***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***

07CTF 2025

A black screen with white numbers

AI-generated content may be incorrect.

Categories

[Misc 2](#_Toc208852800)

[A Star is Born 2](#_Toc208852801)

[Forensics 3](#_Toc208852802)

[pd\_what 3](#_Toc208852803)

[Rev 4](#_Toc208852804)

[Camel in Mumbai 4](#_Toc208852805)

[ftw 4](#_Toc208852806)

[Crypto 8](#_Toc208852807)

[Beta Bet 8](#_Toc208852808)

[Not a backdoor 10](#_Toc208852809)

[Web 14](#_Toc208852810)

[The Agreement 14](#_Toc208852811)

[Worse than SQLi 14](#_Toc208852812)

[Cryptic Mistake 15](#_Toc208852813)

## Misc

### A Star is Born

A screenshot of a computer

AI-generated content may be incorrect.

## Forensics

### pd\_what

A screenshot of a computer

AI-generated content may be incorrect.

## Rev

### Camel in Mumbai

Solve script:

1. import sys, struct, itertools

2.

3. ANCHOR = b"camels are lovely aren't they?"

4.

5. def read\_qword(buf, off):

6. return struct.unpack\_from("<Q", buf, off)[0]

7.

8. def read\_array\_at(buf, off):

9. vals=[]

10. while True:

11. q = read\_qword(buf, off)

12. off += 8

13. if q == 0:

14. break

15. vals.append(q)

16. return vals

17.

18. def ints\_to\_bytes(ints):

19. return bytes(((i >> 1) & 0xFF) for i in ints)

20.

21. def main():

22. path = sys.argv[1] if len(sys.argv) > 1 else "./camel\_in\_mumbai"

23. buf = open(path, "rb").read()

24. p = buf.find(ANCHOR)

25. if p < 0:

26. raise SystemExit("Anchor string not found.")

27. # Offsets relative to the anchor position in this binary (file offsets)

28. arr1\_off = p + 40 # 0x28 after the anchor start

29. arr2\_off = p + 296 # 0x128 after the anchor start

30. arr1 = read\_array\_at(buf, arr1\_off)

31. arr2 = read\_array\_at(buf, arr2\_off)

32. xored = bytes(a ^ b for a,b in zip(ints\_to\_bytes(arr1), itertools.cycle(ints\_to\_bytes(arr2))))

33. flag = (xored.split(b'}',1)[0] + b'}').decode('utf-8', 'ignore')

34. print(flag)

35.

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

37. main()

### ftw

Solve script:

1. import json, math, base64

2. from Crypto.Cipher import AES

3. from Crypto.Util.Padding import unpad

4. from fastecdsa.curve import P256

5. from fastecdsa.point import Point

6.

7. MASK\_30B = (1 << (8\*30)) - 1 # lower 30 bytes

8.

9. # ---- number theory helpers ----

10. def tonelli\_shanks(n, p):

11. """Return y with y^2 == n (mod p) or None if no root; works for odd prime p."""

12. if n % p == 0:

13. return 0

14. # Legendre symbol check

15. ls = pow(n, (p - 1) // 2, p)

16. if ls != 1:

17. return None

18. # p-1 = q \* 2^s with q odd

19. q = p - 1

20. s = 0

21. while q % 2 == 0:

22. q //= 2

23. s += 1

24. # find z a quadratic non-residue

25. z = 2

26. while pow(z, (p - 1) // 2, p) != p - 1:

27. z += 1

28. c = pow(z, q, p)

29. x = pow(n, (q + 1) // 2, p)

30. t = pow(n, q, p)

31. m = s

32. while t != 1:

33. # find least i (0 < i < m) with t^(2^i) == 1

34. i = 1

35. t2i = (t \* t) % p

36. while i < m and t2i != 1:

37. t2i = (t2i \* t2i) % p

38. i += 1

39. b = pow(c, 1 << (m - i - 1), p)

40. x = (x \* b) % p

41. c = (b \* b) % p

42. t = (t \* c) % p

43. m = i

44. return x

45.

46. def point\_from\_x(curve, x):

47. """Return the two curve points with this x, or [] if none."""

