

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



Advanced Cryptography and Coding Theory (CO3083)

Course assignment

Analysis & Exploitation - Group 5

Lecturer:	Nguyễn An Khương	
Advisor:	Trịnh Cao Thắng	
	Đặng Dương Minh Nhật	
Student:	Lê Đức Tâm	- 2213016
	Trần Vĩ Quang	- 2212760
	Vũ Trần Duy Nguyên	- 2212338
	Phạm Gia Trí	- 2213656
	Nguyễn Hữu Nhân	- 2212362

HO CHI MINH CITY, FALL 2025



Contents

1	Member list and workload	2
2	Research and Development	2
2.1	Problem 1: Lattice handling	2
2.1.1	Problem Statement	2
2.1.2	Part (a): Terms Report	2
2.1.3	Part (b): Finding Minimal Polynomial Using Lattices	4
2.1.4	Part (c): Recovering Integer X from Approximate \sqrt{X}	6
2.2	Problem 2: Attacking the Many-Time Pad	7
2.2.1	Problem Statement	7
2.2.2	Step 1: Reconnaissance (Initial Observation)	7
2.2.3	Step 2: Hypothesis & Testing	9
2.2.4	Step 3: Writing the Decryption Script (Exploitation)	13
2.2.5	Step 4: Result Analysis (Post-Exploitation)	14
2.2.6	Step 5: Conclusion (Summary)	15
2.2.7	Lessons from This Attack	15
2.2.8	The Art of Guessing	16
2.2.9	Conclusion	16
3	SMC Exploitation	17
3.1	Interception and API Analysis	17
3.1.1	Step 1: Initial Key Exchange / Handshake	17
3.1.2	Step 2: Session Establishment	18
3.1.3	Step 3: Subsequent Encrypted Messaging	19
3.2	Re-implementation and Protocol Reconstruction	20
3.2.1	Phase 1: Initial Handshake (Parameter Negotiation)	20
3.2.2	Phase 2: Session Establishment (Key Agreement)	20
3.2.3	Phase 3: Secure Messaging	21
4	Exploitation & Proof-of-Concept	21
4.1	Step 1: Finding Weak Curve Parameters	21
4.2	Step 2: Sending Malicious Request & Extracting Public Key	22
4.3	Step 3: Solving the Discrete Logarithm Problem	22
4.4	Step 4: Restoring the Original Private Key	23



1 Member list and workload

No.	Fullname	Student ID	Problems	Task Completion
1	Phạm Gia Trí	2213656	- Task assignment & progress management - SMC exploitation implementation - Technical coordination	100%
2	Trần Vĩ Quang	2212760	- Problem 2: Many-Time Pad attack - Cryptanalysis implementation - Result documentation	100%
3	Vũ Trần Duy Nguyên	2212338	- Problem 1: Lattice handling - Algorithm design and analysis - Experimental evaluation	100%
4	Lê Đức Tâm	2213016	- Result verification - Report integration and writing - GitHub repository management	100%
5	Nguyễn Hữu Nhân	2212362	- SMC exploitation analysis - Exploit development - Technical write-up	100%

2 Research and Development

2.1 Problem 1: Lattice handling

2.1.1 Problem Statement

Given Definitions:

Definition 2.7 (Subfields and Extension Fields): A subfield K of field L is a subset $K \subseteq L$ that is a field. If K is a subfield of field L , then L is the extension field of field K , denoted as L/K (read " L over K ").

Definition 2.8 (Minimal Polynomial): Let $\alpha \in L$, where F is a field and L/F is a field extension. A minimal polynomial of α is a monic polynomial of lowest degree in $F[x]$ where α is a root.

Tasks:

- (1 point) Read Chapter 6 in [HPS08] and present the following terms: vector spaces, linear combinations, independence, bases, orthogonal and orthonormal basis, lattices, fundamental domains, the Shortest Vector Problem (SVP), the Closest Vector Problem (CVP), the Euclidean ball, the Gaussian expected shortest length, and the LLL algorithm.
- (1 point) Given $\alpha = 3 + \sqrt[3]{23}$, with an approximation β of α to 10 decimal places, find a minimal polynomial $f(x)$ of α by reformulating this into a lattice problem.
- (1 point) Suppose you know the first d digits after the decimal place of \sqrt{X} . Show that you can find X by reformulating this problem into a lattice problem.

2.1.2 Part (a): Terms Report

Vector Spaces

A *vector space* $V \subseteq \mathbb{R}^m$ is a set that is closed under vector addition and scalar multiplication. Formally, V is a vector space if for all $\mathbf{v}_1, \mathbf{v}_2 \in V$ and all $\alpha_1, \alpha_2 \in \mathbb{R}$,

$$\alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 \in V.$$

Linear Combinations

Given vectors $\mathbf{v}_1, \dots, \mathbf{v}_k \in V$, any vector of the form

$$\mathbf{w} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \dots + \alpha_k \mathbf{v}_k, \quad \alpha_1, \dots, \alpha_k \in \mathbb{R},$$

is called a *linear combination* of $\mathbf{v}_1, \dots, \mathbf{v}_k$.

Independence

A set of vectors $\mathbf{v}_1, \dots, \mathbf{v}_k \in V$ is (*linearly*) *independent* if

$$\alpha_1 \mathbf{v}_1 + \dots + \alpha_k \mathbf{v}_k = \mathbf{0} \implies \alpha_1 = \dots = \alpha_k = 0.$$

Bases

A basis of a vector space V is a linearly independent set of vectors that spans V . For a lattice L , a *basis* is any set of independent vectors that generates L through integer combinations.

Orthogonal and Orthonormal Bases

A basis $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ is *orthogonal* if

$$\mathbf{v}_i \cdot \mathbf{v}_j = 0 \quad \text{for all } i \neq j.$$

It is *orthonormal* if furthermore $\|\mathbf{v}_i\| = 1$ for all i .

Lattices

Let $\mathbf{v}_1, \dots, \mathbf{v}_n \in \mathbb{R}^m$ be linearly independent. The lattice generated by these vectors is

$$L = \{a_1 \mathbf{v}_1 + \dots + a_n \mathbf{v}_n : a_1, \dots, a_n \in \mathbb{Z}\}.$$

Fundamental Domains

Let L be an n -dimensional lattice with basis $\mathbf{v}_1, \dots, \mathbf{v}_n$. The associated *fundamental domain* (fundamental parallelepiped) is

$$F(\mathbf{v}_1, \dots, \mathbf{v}_n) = \{t_1 \mathbf{v}_1 + \dots + t_n \mathbf{v}_n : 0 \leq t_i < 1\}.$$

The Shortest Vector Problem (SVP)

Given a lattice L , the *Shortest Vector Problem* is to find a nonzero vector

$$\mathbf{v} \in L \setminus \{\mathbf{0}\}$$

that minimizes the norm $\|\mathbf{v}\|$.

