Cryptopals.com

Parth Thakkar
Developing the Industrial Internet of Things
Sunday, March 8, 2025

Background
My objective was to develop a deeper understanding of cryptographic vulnerabilities by implementing and executing attacks that exploit them, using Python as my primary tool.

The Cryptopals challenges are organized into eight sets of increasing difficulty. I focused on Sets 1 and 2, which cover fundamentals of cryptographic operations and attacks against block ciphers.
Challenges Completed:

Set 1: Convert hex to base64, Fixed XOR, Single-byte XOR cipher, Detect single-character XOR, Implement repeating-key XOR, Break repeating-key XOR, AES in ECB mode, Detect AES in ECB mode
Set 2: Implement PKCS#7 padding, Implement CBC mode, ECB/CBC detection oracle, Byte-at-a-time ECB decryption, ECB cut-and-paste, Byte-at-a-time ECB decryption (harder), CBC bitflipping attacks

What I did
I spent 42+ hours on this assignment. For this paper I chose to do Option 1 which is writing cryptography codes in python from cryptopals.com

Technical Setup:

I implemented solutions using Python 3.9, leveraging the `cryptography` library for core AES functionality while writing my own code for most operations to ensure a deep understanding of the underlying mechanisms. I deliberately avoided using high-level cryptographic functions except when explicitly instructed, forcing myself to implement operations like XOR, padding, and mode operations manually.

Methodology:

For each challenge, I followed a consistent approach:

1. Read the challenge description and understand the cryptographic concept involved
2. Research the necessary background if unfamiliar with the concept
3. Write code to implement the required operations or attacks
4. Test against provided examples
5. Refine the solution until successful
6. Document key insights and learning outcomes

This approach provided a structured path through increasingly complex cryptographic concepts while ensuring thorough comprehension at each step.

3. Key Challenges and Solutions

Challenge: Break Repeating-Key XOR (Vigenère Cipher)

Problem Description:
This challenge involved breaking a message encrypted with repeating-key XOR (also known as a Vigenère cipher), an encryption technique that uses a repeating key to XOR against the plaintext. While significantly stronger than single-byte XOR, repeating-key XOR remains vulnerable to statistical analysis.

Approach:
Breaking repeating-key XOR requires several steps:

1. Determine the likely key size
2. Split the ciphertext into blocks based on key size
3. Solve each position as a single-byte XOR problem
4. Combine the results to obtain the complete key

The key insight was using the Hamming distance (number of differing bits) between ciphertext blocks to identify the most likely key size. For true random data, the normalized Hamming distance between blocks should be consistent, but when blocks are encrypted with the same key, patterns emerge.

Implementation:

```
def hamming_distance(bytes1, bytes2):
    """Calculate the Hamming distance between two byte strings."""
    if len(bytes1) != len(bytes2):
        raise ValueError("Byte strings must be of equal length")

    # XOR the bytes together - bits that differ will be 1 in the result
    xor_result = bytes([b1 ^ b2 for b1, b2 in zip(bytes1, bytes2)])

    # Count the number of 1 bits in each byte
    distance = 0
    for byte in xor_result:
        # Count the 1 bits in this byte
        while byte:
            distance += byte & 1
            byte >>= 1

    return distance

def find_key_size(ciphertext, min_size=2, max_size=40):
    """Find the most likely key size for repeating-key XOR."""
    best_distances = []

    for key_size in range(min_size, max_size + 1):
        # Skip if we don't have enough ciphertext
        if len(ciphertext) < key_size * 4:
            continue
        # Take multiple blocks and calculate average distance
        distances = []
        for i in range(4):
            block1 = ciphertext[i * key_size:(i + 1) * key_size]
            block2 = ciphertext[(i + 1) * key_size:(i + 2) * key_size]
            distance = hamming_distance(block1, block2) / key_size
            distances.append(distance)

        # Average the distances
        avg_distance = sum(distances) / len(distances)
        best_distances.append((key_size, avg_distance))

    # Sort by distance (lowest first)
    best_distances.sort(key=lambda x: x[1])
    return best_distances
```

Once the key size was determined, I split the ciphertext into blocks and solved each position as a single-byte XOR problem, using letter frequency analysis to score potential plaintext candidates. The highest scoring solutions for each position revealed the complete key.