48. p = curve.p

49. a = curve.a

50. b = curve.b

51. rhs = (pow(x, 3, p) + (a \* x) + b) % p

52. y = tonelli\_shanks(rhs, p)

53. if y is None:

54. return []

55. if y == 0:

56. return [Point(x, 0, curve=curve)]

57. return [Point(x, y, curve=curve), Point(x, (-y) % p, curve=curve)]

58.

59. # ---- group DLP for 32-bit d: find d s.t. d\*Q = P ----

60. def bsgs\_small\_d(Pt, Qt, N=(1 << 32)):

61. """

62. Solve d in [0,N) for d\*Qt == Pt using baby-step/giant-step.

63. This variant avoids using the library's point-at-infinity.

64. """

65. m = int(math.isqrt(N)) + 1

66.

67. # Baby steps: table[j] = j\*Q for j = 1..m

68. table = {}

69. cur = Qt # 1\*Q

70. for j in range(1, m + 1):

71. table[(cur.x, cur.y)] = j

72. cur = cur + Qt

73.

74. # Precompute -m\*Q

75. mQ = m \* Qt

76. neg\_mQ = -mQ

77.

78. # Giant steps: R\_i = P - i\*m\*Q (i = 0..m)

79. R = Pt

80. for i in range(0, m + 1):

81. key = (R.x, R.y)

82. if key in table:

83. j = table[key] # in 1..m

84. d = i \* m + j

85. if d < N and d \* Qt == Pt:

86. return d

87. R = R + neg\_mQ

88.

89. raise ValueError("d not found in range")

90.

91. def main():

92. with open("challenge.json", "r") as f:

93. data = json.load(f)

94.

95. curve = P256

96. # Read P, Q from file (we trust they match P256.G and a multiple)

97. Px = int(data["P"]["x"], 16); Py = int(data["P"]["y"], 16)

98. Qx = int(data["Q"]["x"], 16); Qy = int(data["Q"]["y"], 16)

99. Ppub = Point(Px, Py, curve=curve)

100. Qpub = Point(Qx, Qy, curve=curve)

101.

102. # Observed 32-byte windows; each encodes out[i] (30B) || top2(out[i+1])

103. observed = [int(h, 16) for h in data["observed"]]

104. outs = [(obs >> 16) for obs in observed] # 30-byte truncated x for i=0..4

105.

106. # Recover 32-bit d from d\*Q = P

107. d = bsgs\_small\_d(Ppub, Qpub, 1 << 32)

108. # print(f"Recovered d = {d}")

109.

110. # Brute the missing 16 MSBs of x(R0) and validate against out[1]

111. p = curve.p

112. mask\_30 = MASK\_30B

113. found = None

114.

115. # We know out[0] (lower 30 bytes of x0). Try all 2^16 candidates for high 16 bits.

116. x0\_lo = outs[0] # 240-bit

117. for hi in range(1 << 16):

118. x0 = (hi << 240) | x0\_lo

119. if x0 >= p:

120. continue

121. pts = point\_from\_x(curve, x0)

122. if not pts:

123. continue

124. for R0 in pts:

125. # s1 = x(d \* R0)

126. S1 = d \* R0

127. s1 = S1.x

128. # R1 = s1 \* Q, compare truncated x with outs[1]

129. R1 = s1 \* Qpub

130. out1 = R1.x & mask\_30

131. if out1 == outs[1]:

132. found = (R0, s1)

133. break

134. if found:

135. break

136.

137. if not found:

138. raise RuntimeError("Failed to reconstruct the state from observed outputs")

139.

140. Rk, sk = found # R0 and s1

141. # Step forward to get final output (k=5)

142. # We already validated k=1. We’ll continue through k=4 and compute k=5 fresh.

143. for idx in range(1, 5):

144. # Optional: consistency check with outs[idx] (already matched for idx=1)

145. Rk = sk \* Qpub

146. assert (Rk.x & mask\_30) == outs[idx], "Consistency check failed"

147. # advance seed

148. sk = (d \* Rk).x

149.