The Closest Vector Problem (CVP)

Given $\mathbf{w} \in \mathbb{R}^m$, the *Closest Vector Problem* asks for a vector $\mathbf{v} \in L$ minimizing

$$\|\mathbf{w} - \mathbf{v}\|.$$

Euclidean Ball

For $\mathbf{a} \in \mathbb{R}^n$ and $R > 0$, the closed Euclidean ball of radius R centered at \mathbf{a} is

$$B_R(\mathbf{a}) = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{a}\| \leq R\}.$$

Gaussian Expected Shortest Length

For an n -dimensional lattice L , the *Gaussian expected shortest length* is

$$\sigma(L) = \sqrt{\frac{n}{2\pi e}} (\det L)^{1/n}.$$

The Gaussian heuristic predicts that a shortest nonzero vector satisfies

$$\|\mathbf{v}_{\text{shortest}}\| \approx \sigma(L).$$

The LLL Algorithm

The *LLL algorithm* is a polynomial-time lattice reduction method that transforms a basis of a lattice L into an *LLL-reduced basis*. A basis $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ is LLL-reduced if:

1. Size Condition:

$$|\mu_{i,j}| = \left| \frac{\mathbf{v}_i \cdot \mathbf{v}_j^*}{\|\mathbf{v}_j^*\|^2} \right| \leq \frac{1}{2} \quad \text{for all } 1 \leq j < i \leq n,$$

where \mathbf{v}_j^* are the Gram-Schmidt vectors.

2. Lovász Condition:

$$\|\mathbf{v}_i^*\|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2 \right) \|\mathbf{v}_{i-1}^*\|^2 \quad \text{for all } 1 < i \leq n.$$

The LLL algorithm provides a $2^{(n-1)/2}$ -approximation to SVP. Combined with Babai's nearest-plane method, it yields an algorithm approximating CVP within a factor of C^n for some constant C .

2.1.3 Part (b): Finding Minimal Polynomial Using Lattices

Strategy: Integer Relation Finding

We seek a minimal polynomial of degree $n = 3$ (since α involves a cube root). We are looking for integer coefficients vector $\mathbf{c} = (c_0, c_1, c_2, c_3)$ such that:

$$c_3\alpha^3 + c_2\alpha^2 + c_1\alpha + c_0 = 0$$

Given the approximation $\beta \approx \alpha$, we construct a lattice problem to find small integers c_i such that:

$$c_3\beta^3 + c_2\beta^2 + c_1\beta + c_0 \approx 0$$

Numerical Setup

- **Approximation (β):** $\sqrt[3]{23} \approx 2.8438669798$, so $\beta \approx 5.8438669800$.
- **Powers of β :**

$$\begin{aligned} \beta^0 &= 1 \\ \beta^1 &\approx 5.8438669800 \\ \beta^2 &\approx 34.1507831018 \\ \beta^3 &\approx 199.5714285743 \end{aligned}$$

- **Scaling Constant (C):** To define the lattice over \mathbb{Z} , we scale by a large constant $C = 10^{11}$ (matching the precision of 10^{-10}).

Lattice Formulation

We construct a lattice \mathcal{L} generated by the rows of the basis matrix B . The basis vectors are embedded in \mathbb{Z}^5 :

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & \lfloor C\beta^0 \rfloor \\ 0 & 1 & 0 & 0 & \lfloor C\beta^1 \rfloor \\ 0 & 0 & 1 & 0 & \lfloor C\beta^2 \rfloor \\ 0 & 0 & 0 & 1 & \lfloor C\beta^3 \rfloor \end{pmatrix}$$

Substituting the values:

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 100,000,000,000 \\ 0 & 1 & 0 & 0 & 584,386,698,000 \\ 0 & 0 & 1 & 0 & 3,415,078,310,180 \\ 0 & 0 & 0 & 1 & 19,957,142,857,430 \end{pmatrix}$$

A vector v in this lattice is a linear combination of the rows with coefficients c_0, c_1, c_2, c_3 :

$$v = (c_0, c_1, c_2, c_3, \lambda)$$

where $\lambda \approx C \cdot f(\beta)$. We seek a "short vector" in this lattice using the LLL algorithm. A short vector implies small coefficients c_i and a very small λ (meaning $f(\beta) \approx 0$).

Resulting Vector and Polynomial

Applying lattice reduction (LLL) yields the short vector v^* :

$$v^* = (-50, 27, -9, 1, \epsilon)$$

The first four components correspond to the coefficients c_0, c_1, c_2, c_3 :

$$c_0 = -50, \quad c_1 = 27, \quad c_2 = -9, \quad c_3 = 1$$

Thus, the candidate polynomial is:

$$f(x) = x^3 - 9x^2 + 27x - 50$$

Verification

We verify this result algebraically using $\alpha = 3 + \sqrt[3]{23}$. Let $x = \alpha$.

$$x = 3 + \sqrt[3]{23}$$

$$x - 3 = \sqrt[3]{23}$$

$$(x - 3)^3 = 23$$

$$x^3 - 3(x^2)(3) + 3(x)(3^2) - 3^3 = 23$$

$$x^3 - 9x^2 + 27x - 27 = 23$$

$$x^3 - 9x^2 + 27x - 50 = 0$$

The minimal polynomial is $f(x) = x^3 - 9x^2 + 27x - 50$.

2.1.4 Part (c): Recovering Integer X from Approximate \sqrt{X}

Algebraic Formulation

Suppose we want to recover an integer X . Let \sqrt{X} be the real square root of X . Assume we are given its decimal expansion, and denote the fractional part by β . Let α be the integer part of \sqrt{X} (which can be estimated or enumerated).

Thus, we have the relation:

$$\sqrt{X} = \alpha + \beta.$$

Squaring both sides gives:

$$X = (\alpha + \beta)^2 = \alpha^2 + 2\alpha\beta + \beta^2.$$

Rearranging, we obtain a polynomial in β :

$$1 \cdot \beta^2 + (2\alpha) \cdot \beta + (\alpha^2 - X) = 0.$$

Let the unknown integer coefficients be:

$$\begin{cases} a_2 = 1, \\ a_1 = 2\alpha, \\ a_0 = \alpha^2 - X. \end{cases}$$

Our objective is to find the integer triple (a_2, a_1, a_0) satisfying:

$$a_2\beta^2 + a_1\beta + a_0 \approx 0.$$

Lattice Reformulation

To find small integers (a_2, a_1, a_0) such that the above expression is close to 0, we construct a lattice \mathcal{L} generated by a basis matrix M .