Results:
This approach successfully broke the repeating-key XOR encryption, revealing both the key and the original message. The decrypted text was a passage from a song, confirming the correctness of the approach.

Lessons Learned:
This challenge demonstrated how statistical patterns persist even in seemingly complex encryption schemes. It also highlighted the importance of key management - a repeating key, no matter how random, introduces patterns that can be exploited.

Challenge: Byte-at-a-Time ECB Decryption

Problem Description:
This challenge simulated an oracle function encrypting messages as:

```
AES-128-ECB(your-string || unknown-string, random-key)
```

The goal was to decrypt the unknown string by making repeated calls to the oracle with carefully chosen inputs.

Approach:
I exploited a fundamental weakness in ECB (Electronic Codebook) mode: identical plaintext blocks encrypt to identical ciphertext blocks. By controlling part of the input and observing how the output changes, I could determine the unknown bytes one at a time.

The attack follows these steps:

1. Determine the block size by observing when output length changes
2. Confirm the oracle is using ECB mode by checking for repeated blocks

3. Craft inputs one byte short of a complete block
4. Build a dictionary mapping every possible next byte to its encrypted output
5. Match the oracle's output with this dictionary to determine each byte

Implementation:

```
def byte_at_a_time_ecb_decryption(oracle_function):
    """Decrypt the unknown string using byte-at-a-time ECB decryption."""
    # Determine the block size
    block_size = detect_block_size(oracle_function)
    print(f"Detected block size: {block_size}")

    # Detect that the function is using ECB
    is_ecb = detect_ecb_mode(oracle_function, block_size)
    print(f"ECB mode detected: {is_ecb}")

    # Decrypt the unknown string byte by byte
    decrypted = bytearray()

    # First, determine the length of the unknown string
    base_length = len(oracle_function(b""))

    # Continue until we've decrypted the entire unknown string
    for byte_pos in range(base_length):
        # Determine which block this byte will be in
        block_index = byte_pos // block_size

        # Create input that's one byte short of a full block
        padding_length = (block_size - 1) - (byte_pos % block_size)
        crafted_input = b"A" * padding_length

        # Get the target ciphertext block
        target_ciphertext = oracle_function(crafted_input)
        target_block = target_ciphertext[block_index * block_size:(block_in-
dex + 1) * block_size]

        # Try all possible bytes for the last position
        found = False
        for byte_value in range(256):
            test_input = crafted_input + decrypted + bytes([byte_value])
            test_ciphertext = oracle_function(test_input)
            test_block = test_ciphertext[block_index * block_size:(block_in-
dex + 1) * block_size]

            if test_block == target_block:
                decrypted.append(byte_value)
                found = True
                break

        if not found:
            break

    return decrypted
```

Results:
The attack successfully decrypted the unknown string, which turned out to be lyrics from a song. This demonstrates how ECB mode can be exploited to completely break the confidentiality of encrypted data, even without knowing the key.

Lessons Learned:
This challenge revealed why ECB mode is considered insecure for most applications. Its deterministic nature allows attackers to learn patterns in the plaintext by observing the ciphertext. This underscores why modern applications should use authenticated encryption modes like GCM instead.

Challenge: CBC Bitflipping Attack

Problem Description:
This challenge presented an encryption oracle that:

1. Takes user input and sanitizes it (removing ';' and '=' characters)
2. Places the sanitized input within a specific string format
3. Encrypts the result using AES-CBC with a random key

The goal was to manipulate the ciphertext to decrypt with ";admin=true;" in the plaintext, despite the sanitization.

Approach:
The key insight is that in CBC mode, modifying a byte in ciphertext block N-1 causes a predictable change in the plaintext of block N after decryption. This occurs because during CBC decryption, each plaintext block is XORed with the previous ciphertext block.