150. # Now compute final output (k=5)

151. R5 = sk \* Qpub

152. out5 = R5.x & mask\_30

153. out5\_bytes = out5.to\_bytes(30, "big")

154. key = out5\_bytes[:16]

155.

156. # Decrypt

157. ct = base64.b64decode(data["ciphertext"])

158. iv = base64.b64decode(data["iv"])

159. cipher = AES.new(key, AES.MODE\_CBC, iv)

160. flag = unpad(cipher.decrypt(ct), AES.block\_size)

161. print(flag.decode(errors="ignore"))

162.

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

164. main()

165.

## Crypto

### Beta Bet

Solve script:

1. import sys

2. import string

3. from collections import Counter

4.

5. ALPHA = string.ascii\_lowercase

6.

7. def residue\_to\_lowercase\_letter(res\_mod\_26: int) -> str:

8. """

9. Map a residue modulo 26 to the unique lowercase 'a'..'z' letter

10. that has the same ASCII code residue modulo 26.

11. (For lowercase, this is 1-to-1 because ord('a'..'z') are 97..122.)

12. """

13. # Precompute once per run for speed/readability

14. if not hasattr(residue\_to\_lowercase\_letter, "\_map"):

15. residue\_to\_lowercase\_letter.\_map = {ord(ch) % 26: ch for ch in ALPHA}

16. return residue\_to\_lowercase\_letter.\_map[res\_mod\_26]

17.

18. def recover\_middle(cipher\_lines):

19. """

20. Each line has the form:

21. PREFIX + encrypt(MIDDLE) + SUFFIX

22. where the encrypt step is:

23. c = (ord(p) + ord(k)) % 26; then output chr(c + ord('a'))

24. and k is drawn from 'b'..'z' only (never 'a').

25.

26. For a fixed plaintext position:

27. seen ciphertext residues = { (ord(p) + ord(k)) % 26 | k in 'b'..'z' }

28. That's 25 residues; the ONLY missing residue is (ord(p) + ord('a')) % 26.

29. So:

30. missing = (ord(p) + ord('a')) % 26

31. => ord(p) % 26 = (missing - (ord('a') % 26)) % 26

32. = (missing - 19) % 26

33. """

34. # detect prefix/suffix lengths (use first and brace heuristics)

35. first = cipher\_lines[0]

36. # Heuristic for this challenge format: prefix ends at the first '{',

37. # suffix is the final '}' (kept generic in case lengths vary)

38. try:

39. lbrace = first.index('{')

40. rbrace = first.rindex('}')

41. except ValueError:

42. # Fallback: assume prefix len=6, suffix len=1 (matches the generator)

43. lbrace, rbrace = 6, len(first) - 1

44.

45. prefix = first[:lbrace]

46. suffix = first[rbrace+0:] # usually "}"

47. mids = [line[lbrace+1:rbrace] for line in cipher\_lines]

48.

49. L = len(mids[0])

50. # Sanity: all same length

51. assert all(len(m) == L for m in mids), "Inconsistent ciphertext lengths"

52.

53. # Column-wise recovery

54. pieces = []

55. options\_per\_pos = [] # keep candidates for each column (to report clearly)

56. A = ord('a')

57. A\_mod = A % 26 # == 19

58.

59. for i in range(L):

60. col = [m[i] for m in mids]

61. # ciphertext residue is simply (ord(c) - ord('a'))

62. seen\_residues = {ord(c) - A for c in col}

63. # Find which residues 0..25 never appeared at this column

64. missing\_residues = [r for r in range(26) if r not in seen\_residues]

65.

66. # Map each missing residue back to a plaintext lowercase letter

67. # ord(p) % 26 = (missing - 19) % 26

68. candidates = []

69. for mr in missing\_residues:

70. p\_res = (mr - A\_mod) % 26

71. candidates.append(residue\_to\_lowercase\_letter(p\_res))

72.

73. options\_per\_pos.append(sorted(candidates))

74.

75. if len(candidates) == 1:

76. pieces.append(candidates[0])

77. else:

78. # under-sampled -> ambiguity: show all consistent choices

79. pieces.append("[" + "".join(sorted(candidates)) + "]")

80.