Choose a sufficiently large scaling constant K (corresponding to the precision of β , e.g., $K = 10^{10}$). Define the basis vectors of the lattice as follows:

- Vector for the constant term:

$$v_0 = (1, 0, 0, K)$$

- Vector for the β term:

$$v_1 = (0, 1, 0, K\beta)$$

- Vector for the β^2 term:

$$v_2 = (0, 0, 1, K\beta^2)$$

The basis matrix M (whose rows correspond to these vectors) is:

$$M = \begin{pmatrix} 1 & 0 & 0 & K \\ 0 & 1 & 0 & K\beta \\ 0 & 0 & 1 & K\beta^2 \end{pmatrix}.$$

Any vector in the lattice \mathcal{L} has the form:

$$v = a_0v_0 + a_1v_1 + a_2v_2 = (a_0, a_1, a_2, K(a_0 + a_1\beta + a_2\beta^2)).$$

To make the last component

$$K(a_0 + a_1\beta + a_2\beta^2)$$

as small as possible (i.e., close to 0), the LLL algorithm will identify the shortest vector in this lattice.

Algorithm Application and Recovery of X

Apply the LLL algorithm to the basis matrix M to obtain a reduced basis. Let the first (shortest) vector of the reduced basis be:

$$u = (T_1, T_2, T_3, \varepsilon).$$

Given the structure of the lattice, we can match components of u to the polynomial coefficients:

$$\begin{cases} a_0 = T_1, \\ a_1 = T_2, \\ a_2 = T_3. \end{cases}$$

From the algebraic formulation, we can now reconstruct:

1. Check a_2 . If $a_2 = 1$ (or -1), the solution is valid.
2. Compute the integer part α :

$$\alpha = \frac{a_1}{2}.$$

3. Recover the desired integer X :

$$\alpha^2 - X = a_0 \quad \Rightarrow \quad X = \alpha^2 - a_0.$$

Conclusion

We have successfully recovered the integer X from the decimal fraction β of \sqrt{X} by reducing the problem to finding a shortest vector (SVP) in a lattice using the LLL algorithm.

2.2 Problem 2: Attacking the Many-Time Pad

2.2.1 Problem Statement

It is understood that most classical ciphers are now broken, including the Many-Time Pad. In this problem, we need to write an attack script (in Python or any other programming language) to attack a bunch of ciphertexts encrypted using the same key in One-Time Pad (hence, the name Many-Time Pad) and explain the core principles behind the attack.

We are given a collection of hexadecimal-encoded ciphertexts. Our task is to recover either the secret key or the plaintexts.

2.2.2 Step 1: Reconnaissance (Initial Observation)

When I received this pile of hexadecimal data, my first instinct was to scan through it looking for patterns.

Code Snippet 1.1: Loading and examining the data

```
1 # Read the ciphertext file
2 with open('ciphertexts2.txt', 'r') as f:
3     data = f.read()
4
5 # Split into lines
6 lines = data.strip().split('\n')
7 print(f"Total lines: {len(lines)}")
```

```
8 print("\nFirst 10 lines (showing first 50 chars each):")
9 for i in range(10):
10     print(f"Line {i+1:2d}: {lines[i][:50]}...")
```

Total lines: 36

First 10 lines (showing first 50 chars each):

```
Line 1: 1604104516091408623810043945081e621e1016240a0f1e2b...
Line 2: 000000007431040c2f1f55083b0b08192d1e550b3111160230...
Line 3: 1604104520001300624b370921004139270d1842740a130425...
Line 4: 000000007431040c2f1f550a32110403621b1a173f450e1d32...
Line 4: 000000007431040c2f1f550a32110403621b1a173f450e1d32...
Line 4: 000000007431040c2f1f550a32110403621b1a173f450e1d32...
Line 5: 070a13003711081b274c370921004139270d180c3a02411f27...
Line 6: 11091610260c151462251b033b170c0c36051a0b74040f0962...
Line 7: 000000007431040c2f1f55043a040d14380955093b02124d24...
Line 8: 16040700351141053702010c3a024104314c144524170e0c21...
Line 9: 000000007431040c2f4c18003907041f314c1a0320000f4d37...
Line 10: 0b02160c30000f19621e1016240a0f1e274c1c0b220a0d1b27...
```

Figure 1: Total lines count and print first 10 lines

Code Snippet 1.2: Looking for repeating patterns

```
1 # Count lines starting with same prefix
2 from collections import Counter
3
4 # Check first 8 hex chars (4 bytes)
5 prefixes = [line[:8] for line in lines]
6 prefix_counts = Counter(prefixes)
7
8 print("Most common prefixes:")
9 for prefix, count in prefix_counts.most_common(5):
10     print(f" {prefix}: appears {count} times")
11
12 # Find lines starting with 00000000
13 null_lines = [i+1 for i, line in enumerate(lines) if line.startswith('00000000')]
14 print(f"\nLines starting with 00000000: {null_lines}")
15 print(f"Total: {len(null_lines)} lines")
```

```
Most common prefixes:
00000000: appears 12 times
16041045: appears 6 times
1604101c: appears 5 times
070a1300: appears 2 times
11091610: appears 2 times

Lines starting with 00000000: [2, 4, 7, 9, 16, 18, 19, 23, 27, 28, 30, 32]
Total: 12 lines
PS D:\learn\z_251\ACCT\asmt\cau2>
```

Figure 2: pattern checking

I discovered 2 anomalies:

Anomaly 1: Repetition at the beginning of lines

Many lines start with the sequence 16041045. This suggests that these sentences begin with the same word.

Anomaly 2: Lines with 00000000

In XOR encryption, the number 0 (null byte) is the "Achilles' heel."

XOR formula: $A \oplus A = 0$

This means:

- If Ciphertext = 0x00 → Plaintext and Key are identical at that position
- Or: If Plaintext = 0x00 → Ciphertext is the Key itself

2.2.3 Step 2: Hypothesis & Testing

I have two attack vectors:

Attack Vector 1: Attacking the repeating pattern 16041045 (Known-Plaintext Attack)

This is English text. What word appears most frequently at the beginning of a sentence?

Definitely "The " (4 characters: T, h, e, and space).