I exploited this property by:

1. Creating a ciphertext with known plaintext at a predictable position
2. Determining which ciphertext blocks influence the target plaintext area
3. Calculating the XOR difference between the current plaintext and desired plaintext
4. Applying those XOR differences to the previous ciphertext block

Implementation:

```
def perform_bitflipping_attack():
    """Perform a CBC bitflipping attack to forge admin privileges."""
    # Create an encrypted message with known plaintext
    user_input = "A" * 16  # One full block of As
    iv, ciphertext = encrypt_userdata(user_input)

    # Convert to bytearray for modification
    ciphertext_array = bytearray(ciphertext)

    # Determine where our controlled input appears in the plaintext
    prefix = "comment1=cooking%20MCs;userdata="
    prefix_blocks = len(prefix) // 16  # Full blocks in the prefix
    prefix_remainder = len(prefix) % 16  # Remaining bytes

    # Calculate which block to modify and the offset
    target_block = prefix_blocks  # The block containing our input

    # What we want to inject: ";admin=true;"
    injection = b";admin=true;"

    # Original plaintext in our controlled block
    original = b"A" * 16

    # For each position we want to change
    for i in range(len(injection)):
        # Position in the ciphertext to modify
        pos = (target_block - 1) * 16 + prefix_remainder + i

        # Calculate XOR of: original_byte XOR injection_byte
        xor_value = original[i] ^ injection[i]

        # Apply the change to the previous block (or IV)
        if target_block == 0:
            iv_array = bytearray(iv)
            iv_array[prefix_remainder + i] ^= xor_value
            iv = bytes(iv_array)
        else:
            ciphertext_array[pos] ^= xor_value

    return iv, bytes(ciphertext_array)
```

Results:
The attack successfully produced a modified ciphertext that, when decrypted, contained ";admin=true;" in the plaintext. This demonstrates how an attacker can manipulate encrypted data to produce specific plaintext values, bypassing input sanitization.

Lessons Learned:
This challenge illustrates why encryption alone is insufficient to ensure data integrity. Without message authentication (such as HMAC or using authenticated encryption modes like GCM), attackers can modify ciphertext to produce predictable changes in the plaintext after decryption. This attack has real-world implications for systems that encrypt cookies or authentication tokens using CBC without proper integrity checks.

4. Cryptographic Concepts Explored

Through these challenges, I gained practical experience with several fundamental cryptographic concepts:

Symmetric Encryption

Working with AES encryption provided hands-on understanding of:

- **Block ciphers** - Encrypting fixed-size blocks of data (16 bytes for AES)
- **Modes of operation** - How block ciphers handle data larger than one block

- **Padding schemes** - Ensuring data fits evenly into blocks

Encryption Modes

I implemented and attacked different modes of encryption:

- **ECB (Electronic Codebook)** - The simplest mode, which encrypts each block independently
- **CBC (Cipher Block Chaining)** - Each block is XORed with the previous ciphertext block before encryption

ECB mode revealed serious weaknesses - identical plaintext blocks produce identical ciphertext blocks, leaking information about patterns in the data. This was vividly demonstrated when detecting ECB-encrypted data among random samples.

CBC mode provided better security by chaining blocks together, but introduced its own vulnerabilities through the bitflipping attack. This highlighted why modern systems prefer authenticated encryption modes like GCM.

Padding Methods

PKCS#7 padding proved crucial for encrypting data that isn't a multiple of the block size. Implementing and validating this padding revealed subtleties that could lead to padding oracle attacks if not handled carefully.