81. ambiguous\_mid = "".join(pieces)

82. # A single “best guess” by picking the first option at each position

83. best\_guess\_mid = "".join(

84. opts[0] if len(opts) >= 1 else "?"

85. for opts in options\_per\_pos

86. )

87.

88. return prefix, ambiguous\_mid, best\_guess\_mid, suffix

89.

90. def main():

91. if len(sys.argv) < 2:

92. print(f"Usage: {sys.argv[0]} out.txt")

93. sys.exit(1)

94.

95. with open(sys.argv[1], "r", encoding="utf-8") as f:

96. lines = [line.rstrip("\n") for line in f if line.strip()]

97.

98. # Basic sanity

99. if not lines:

100. print("No lines found in input.")

101. sys.exit(1)

102.

103. prefix, ambiguous\_mid, best\_mid, suffix = recover\_middle(lines)

104.

105. print("[\*] Prefix:", prefix)

106. print("[\*] Suffix:", suffix)

107. print("[\*] Recovered middle (with ambiguities in brackets):")

108. print(ambiguous\_mid)

109. print()

110. print("[\*] Best-guess middle (pick first option in each bracket):")

111. print(best\_mid)

112. print()

113. # If it looks like a typical flag, print full strings too

114. print("[\*] Flag with ambiguities:")

115. print(f"{prefix}{{{ambiguous\_mid}}}{suffix if suffix != '' else ''}")

116. print("[\*] Best-guess flag:")

117. print(f"{prefix}{{{best\_mid}}}{suffix if suffix != '' else ''}")

118.

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

120. main()

121.

### Not a backdoor

Solve script:

1. import json, math, base64

2. from Crypto.Cipher import AES

3. from Crypto.Util.Padding import unpad

4. from fastecdsa.curve import P256

5. from fastecdsa.point import Point

6.

7. MASK\_30B = (1 << (8\*30)) - 1 # lower 30 bytes

8.

9. # ---- number theory helpers ----

10. def tonelli\_shanks(n, p):

11. """Return y with y^2 == n (mod p) or None if no root; works for odd prime p."""

12. if n % p == 0:

13. return 0

14. # Legendre symbol check

15. ls = pow(n, (p - 1) // 2, p)

16. if ls != 1:

17. return None

18. # p-1 = q \* 2^s with q odd

19. q = p - 1

20. s = 0

21. while q % 2 == 0:

22. q //= 2

23. s += 1

24. # find z a quadratic non-residue

25. z = 2

26. while pow(z, (p - 1) // 2, p) != p - 1:

27. z += 1

28. c = pow(z, q, p)

29. x = pow(n, (q + 1) // 2, p)

30. t = pow(n, q, p)

31. m = s

32. while t != 1:

33. # find least i (0 < i < m) with t^(2^i) == 1

34. i = 1

35. t2i = (t \* t) % p

36. while i < m and t2i != 1:

37. t2i = (t2i \* t2i) % p

38. i += 1

39. b = pow(c, 1 << (m - i - 1), p)

40. x = (x \* b) % p

41. c = (b \* b) % p

42. t = (t \* c) % p

43. m = i

44. return x

45.

46. def point\_from\_x(curve, x):

47. """Return the two curve points with this x, or [] if none."""

48. p = curve.p

49. a = curve.a

50. b = curve.b

51. rhs = (pow(x, 3, p) + (a \* x) + b) % p

52. y = tonelli\_shanks(rhs, p)

53. if y is None:

54. return []

55. if y == 0:

56. return [Point(x, 0, curve=curve)]

57. return [Point(x, y, curve=curve), Point(x, (-y) % p, curve=curve)]

58.

59. # ---- group DLP for 32-bit d: find d s.t. d\*Q = P ----

60. def bsgs\_small\_d(Pt, Qt, N=(1 << 32)):

61. """

62. Solve d in [0,N) for d\*Qt == Pt using baby-step/giant-step.

63. This variant avoids using the library's point-at-infinity.

64. """

65. m = int(math.isqrt(N)) + 1

66.