Code Snippet 2.1: First attempt - testing "The "

```
1 # Test if 16041045 decrypts to "The "  
2 cipher_hex = "16041045"  
3 guess = "The "  
4  
5 # Convert hex to bytes  
6 cipher_bytes = bytes.fromhex(cipher_hex)  
7  
8 # XOR to find key  
9 key_bytes = []  
10 for i in range(4):  
11     key_byte = cipher_bytes[i] ^ ord(guess[i])  
12     key_bytes.append(key_byte)  
13     print(f"Byte {i+1}: 0x{cipher_bytes[i]:02x} XOR '{guess[i]}' (0x{ord(guess[i]):02x}) = 0x{key_byte:02x} ('{chr(key_byte)}')")  
14  
15 key_fragment = ''.join(chr(b) for b in key_bytes)  
16 print(f"\nKey fragment found: '{key_fragment}'")
```

```
Byte 1: 0x16 XOR 'T' (0x54) = 0x42 ('B')  
Byte 2: 0x04 XOR 'h' (0x68) = 0x6c ('l')  
Byte 3: 0x10 XOR 'e' (0x65) = 0x75 ('u')  
Byte 4: 0x45 XOR ' ' (0x20) = 0x65 ('e')  
  
Key fragment found: 'Blue'
```

Figure 3: Known-Plaintext Attack - Testing "The "

I performed some quick calculations (crib dragging):

→ I pieced together the first fragment of the Key: "Blue".

Cipher byte	Guess Plaintext	Key calculation
Byte 1: 0x16	'T' (0x54)	$0x16 \oplus 0x54 = 0x42$ ('B')
Byte 2: 0x04	'h' (0x68)	$0x04 \oplus 0x68 = 0x6C$ ('I')
Byte 3: 0x10	'e' (0x65)	$0x10 \oplus 0x65 = 0x75$ ('u')
Byte 4: 0x45	' ' (0x20)	$0x45 \oplus 0x20 = 0x65$ ('e')

Table 1: Key recovery through known-plaintext attack

This sounds very reasonable (Blue as in Blue Team?). I continued testing with the next 4 bytes of the first line (16091408).

Code Snippet 2.2: Testing different guesses for next word

```

1 # We have "Blue" - what comes next?
2 # Let's test a few possibilities
3
4 known_key = "Blue"
5 next_cipher_hex = "16091408"
6 next_cipher_bytes = bytes.fromhex(next_cipher_hex)
7
8 guesses = ["Team", "Flag", "Code", "Secu"]
9
10 print("Testing different guesses for next 4 bytes:")
11 for guess in guesses:
12     key_bytes = []
13     for i in range(4):
14         key_byte = next_cipher_bytes[i] ^ ord(guess[i])
15         key_bytes.append(key_byte)
16
17     key_fragment = ''.join(chr(b) for b in key_bytes)
18
19     # Check if it matches our known key pattern
20     full_key = known_key + key_fragment
21     print(f"\nGuess: '{guess}' -> Key becomes: '{full_key}'")
22
23     # Check if first 4 bytes match
24     matches = all(key_bytes[i] == ord(known_key[i]) for i in range(4))
25     if matches:
26         print(f"    -> MATCH! This continues our key pattern!")
27     else:
28         print(f"    -> No match. Expected 'Blue' pattern to repeat.")

```

```
Testing different guesses for next 4 bytes:

Guess: 'Team' -> Key becomes: 'BlueBlue'
-> MATCH! This continues our key pattern!

Guess: 'Flag' -> Key becomes: 'BluePeuo'
-> No match. Expected 'Blue' pattern to repeat.

Guess: 'Code' -> Key becomes: 'BlueUfpm'
-> No match. Expected 'Blue' pattern to repeat.

Guess: 'Secu' -> Key becomes: 'BlueElw}'
-> No match. Expected 'Blue' pattern to repeat.
```

Figure 4: Hypothesizing the Next Four Key Bytes

Code Snippet 2.3: Verifying "Team" guess

```
1 # Let's verify if "BlueTeam" is the correct key
2 # by decrypting the first line
3
4 full_key = "BlueTeam"
5 first_line_hex = lines[0]
6
7 # Decrypt first 50 characters
8 cipher_bytes = bytes.fromhex(first_line_hex[:100]) # First 50 bytes
9 key_bytes = full_key.encode('utf-8')
10
11 decrypted = []
12 for i in range(len(cipher_bytes)):
13     decrypted_byte = cipher_bytes[i] ^ key_bytes[i % len(key_bytes)]
14     decrypted.append(decrypted_byte)
15
16 plaintext = bytes(decrypted).decode('utf-8', errors='ignore')
17 print(f"Key: '{full_key}'")
18 print(f"First 50 chars decrypted: '{plaintext}'")
19 print("\nDoes this look like English? If yes, we found the key!")
```

```
Key: 'BlueTeam'
First 50 chars decrypted: 'The Blue Team is responsible for defending an orga'
Does this look like English? If yes, we found the key!
```

Figure 5: Key Verification - Decrypting First 50 Characters

Assuming the next word is related to "Team":

```
1 Cipher 0x16 ^ Key 'T' = 'T' (Makes sense!)
2 Cipher 0x09 ^ Key 'e' = 'e' (Makes sense!)
3 Cipher 0x14 ^ Key 'a' = 'a' (Makes sense!)
4 Cipher 0x08 ^ Key 'm' = 'm' (Makes sense!)
```

→ Key found: BlueTeam

Attack Vector 2: Attacking lines with 00000000

I looked at line 2: 000000007431...

If Cipher byte = 0x00, then Plaintext byte == Key byte.

Code Snippet 2.4: Analyzing null byte lines

```
1 # Find a line starting with 00000000
2 null_line = lines[1] # Line 2
3 print(f"Line with null bytes: {null_line[:40]}...")
4
5 # Extract the null bytes and following bytes
6 null_count = 0
7 for i in range(0, len(null_line), 2):
8     if null_line[i:i+2] == "00":
9         null_count += 1
10    else:
11        break
12
13 print(f"\nFirst {null_count} bytes are 0x00")
14 print("This means: Plaintext = Key at these positions")
15 print(f"Or: The plaintext starts with {null_count} null bytes")
16
17 # Look at bytes after the nulls
18 after_nulls_hex = null_line[null_count*2:null_count*2+16]
19 after_nulls_bytes = bytes.fromhex(after_nulls_hex)
20 print(f"\nBytes after nulls: {after_nulls_hex}")
21 print(f"As integers: {[f'0x{b:02x}' for b in after_nulls_bytes]}")
```

```
Line with null bytes: 000000007431040c2f1f55083b0b08192d1e550b...

First 4 bytes are 0x00
This means: Plaintext = Key at these positions
Or: The plaintext starts with 4 null bytes

Bytes after nulls: 7431040c2f1f5508
As integers: ['0x74', '0x31', '0x04', '0x0c', '0x2f', '0x1f', '0x55', '0x08']
```