Cryptanalysis Techniques

The challenges introduced several cryptanalysis methods:

- **Frequency analysis** - Breaking single-byte XOR by analyzing letter frequencies
- **Statistical distance measures** - Using Hamming distance to detect patterns
- **Known-plaintext attacks** - Leveraging known sections of plaintext to deduce keys
- **Chosen-plaintext attacks** - Crafting specific inputs to learn about encryption behavior

Vulnerabilities in Implementation

Perhaps most importantly, these challenges demonstrated that secure algorithms can be undermined by implementation choices:

- Using predictable keys or IVs
- Choosing weak modes of operation
- Failing to validate message integrity
- Leaking information through error messages or timing

5. Correlation with Class Materials

The Cryptopals challenges significantly reinforced and expanded upon concepts from my cryptography coursework. While lectures provided theoretical foundations, these hands-on exercises transformed abstract concepts into concrete understanding.

Reinforced Concepts

Several theoretical concepts became much clearer through implementation:

1. **Block Cipher Modes**: The classroom description of ECB mode vulnerabilities became dramatically apparent when I could actually see patterns persisting in ECB-encrypted data. Similarly, the chaining mechanism of CBC mode became intuitive once I implemented it myself.
2. **XOR Properties**: The mathematical properties of XOR operations, particularly its invertibility ($A \oplus B \oplus B = A$), moved from theoretical knowledge to practical tool when exploiting the CBC bitflipping vulnerability.
3. **Frequency Analysis**: Statistical approaches to cryptanalysis, which seemed abstract in lectures, became powerful tools when breaking substitution ciphers.

New Insights

The challenges also provided insights that weren't fully covered in class:

1. **Implementation Details Matter**: Small implementation choices can completely undermine theoretically secure algorithms. For instance, the difference between using a random IV versus a fixed IV in CBC mode is the difference between security and complete system compromise.
2. **Oracle Attacks**: The concept of using a system as an "oracle" to leak information about encrypted data was only briefly mentioned in class, but became central to several challenges. This highlighted how systems can leak information in unexpected ways.
3. **Practical Exploitation**: While class covered theoretical attacks, implementing them demonstrated the practical difficulty and specific techniques required. For example, knowing that ECB is vulnerable is different from actually performing a byte-at-a-time decryption attack.

Misconceptions Cleared

Some misconceptions from class were corrected:

1. **Block Size vs. Key Size**: I had initially confused these concepts, but the challenges clarified that block size (the amount of data encrypted at once) is distinct from key size (which determines security against brute force).
2. **Security of CBC Mode**: While class presented CBC as significantly more secure than ECB, the bitflipping attack demonstrated that CBC without message authentication is still vulnerable in many scenarios.

6. Conclusion

The Cryptopals challenges provided an invaluable hands-on education in cryptographic implementation and exploitation. Working through these exercises transformed my understanding from theoretical knowledge to practical skill, revealing the critical gap between understanding cryptographic theory and implementing secure systems.

Several key lessons stand out:

1. **Mode selection matters**: The choice between ECB, CBC, and other modes has profound security implications. Modern applications should use authenticated encryption modes like GCM.
2. **Integrity is as important as confidentiality**: Many of the successful attacks exploited the lack of integrity checking. Message authentication codes (MACs) or authenticated encryption are essential.
3. **Implementation details can undermine theoretical security**: Even with a secure algorithm, choices like key management, padding handling, and error responses can create exploitable vulnerabilities.
4. **Practical attacks are often more feasible than theory suggests**: While cryptographic algorithms may be mathematically secure, implementation patterns often enable practical attacks that don't require breaking the underlying mathematics.

This experience has fundamentally changed how I approach cryptographic security. Rather than viewing it as a collection of algorithms and theories, I now see it as a complex system where implementation details, user interaction, and error handling all contribute to overall security.

For future exploration, I'm interested in exploring more modern attacks like padding oracles, timing attacks, and side-channel analysis, which represent the frontier of applied cryptographic vulnerabilities. I also plan to explore how formal verification techniques might help prove the absence of certain classes of implementation vulnerabilities.

The most valuable insight from this project is the realization that cryptography is not simply about choosing the right algorithm, but about understanding the entire system and its potential weaknesses. As Bruce Schneier aptly noted, "Anyone can design a security system that he himself cannot break." The true test is creating systems that withstand attacks from motivated adversaries with diverse techniques at their disposal.