67. # Baby steps: table[j] = j\*Q for j = 1..m

68. table = {}

69. cur = Qt # 1\*Q

70. for j in range(1, m + 1):

71. table[(cur.x, cur.y)] = j

72. cur = cur + Qt

73.

74. # Precompute -m\*Q

75. mQ = m \* Qt

76. neg\_mQ = -mQ

77.

78. # Giant steps: R\_i = P - i\*m\*Q (i = 0..m)

79. R = Pt

80. for i in range(0, m + 1):

81. key = (R.x, R.y)

82. if key in table:

83. j = table[key] # in 1..m

84. d = i \* m + j

85. if d < N and d \* Qt == Pt:

86. return d

87. R = R + neg\_mQ

88.

89. raise ValueError("d not found in range")

90.

91. def main():

92. with open("challenge.json", "r") as f:

93. data = json.load(f)

94.

95. curve = P256

96. # Read P, Q from file (we trust they match P256.G and a multiple)

97. Px = int(data["P"]["x"], 16); Py = int(data["P"]["y"], 16)

98. Qx = int(data["Q"]["x"], 16); Qy = int(data["Q"]["y"], 16)

99. Ppub = Point(Px, Py, curve=curve)

100. Qpub = Point(Qx, Qy, curve=curve)

101.

102. # Observed 32-byte windows; each encodes out[i] (30B) || top2(out[i+1])

103. observed = [int(h, 16) for h in data["observed"]]

104. outs = [(obs >> 16) for obs in observed] # 30-byte truncated x for i=0..4

105.

106. # Recover 32-bit d from d\*Q = P

107. d = bsgs\_small\_d(Ppub, Qpub, 1 << 32)

108. # print(f"Recovered d = {d}")

109.

110. # Brute the missing 16 MSBs of x(R0) and validate against out[1]

111. p = curve.p

112. mask\_30 = MASK\_30B

113. found = None

114.

115. # We know out[0] (lower 30 bytes of x0). Try all 2^16 candidates for high 16 bits.

116. x0\_lo = outs[0] # 240-bit

117. for hi in range(1 << 16):

118. x0 = (hi << 240) | x0\_lo

119. if x0 >= p:

120. continue

121. pts = point\_from\_x(curve, x0)

122. if not pts:

123. continue

124. for R0 in pts:

125. # s1 = x(d \* R0)

126. S1 = d \* R0

127. s1 = S1.x

128. # R1 = s1 \* Q, compare truncated x with outs[1]

129. R1 = s1 \* Qpub

130. out1 = R1.x & mask\_30

131. if out1 == outs[1]:

132. found = (R0, s1)

133. break

134. if found:

135. break

136.

137. if not found:

138. raise RuntimeError("Failed to reconstruct the state from observed outputs")

139.

140. Rk, sk = found # R0 and s1

141. # Step forward to get final output (k=5)

142. # We already validated k=1. We’ll continue through k=4 and compute k=5 fresh.

143. for idx in range(1, 5):

144. # Optional: consistency check with outs[idx] (already matched for idx=1)

145. Rk = sk \* Qpub

146. assert (Rk.x & mask\_30) == outs[idx], "Consistency check failed"

147. # advance seed

148. sk = (d \* Rk).x

149.

150. # Now compute final output (k=5)

151. R5 = sk \* Qpub

152. out5 = R5.x & mask\_30

153. out5\_bytes = out5.to\_bytes(30, "big")

154. key = out5\_bytes[:16]

155.

156. # Decrypt

157. ct = base64.b64decode(data["ciphertext"])

158. iv = base64.b64decode(data["iv"])

159. cipher = AES.new(key, AES.MODE\_CBC, iv)

160. flag = unpad(cipher.decrypt(ct), AES.block\_size)

161. print(flag.decode(errors="ignore"))

162.

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

164. main()

165.

## Web

### The Agreement

A black screen with white text

AI-generated content may be incorrect.

### Worse than SQLi

A screenshot of a computer

AI-generated content may be incorrect.

### Cryptic Mistake

A screenshot of a computer error

AI-generated content may be incorrect.