Figure 6: Structural Analysis of the Null Byte Line

Code Snippet 2.5: Cross-verifying with our found key

```
1 # Use our found key "BlueTeam" to decrypt the null byte line
2 key = "BlueTeam"
3 null_line_hex = lines[1]
4
5 cipher_bytes = bytes.fromhex(null_line_hex)
6 key_bytes = key.encode('utf-8')
7
8 # Decrypt first 30 bytes
9 decrypted = []
10 for i in range(min(30, len(cipher_bytes))):
11     decrypted_byte = cipher_bytes[i] ^ key_bytes[i % len(key_bytes)]
12     decrypted.append(decrypted_byte)
13
14 plaintext = bytes(decrypted).decode('utf-8', errors='ignore')
15 print(f"Key: '{key}'")
16 print(f"Decrypted (first 30 chars): '{plaintext}'")
17
18 # Show byte-by-byte for first 8 bytes
```

```

19 print("\nByte-by-byte decryption:")
20 for i in range(8):
21     c = cipher_bytes[i]
22     k = key_bytes[i % len(key_bytes)]
23     p = c ^ k
24     print(f"Position {i}: 0x{c:02x} XOR 0x{k:02x} ('{chr(k)}') = 0x{p:02x} ('{chr(p)}' if 32 <= p < 127 else '?')")

```

```

Key: 'BlueTeam'
Decrypted (first 30 chars): 'Blue Teams monitor networks fo'

Byte-by-byte decryption:
Position 0: 0x00 XOR 0x42 ('B') = 0x42 ('B')
Position 1: 0x00 XOR 0x6c ('l') = 0x6c ('l')
Position 2: 0x00 XOR 0x75 ('u') = 0x75 ('u')
Position 3: 0x00 XOR 0x65 ('e') = 0x65 ('e')
Position 4: 0x74 XOR 0x54 ('T') = 0x20 (' ')
Position 5: 0x31 XOR 0x65 ('e') = 0x54 ('T')
Position 6: 0x04 XOR 0x61 ('a') = 0x65 ('e')
Position 7: 0x0c XOR 0x6d ('m') = 0x61 ('a')

```

Figure 7: Cross-Verification using BlueTeam on the Null Line

This means the first 4 characters of this line are the first 4 characters of the Key.

However, if I don't know the Key yet, I would look at the bytes following the zeros: 74 31 04 0c ...

At this point, I tried using the Key "BlueTeam" that I guessed from Attack Vector 1 to decode this section:

$$0x74 \oplus 'T' \text{ (5th character of key)} = \dots \text{yields meaningful result} \quad (1)$$

Combining both attack vectors, I'm 99.9% certain the Key is BlueTeam.

2.2.4 Step 3: Writing the Decryption Script (Exploitation)

After obtaining the Key, I didn't continue calculating by hand but wrote a small Python script to do this manual work.

Script logic:

1. Iterate through each Hex line
2. Convert Hex to Bytes
3. Loop through each Byte, XOR it with the corresponding character in Key BlueTeam (using % operator to repeat the Key)
4. Print the result to screen

```

1 data = ""
2 ...
3 ""
4

```

```
5 key = "BlueTeam"
6
7 def xor_decrypt(hex_string, key):
8     try:
9         ciphertext = bytes.fromhex(hex_string)
10        key_bytes = key.encode('utf-8')
11        decrypted = []
12        for i in range(len(ciphertext)):
13            decrypted.append(ciphertext[i] ^ key_bytes[i % len(key_bytes)])
14        return bytes(decrypted).decode('utf-8', errors='ignore')
15    except Exception as e:
16        return f"[Error decoding line: {e}]"
17
18 # Try each row in print out
19 print("--- FULL DECRYPTED TEXT ---")
20 for line in data.strip().split('\n'):
21     print(xor_decrypt(line, key))
```

```
--- FULL DECRYPTED TEXT ---
The Blue Team is responsible for defending an organization's information systems against cyber attacks.
Blue Teams monitor networks for intrusions, anomalies, and suspicious behavior.
The term 'Blue Team' originates from military wargaming exercises.
Blue Teams often work opposite Red Teams, which simulate attacks to test defenses.
Effective Blue Teaming relies on continuous monitoring and incident response.
Security Information and Event Management (SIEM) systems are essential Blue Team tools.
Blue Teams analyze logs from firewalls, servers, and endpoints to detect threats.
Threat hunting is a proactive Blue Team practice that seeks hidden attackers.
Blue Team members often use frameworks like MITRE ATT&CK to categorize adversary tactics.
Incident response involves preparation, detection, containment, eradication, and recovery.
The Blue Team's primary goal is to minimize dwell time – the period attackers remain undetected.
Digital forensics is a critical Blue Team skill for investigating security breaches.
Network segmentation helps the Blue Team limit the spread of intrusions.
Endpoint Detection and Response (EDR) tools allow real-time monitoring of workstation activity.
The Blue Team enforces security policies, access controls, and least privilege principles.
Blue Teams use intrusion detection and prevention systems (IDS/IPS) to identify malicious traffic.
A Blue Team exercise may be called a defensive cyber operation or a live-fire drill.
Blue Teams often collaborate with Purple Teams to improve both attack and defense strategies.
Blue Team analysts use packet capture tools like Wireshark for traffic inspection.
They maintain baselines of normal network behavior to detect anomalies.
Security awareness training is part of the Blue Team's human defense layer.
They perform vulnerability management by regularly scanning and patching systems.
Blue Teams document incidents thoroughly to improve future defense posture.
They often use honeypots and deception technologies to lure and analyze attackers.
The SOC (Security Operations Center) is the Blue Team's main command center.
Automation and SOAR (Security Orchestration, Automation, and Response) tools help Blue Teams scale.
Blue Teams often rely on threat intelligence feeds to stay ahead of new attack trends.
Blue Teamers must understand common attack vectors like phishing, ransomware, and privilege escalation.
Effective communication between Blue and Red Teams is vital for organizational resilience.
Blue Teams monitor both internal and external attack surfaces.
Behavioral analytics and AI are increasingly used in Blue Team detection tools.
Blue Teams play a major role in compliance and audit readiness.
They ensure backups and disaster recovery plans are tested regularly.
A good Blue Team measures success not just by prevention, but by detection and response speed.
They often participate in cyber defense competitions like the Collegiate Cyber Defense Competition (CCDC).
The best Blue Teams cultivate a continuous learning culture to keep up with evolving threats.
```

Figure 8: Final result

2.2.5 Step 4: Result Analysis (Post-Exploitation)

After running the script, I skimmed through the Plaintext content.

The content talks about the definitions of Blue Team and Red Team in security.

2.2.6 Step 5: Conclusion (Summary)

Encryption type: Vigenère (or more precisely, Repeating Key XOR).

