33quot;, quot; 0cc175b9c0f1b6a83]$

 $\mathbf{p}_2 = [quot; 32096c2e0eff 33d844ee6d675407ace18289357dquot;, quot; b6589fc6ab0dc82cf12099d1c2d40ab994e8410cquot;, quot; b6589fc6ab0dc82cf12099d1c2d40ab994e8410cquot; quot; b6589fc6ab0dc82cf12099d1c2d40ab994e8410cquot; quot; b6589fc6ab0dc82cf12090dc82cf1209d1c2d40ab94e8410cquot; quot; b6589fc6ab0dc82cf1209d1c2d40ab94e8410cquot; quot; b6589fc6ab0dc82cf1209d1c2d40ab94e8410cquot; quot; quo$

 $p_4 = [quot; 01969a94bcf90f8aad4c3afefc7bc046quot;, quot; f832cb995a8ecd24789c022d4c93913bquot;]$

print 39;39;.join($[dict_1[x]forxin(p_1 + p_2 + p_3 + p_4)]$)

Flag:

flag:CODoU31rWVGus

Theme: Cryptography

	\sim	
Plain text :		
Key :		Generate Key
Cipher text :		Encrypt
EXAM	PLES:	Cinhortext
Flainteat	Key	
NZCSC'20	LƦcCmZD	CŅé0Cøht
Cyber Security	4ù[R.ņµÉòĿw%%&	wÒ97\ýäæľeĴLQ_
Cryptography	ņZnĻbªGEYÅèղ	Ü(ťzk <u>K</u> 78õĐ»

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 (àĵjÈ4) 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 ($\dot{e}_{\dot{c}}$ $\hat{s}[\hat{A}\zeta)$ and click generate key and encrypt to obtain the flag

Flag: flag:49e3395f08eb

Theme: Network Traffic Analysis

Tools Used:

https://www.wireshark.org/

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

Click here for the file.