Weaknesses:

- Short and repeating key
- Information leaked through null bytes (0x00) (when plaintext matches key)
- Common words can be guessed (Known-plaintext) at the beginning of sentences ("The ")

Key: BlueTeam

2.2.7 Lessons from This Attack

What I learned about cryptanalysis:

1. **Pattern recognition is your first tool.** Before doing any math, just look at the data. Repeated patterns, null bytes, unusual structures—these all tell you something.
2. **Context matters.** Knowing the plaintext is likely English text shaped my entire approach. I immediately thought of common words like "The", "and", "is", etc.
3. **Start small, verify often.** I didn't try to guess the entire key at once. I guessed 4 bytes, verified it made sense, then extended. Each step confirmed the previous one.
4. **Null bytes are gifts.** In XOR encryption, seeing 0x00 in ciphertext is like the system handing you a piece of the key for free.
5. **Trust your intuition, but verify mathematically.** When "Blue" appeared, it seemed too good to be true, but the math confirmed it. When "BlueTeam" made thematic sense (security context), I tested it rigorously.

Why attackers love Many-Time Pad:

This isn't a weakness in XOR itself—XOR is perfectly fine for encryption. The problem is *how it was used*:

- **Short key (8 bytes)** means high repetition
- **Same key for multiple messages** creates statistical patterns
- **Null bytes in ciphertext** leak key information directly

In real-world cryptography, we avoid these issues by:

- Using proper key derivation functions
- Never reusing keys (or using nonces/IVs)
- Using authenticated encryption (e.g., AES-GCM)
- Making keys at least 128 bits and truly random

2.2.8 The Art of Guessing

What looks like "guessing" is actually informed hypothesis testing:

Step 1: Gather context clues

- Challenge type: CTF, security course → likely security-themed text
- Language: Probably English (most common in CTF)
- Format: Clean hex strings → likely text, not binary data

Step 2: Use frequency analysis intuition

- English sentences start with: The, A, In, It, This, There...
- "The " (with space) is far more common than others
- If wrong, "and ", "is ", "be " are next attempts

Step 3: Look for confirmation

- Does partial key recovery yield readable text? ("Blue" → yes)
- Does extending the guess stay coherent? ("BlueTeam" → yes)
- Do null bytes align with our key? (yes)

This isn't random guessing—it's systematic hypothesis testing guided by experience.

2.2.9 Conclusion

Attack Summary:

- **Vulnerability:** Many-Time Pad with short, repeating key
- **Attack method:** Known-plaintext attack ("The ") + null byte analysis
- **Key recovered:** BlueTeam
- **Result:** All plaintexts decrypted successfully

This problem reinforced that cryptographic security isn't just about complex algorithms—it's about proper implementation. Even theoretically unbreakable systems (like One-Time Pad) become trivially breakable when basic rules are violated.

The most valuable skill I gained wasn't memorizing formulas, but developing intuition: knowing where to look, what to try first, and how to verify each step. That's the real art of cryptanalysis.

3 SMC Exploitation

3.1 Interception and API Analysis

To analyze the cryptographic protocol implemented by the application, a Man-in-the-Middle (MitM) proxy was deployed to intercept HTTP/2 traffic between the client (Android/Java okhttp client) and the server (`crypto-assignment...workers.dev`). By installing a custom Certificate Authority (CA) on the client device, we successfully decrypted the TLS traffic.

The analysis of the captured traffic reveals a three-step security protocol designed to establish a shared secret key and exchange encrypted messages. The process is documented below.

3.1.1 Step 1: Initial Key Exchange / Handshake

The communication initiates with a handshake request where the client proposes the cryptographic parameters for the key exchange.

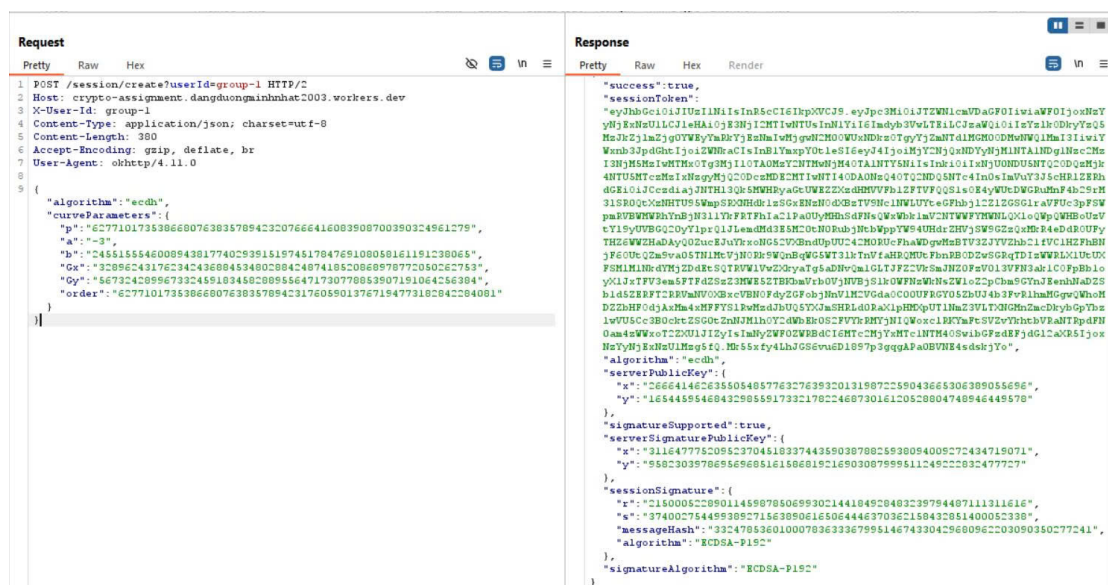


Figure 9: Initial Handshake Request and Response

3.1.1.1 Request Analysis:

The client sends a POST request to `/session/create?userId=group-1`.

- **algorithm:** Set to "ecdh", indicating the use of Elliptic Curve Diffie-Hellman for key agreement.
- **curveParameters:** Specifies the domain parameters for the elliptic curve (identified as NIST P-192 based on the prime p):
 - **p, a, b:** Define the curve equation $y^2 = x^3 + ax + b \pmod{p}$.
 - **Gx, Gy:** The coordinates of the base point (generator).
 - **order:** The order of the subgroup generated by G .



3.1.1.2 Response Analysis: The server accepts the parameters and establishes the session context.

- **sessionToken:** A JSON Web Token (JWT) used to identify and authenticate this specific session in subsequent requests.
- **serverPublicKey:** The server's ephemeral ECDH public key (x, y) . The client will use this to compute the shared secret.
- **serverSignaturePublicKey:** The server's long-term ECDSA public key used to verify the server's identity.
- **sessionSignature:** Contains the ECDSA signature components (r, s) and the message hash. This proves that the **serverPublicKey** actually came from the server, preventing Man-in-the-Middle attacks during the handshake.