•	Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT) Total Length: 201 Totalicsticn: 0x0000 (0) Flags: 0x4000, Don't fragment Tume to live: 64	
•	<pre>protocol: UW [1] Stability incorrect, should be 0x797b(may be caused by "IP checksum offlond"?) [[Gader: Inchecksum: 5x787b] Source: 192.188.9.1</pre>	
0000 0019 0020 0030 0040 0050 0050 0050 0080 0090 0020 0020 0040	Destination: 25: 25: 25: 25: 25: 25: 25: 25: 25: 25	
•	* complex.pcapng Packets: 24326 · Displayed: 24326 (100.0%) Profile: Def complex.pcapng –	ault
<u>F</u> ile	Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help $\boxed{0}$ (\textcircled{O}) $\boxed{1}$ (\textcircled{O}) $\boxed{2}$ (\textcircled{A} A \textcircled{A} A	
No.	ply a display filter <ctrl-></ctrl-> Ctrl-/> Expression Time Source Destination Protocol Length Info 10 a control top 24 60 arXiv:155 025 102 42 60 arXiv:155 025 245 62006 arXiv:173	+
	2 0 326655 192 168 0 131 192 168 0 131 192 165 0 156 170 170 170 185 0 170 170 189 175 0 1	
	11 1.69762 192,188,0,31 162,159,136,224 170 54 33744443 [ACK] Seq:1 Ack=365 Win=2227 Len=0 12.198464 152,159,126,124 152,1268,0,131 1152,12 45 Application Data 13.198463 152,159,124 152,159,131 152,159,151 153 175,12 45 Application Data 13.198463 152,151 153,151 152,159,151 153 154 175,12 45 Application Data 152,12155 123,158,131 153,159,158,234 175 154 154 154 154 154 154 154 154 154 15	
	12 2,432449 UDIQUIT_10170147 4 FORDERSI AND 10 4 MO MAS 102,108,01141 102,108,0144 4 FORDERSI 11 102,108,0144 4 FORDESSI 11 102,104,0144 4 FORDESSI 11 102,104,0144 4 FORDESSI 11 102,1044 4 FORDESSI 11 104,1044	-
Fr Et In F	me 1: zis bytes on wire (1/20 bits), zis bytes captureb (1/20 bits) erref 11, Src: DyLinkT_checkec (e: 60:60:60:fis160:ec), bits: Broadcast 6160 = Version: 4 Persion: 4 Persion: 5 Persion 200, 200, 200, 200, 200, 200, 200, 200	
0000	[Header checksum status: Bad] [Calculated Checksum: status: Bad] [Source: 192.168.0.1 Destination: 255.255.255.255 Pestination: 255.255.255.255 0 c9 80 00 40 50 00 421 10 5001 cd as 80 01 ff ff	v
0020 0030 0040 0050 0060 0070 0080 0080 0080 0080 0080 008	ff ff 90 66 10 b0 60 10 c0 86 47 40 41 44 4e 4f 55 ff ff 90 66 110 c0 80 90 90 cc 086 45 ff ff 90 66 112 c0 80 90 90 cc 086 45 ff ff 90 66 112 ff 100 6	
	It spells out "You're on the right track"	
•	Header checksum (ip.checksum), 2 bytes Packets: 24326 · Displayed: 24326 (100.0%) Profile: Def complex.pcapng -	ault ×
<u>F</u> ile	Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help Image:	
No.	ply a display filter <ch></ch> Time Source Destination Protocol LengthInfo 15 - 086666 1 022-168-0-1 225,225,225 UP 215 38966 - 7437 Lene173 UP 215 38966 - 7437 Lene173 UP 215 38966 - 7437 Lene174 Lene174 Lene175 Total-2550000007 Total-2550000007 Total-25500000007 Total-25500000007 Total-25500000007 Total-255000000007 Total-255000000007 Total-255000000007 Total-255000000007 Total-255000000007 Total-25500000007 Total-25500000000000000000000000000000000000	+
[5 6:328225 142:168.0.156 132:168.0.131 TOP 176 8080 - 34536 [DSW, AAX] [Swith Ack-111 Win-243 Lensib [Swith 2556829 Taker-255869000] [CP segment of a reassembled Poul 6 0:32855 142:168.0.131 122:168.0.131 TOP 6 8040 - 34536 [DSW, AAX] [Swith Ack-111 Win-243 Lensib [Swith 2556829 Taker-255869000] [CP segment of a reassembled Poul 6 0:32855 142:168.0.131 122:168.0.131 TOP 6 83458 - 8060 [AAX] [Swith Ack-111 Win-243 Lensib [Swith 2556829 Taker-255869000] [CP segment of a reassembled Poul 6 0:32855 142:168.0.131 122:168.0.131 TLYL.2 155 Application Data 6 1.167376 122:169:188.234 192:168.0.131 TLYL.2 155 Application Data 7 1.171419 152:159:186.234 192:168.0.131 TLYL.2 155 Application Data 7 1.171419 152:159:186.234 192:168.0.131 TLYL.2 155 Application Data 7 1.171419 152:159:168.234 192:168.0.131 TLYL.2 155 Application Data	
	101.607305 102.159.136.234 102.168.6.131 112.7.2 252 Application Data 11.60765 102.168.0.131 102.169.131 112.7.169.130.234 170 43744 -443 [AcK] Seq:1 Ack=361 Win=2227 Len=9 13.1.908466 102.159.136.234 102.168.0.131 112.7.2 98 Application Data 13.908466 102.159.136.234 102.168.0.133 112.7.2 98 Application Data 13.908466 102.159.136.234 102.168.0.133 112.7.2 98 Application Data 13.908466 102.159.136.234 102.168.0.133 112.7.2 98 Application Data 13.908467 102.168.0.134 102.159.1368.244 107 12.7.2 12.7.2.7.2 10.7	
	17 2.108000 102.10800.155 102.168.0.131 100 66 TCP Atked unisens segment) 8080 - 34574 [AtkK] Seq=1 Ack=2 Win=243 Len=0 Tsval=2560913 Tsecr=2524080246 12 2.302240 105.2382240 105.01157 [Cl1112] (105.01163) 104 105.01157 [Cl1112] (105.01163) 12 2.802740 105.108.01157 [Cl1112] (105.01163) 104 105.01157 [Cl1112] (105.01163) 105.01163 21 2.007375 105.108.0131 102.159.158.013 102.159.158.013 102.159.158.013 102.159.158.013 22 2.007376 102.158.0.314 102.159.138.0.244 TCP 54.33444 - 443 [Ack] Seq=1 Ack=39 Wiln=2227 Len=9 22 2.007376 102.158.0.31 102.159.136.0.244 TCP 54.33744 - 443 [Ack] Seq=1 Ack=39 Wiln=2227 Len=9 22 2.007376 102.158.0.31 102.159.136.0.244 TCP 54.33741 - 443 [Ack] Seq=1 Ack=39 Wiln=227 Len=9 22 2.007376 102.158.0.31 102.159.136.0.244 TCP 54.33741 - 443 [Ack] Seq=1 Ack=39 Wiln=227 Len=9 22 2.007376 102.159.136.0.31 102.159.136.0.244 TCP 54.33741 - 443 [Ack] Seq=1 Ack=30 Wiln=227 Len=9 23 2.7050599 Xiaon10.0.471011ab (Gita Poly Ulter) TSF55656 AB9 G0 Win Chas 1322.21869.0.107 104	-
Fr ► Et ▼ In	cre.cr90ulc 0ugaery_gr:10:05 AldOHLU_gr:10:100 NPV 42 192.100.0.4.131 1S &1 10:10190197175:03 mon 15: 56 bytes on wire (425 bits), 56 bytes captured (422 bits) NPV exected (422 bits), 56 bytes captured (422 bits) NPV exected (422 bits), 56 bytes captured (422 bits), 56 bytes (422 bits), 56 bytes captured (422 bits), 56 byte	•
+ +	<pre> 0101 = Header Length: 20 bytes (5) Differentiated Services Field: & RADE (DSCP: CSB, ECN: Not-ECT) Total Length: 40 Identification: RASCFA (15610) Flags: RADEAD, Don't fragment Time to live: 64 Protocol: TCP (6) Header checks(e: 0x7784 incorrect, should be 0x1121(may be caused by "IP checksum offload"?)</pre>	
0000 0010 0020	Header checksm: status: stat [Calculated Checksm: status: stat] Bestination: 152,159,136,234 ec 68 6b fa 6e ec 1c 10 ed of f 56 38 60 45 00 + k	-
9939	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

Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal
	20	32	@	40	64	•	60	96
!	21	33	Α	41	65	a	61	97
"	22	34	В	42	66	b	62	98
#	23	35	С	43	67	C	63	99
\$	24	36	D	44	68	d	64	100
%	25	37	E	45	69	е	65	101
&	26	38	F	46	70	f	66	102
•	27	39	G	47	71	g	67	103
(28	40	н	48	72	h	68	104
)	29	41	I	49	73	i	69	105
*	2a	42	J	4a	74	j	6a	106
+	2b	43	к	4b	75	k	6b	107
,	2c	44	L	4c	76	I	6c	108
-	2d	45	M	4d	77	m	6d	109
	2e	46	N	4e	78	n	6e	110
I	2f	47	0	4f	79	0	6f	111
0	30	48	P	50	80	p	70	112
1	31	49	Q	51	81	q	71	113
2	32	50	R	52	82	r	72	114
3	33	51	S	53	83	s	73	115
4	34	52	Т	54	84	t	74	116
5	35	53	U	55	85	u	75	117
6	36	54	V	56	86	v	76	118
7	37	55	W	57	87	w	77	119
8	38	56	X	58	88	x	78	120
9	39	57	Y	59	89	У	79	121
:	3a	58	Z	5a	90	z	7a	122
;	3b	59	[5b	91	{	7b	123
<	3c	60	١	5c	92	I	7c	124
=	3d	61]	5d	93	}	7d	125
>	3e	62	۸	5e	94	~	7e	126
?	3f	63		5f	95	Delete	7f	127

			F	rogrammi	ng Mode	~			_ ×
778D-1121 = 666C									
									666C
Hexadeo	Hexadecimal ▼ 63154# = 2622010 0000 0000 0000 0000 0000 0000 0000								
√n	31 		x •	(15	< ¥	> •	0 á	
с	D	E	F	+	mod	ones	twos	x	
8	9	A	в	*	AND	NOT	1	xy	x ⁻¹
4	5	6	7	-	OR	Ø	log	ln	int
0	1	2	3	+	XOR	=	fact	x!	frac

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

Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decima
	20	32	@	40	64	•	60	96
!	21	33	Α	41	65	a	61	97
	22	34	В	42	66	b	62	98
#	23	35	С	43	67	C	63	99
\$	24	36	D	44	68	d	64	100
%	25	37	E	45	69	е	65	101
&	26	38	F	46	70	f	66	102
•	27	39	G	47	71	g	67	103
(28	40	Н	48	72	h	68	104
)	29	41	1	49	73	i	69	105
*	2a	42	J	4a	74	j	6a	106
+	2b	43	к	4b	75	k	6b	107
,	2c	44	L	4c	76	1	6c	108
-	2d	45	M	4d	77	m	6d	109
	2e	46	N	4e	78	n	6e	110
1	2f	47	0	4f	79	0	6f	111
0	30	48	P	50	80	p	70	112
1	31	49	Q	51	81	q	71	113
2	32	50	R	52	82	r	72	114
3	33	51	S	53	83	s	73	115
4	34	52	Т	54	84	t	74	116
5	35	53	U	55	85	u	75	117
6	36	54	V	56	86	v	76	118
7	37	55	w	57	87	w	77	119
8	38	56	x	58	88	x	78	120
9	39	57	Y	59	89	v	79	121
:	3a	58	Z	5a	90	z	7a	122
:	3b	59	[5b	91	{	7b	123
<	3c	60	1	5c	92	1	7c	124
-	3d	61	1	5d	93	}	7d	125
>	3e	62	^	5e	94	~	7e	126
2	24	62		F f	05	Delete	74	127

			Р	rogrammi	ng Mode	~			- ×
778D-11	21						= 666	SC	
A14B-3F	E4						= 616	57	
76DF-30	A9						= 3A3	36	
									3A36
Hexade	cimal 🔻							35066	в = 14902
	0	000 00	00 000	0 0000	0000	0000 00	000 000	0 0	
	6	3			47			32	
	3:	1	00 000	0 0000	15	1010 00	011 011	0	
√n	∱n)	× •	()	< ¥	> ~	á	
С	D	E	F	÷	mod	ones	twos	×	
	9	A	В	×	AND	NOT	1	xy	x ⁻¹
8									
8	5	6	7	-	OR	Ø	log	ln	int

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 :)

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.

Hieroglyphs – In Whose Tomb Did Cryptography First Discovered? https://discoveringegypt.com/Hieroglyph-typewriter-ipad.html

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.

Theme: Reverse Engineering

<u>Cli</u>	<u>ck here to do</u>	wnload the	<u>binary</u>

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 uint $32_t(1)$. If a packet is sent containing uint $32_t(0)$, the C

Solution Code: getFlag.py

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

The Key needs to be reversed The solution has this key in secret.key key = quot;quot; with open(quot;secret.keyquot;, quot;rbquot;) as f: key = f.read()

The request for a flag is $uint32_t(0)$ AEAD is used, so flipping a cyphertext bit won 39; two rk

Context aesgcmctx = AESGCM(key)

Payload and IV payload = struct.pack(quot;Iquot;, 0) iv = os.urandom(16)

Encrypt and append cyphertext = iv + aesgcmctx.encrypt(iv, payload, None)

Get the flag cryptflag = requests.post(quot;http://sushi.nzcsc.org.nz/c2quot;, data=cyphertext).content

 $Decrypt it! \ cfiv = cryptflag[:16] \ cfdata = cryptflag[16:] \ print(quot;Got: \ quot;.format(aesgcmctx.decrypt(cfiv, cfdata, None).decode()))$

Theme: Shredded File

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

Oops we shredded the flag Click here for the 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

Finally, online tool such as Cryptii, can be used to decode the Base64 string. The decoded string contains flag,

flag:6sg4s1ax0n2, at the end.

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 'abcdefghijklmnop' 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 website is under construction!!!

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:cjf1nnsfpo2b