3.1.2 Step 2: Session Establishment

After verifying the server's signature, the client generates its own key pair and sends the public component to the server.

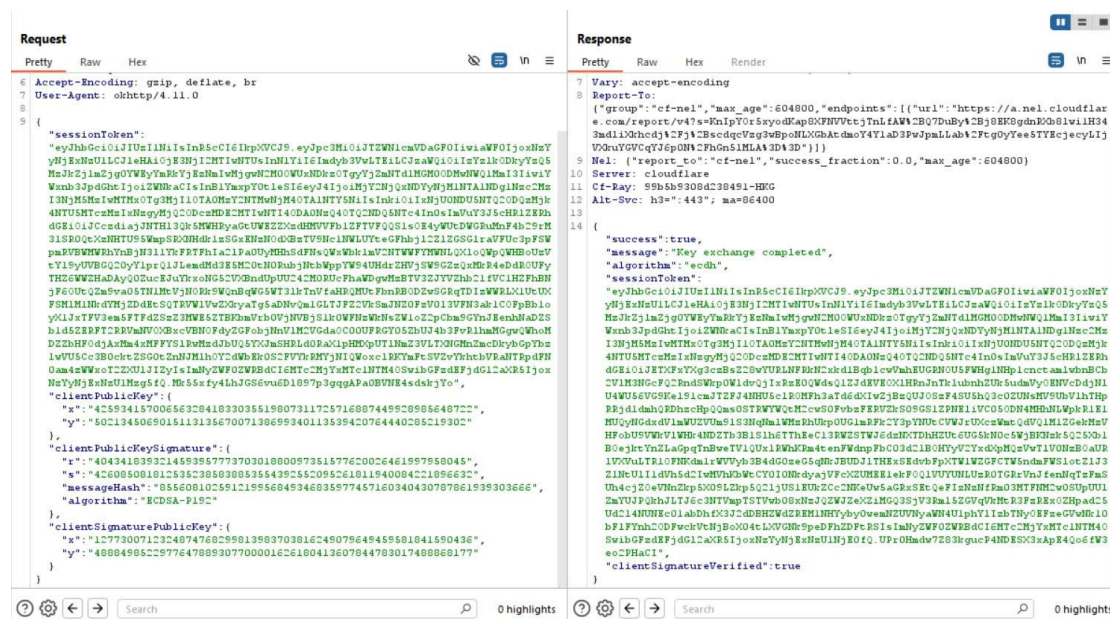


Figure 10: Client Public Key Submission

3.1.2.1 Request Analysis:

- **sessionToken:** The JWT received in Step 1, linking this request to the initiated handshake.
- **clientPublicKey:** The client's ECDH public key (x, y) . At this point, both parties can independently calculate the same shared secret $K = d_{client} \times Q_{server} = d_{server} \times Q_{client}$.
- **clientPublicKeySignature:** The client signs its own public key using ECDSA. This ensures non-repudiation and authentication of the client.



- **clientSignaturePublicKey**: The public verification key corresponding to the private signing key used by the client.

3.1.2.2 Response Analysis:

- **success: true**: Indicates the server successfully verified the client's signature and derived the shared secret.
- **message: "Key exchange completed"**. The secure channel is now established.

3.1.3 Step 3: Subsequent Encrypted Messaging

With the shared secret established, the client and server switch to encrypted communication. The specific symmetric encryption algorithm (likely AES) uses the derived shared secret.

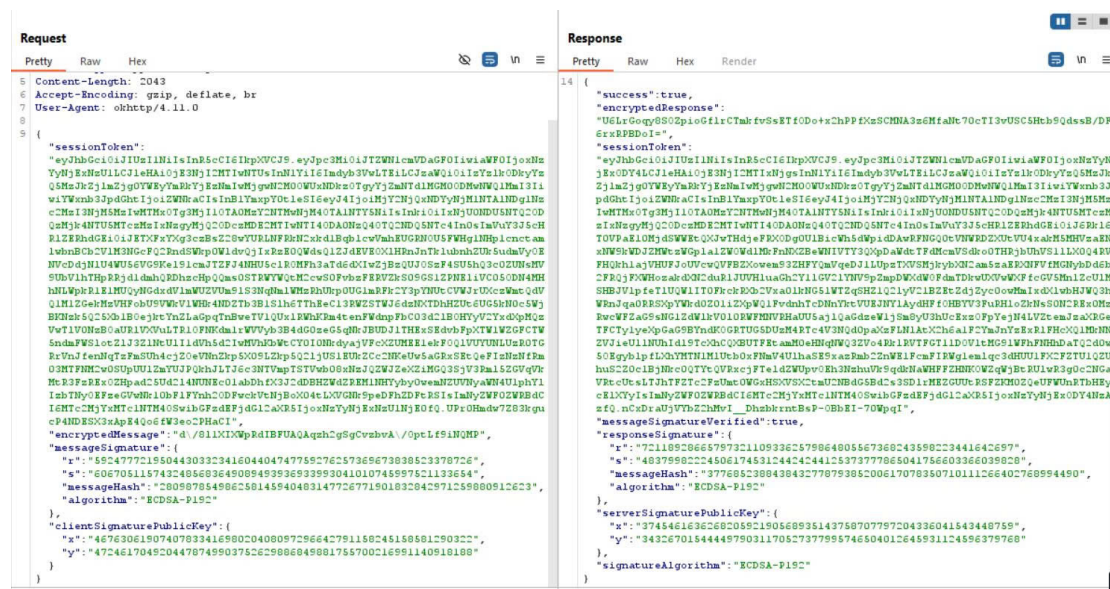


Figure 11: Encrypted Data Exchange

3.1.3.1 Request Analysis:

- **encryptedMessage**: The actual application data (payload), encrypted using the shared session key. It appears as a Base64 encoded string.
- **messageSignature**: An ECDSA signature over the message (or its hash). This ensures data integrity; if the encrypted message is altered in transit, the signature verification will fail.
- **clientSignaturePublicKey**: Included to allow the server to verify the signature of this specific message.

3.1.3.2 Response Analysis:

- **encryptedResponse**: The server's response, also encrypted with the shared key.

- **messageSignatureVerified: true:** Confirms the server validated the integrity of the client's request.
- **responseSignature:** The server signs its response, providing mutual authentication for every single message exchanged within the session.

3.2 Re-implementation and Protocol Reconstruction

Based on the reverse engineering of the decompiled Java classes (`ECDHKeyExchange`, `ECDSASignature`, and `CryptoManager`), the cryptographic protocol was reconstructed. The application implements a custom security layer using NIST P-192 for Elliptic Curve Cryptography.

The protocol flow consists of three distinct phases:

3.2.1 Phase 1: Initial Handshake (Parameter Negotiation)

The client initiates the connection by defining the elliptic curve parameters. The server responds with its ephemeral public key and a digital signature to prevent Man-in-the-Middle attacks.

Client Request Logic:

```
1 // Defined in ECDHKeyExchange.java
2 JSONObject params = new JSONObject();
3 params.put("algorithm", "ecdh");
4 params.put("curve", "P-192");
5 params.put("p", "6277101735..."); // Curve P-192 prime
6 params.put("gx", "3289624317..."); // Generator X
7 // ... sends parameters to /session/create
```

Server Response Logic:

```
1 // Server generates ephemeral key pair
2 serverPriv, serverPub = GenerateKeyPair(P-192);
3 // Server signs its public key
4 signature = Sign(serverPub, serverLongTermKey);
5 return { sessionToken, serverPub, signature };
```

3.2.2 Phase 2: Session Establishment (Key Agreement)

The client validates the server's signature, generates its own key pair, and computes the shared secret. The shared secret is then processed via PBKDF2 to derive the symmetric AES encryption key.

Client Request Logic:

```
1 // 1. Verify Server Signature
2 verify(serverPub, signature, serverLongTermPub);
3
4 // 2. Generate Client Keys & Shared Secret
5 clientPriv, clientPub = GenerateKeyPair();
6 sharedPoint = ScalarMult(serverPub, clientPriv);
7
8 // 3. Derive AES Key (CryptoManager.java)
9 aesKey = PBKDF2(secret=sharedPoint.x, salt=0, iter=1000);
10
```

```
11 // 4. Send Public Key
12 clientSig = Sign(clientPub, clientPriv);
13 send(sessionToken, clientPub, clientSig);
```

Server Response Logic:

```
1 // Server calculates the same shared secret
2 sharedPoint = ScalarMult(clientPub, serverPriv);
3 aesKey = PBKDF2(secret=sharedPoint.x, ...);
4
5 if (verify(clientPub, clientSig)) {
6     return { success: true, message: "Key exchange completed" };
7 }
```

3.2.3 Phase 3: Secure Messaging

All subsequent communications are encrypted using AES-GCM (Galois/Counter Mode). To ensure non-repudiation and integrity, every message payload is also signed using ECDSA.

Client Request Logic:

```
1 // Encrypt Payload
2 iv = Random(12);
3 ciphertext = AES_GCM_Encrypt(plaintext, aesKey, iv);
4
5 // Sign Ciphertext
6 signature = Sign(ciphertext, clientPriv);
7
8 send(ciphertext, signature);
```

Server Response Logic:

```
1 // Verify and Decrypt
2 if (Verify(ciphertext, signature)) {
3     plaintext = AES_GCM_Decrypt(ciphertext, aesKey);
4
5     // Process request and prepare response
6     responseCipher = AES_GCM_Encrypt(responsePayload, aesKey);
7     responseSig = Sign(responseCipher, serverPriv);
8
9     return { responseCipher, responseSig };
10 }
```

4 Exploitation & Proof-of-Concept

This section details the step-by-step reconstruction of the "Invalid Curve Attack" used to extract the server's private key. The attack leverages the server's failure to validate whether the client-provided elliptic curve parameters belong to the standard P-192 curve.

4.1 Step 1: Finding Weak Curve Parameters

Before initiating the attack, it is necessary to identify a weak Elliptic Curve $E'(a, b)$ and a base point G' such that the curve's Order contains many small prime factors.

Objective: To create a mathematical problem where the Discrete Logarithm Problem (DLP) is computationally easy to solve.

Methodology:

- Iterate through small integer values for parameters a and b (e.g., from -5 to 5).
- Calculate the order of the corresponding curves over the finite field \mathbb{F}_p .
- Select a curve where the order is a "smooth number" (composed of small prime factors).

Result: We identified a malicious set of parameters:

$$a = -5, \quad b = 0$$

The resulting weak order is:

$$\text{Order}_{\text{weak}} = 78463771692333509547947367790095830201048858754879062016$$

Factorization analysis reveals that this order is highly composite ($2^{60} \times 3 \times 17 \times 257 \times 641 \times 65537 \times 274177 \times 6700417 \times 67280421310721$), making it vulnerable to Pohlig-Hellman attacks.

4.2 Step 2: Sending Malicious Request & Extracting Public Key

In this step, we force the server to perform scalar multiplication using its secret private key on the weak curve identified in Step 1.

Methodology:

- A POST request is sent to the endpoint `/session/create`.
- The JSON payload is modified to inject the weak curve parameters ($a = -5, b = 0$) instead of the standard P-192 parameters.

Vulnerability: The server accepts the parameters without validation and uses its existing Private Key (d) to compute the public key point:

$$Q_{\text{server_weak}} = d \times G_{\text{weak}}$$

Result: The server responds with a `serverPublicKey`. Critically, this point lies on the weak curve under our control, leaking information about d relative to the weak order.

4.3 Step 3: Solving the Discrete Logarithm Problem

Using the public key obtained in Step 2, we recover the private key modulo the weak order.

Algorithms Used:

- **Pohlig-Hellman Algorithm:** Decomposes the large DLP into smaller sub-problems based on the prime factors of the order.
- **Baby-Step Giant-Step (BSGS):** Solves each smaller sub-problem efficiently.
- **Chinese Remainder Theorem (CRT):** Recombines the results from the sub-problems to find the solution modulo $\text{Order}_{\text{weak}}$.

Result: We successfully computed a value x such that:

$$d \equiv x \pmod{\text{Order}_{\text{weak}}}$$

The calculated residue x is approximately 4.913×10^{55} .

4.4 Step 4: Restoring the Original Private Key

Since the Order_{weak} is slightly smaller than the full key space of the standard P-192 curve, the actual private key d might be larger than Order_{weak} . We must find the integer coefficient k .

Formula:

$$d = x + k \times \text{Order}_{weak}$$

Methodology:

- A brute-force loop iterates through potential values of k (e.g., range $[-100, 100]$).
- For each k , a candidate key $d_{candidate}$ is generated.
- The candidate key is verified against the **real** P-192 curve public key (obtained from a legitimate request):

$$P_{check} = d_{candidate} \times G_{real}$$

- If P_{check} matches the server's real public key, the private key is confirmed.

Final Result: The attack successfully recovered the server's private key with $k = 23$:

$$d = 1853800949957290197646800317665265411650411285239850323051 \quad (2)$$

Github repo for attack scripts and solution codes used in the assignment:

https://github.com/trekof/ACCT_assignment