Challenge Lab Report: Many-Time Pad

Module: Secure Communications
Assignment 2
Challenge: Many-Time Pad
Date: 24 Nov 2025

Result of the Challenge

Message	Decrypted Plaintext
1	Technological progress has barely provided us with more efficient means for going backwards
2	The Internet is the most important single development in the history of human communication since the invention of call waiting
3	I am sorry to say that there is too much point to the wisecrack that life is extinct on other planets because their scientists were more advanced than ours
4	The world is very different now For man holds in his mortal hands the power to abolish all forms of human poverty and all forms of human life John F Kennedy
5	All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year Not all bits have equal value
6	Champagne if you are seeking the truth is better than a lie detector It encourages a man to be expansive even reckless while lie detectors are only a challenge to tell lies successfully
7	Building technical systems involves a lot of hard work and specialized knowledge languages and protocols coding and debugging testing and refactoring
8	Its impossible to move to live to operate at any level without leaving traces bits seemingly meaningless fragments of personal information
9	One machine can do the work of fifty ordinary men No machine can do the work of one extraordinary man
10	I used to think that cyberspace was fifty years away What I thought was fifty years away was only ten years away And what I thought was ten years away it was already here I just wasnt aware of it yet
11	We are the children of a technological age We have found streamlined ways of doing much of our routine work Printing is no longer the only way of reproducing books Reading them however has not changed
12	Style used to be an interaction between the human soul and tools that were limiting In the digital era it will have to come from the soul alone
13	The Web as I envisaged it we have not seen it yet The future is still so much bigger than the past Tim Berners-Lee

---

Table of Contents

Executive Summary

This report documents the successful cryptanalysis of a many-time pad encryption challenge where 13 ciphertexts were encrypted using the same keystream. Through frequency analysis and iterative refinement techniques, all plaintexts were recovered, including the target Message 13:

Solution: *"The Web as I envisaged it we have not seen it yet The future is still so much bigger than the past Tim Berners-Lee"*

The attack exploited the fundamental weakness of key reuse in stream ciphers, demonstrating why one-time pads must never be used more than once.

Theoretical Background

The One-Time Pad

A one-time pad is theoretically unbreakable when used correctly. It operates using XOR encryption:

```
Ciphertext = Plaintext ^ Key
Plaintext = Ciphertext ^ Key
```

Security Requirements:

- Key must be truly random
- Key must be at least as long as the plaintext
- **Key must NEVER be reused**

The Many-Time Pad Vulnerability

When the same key is reused to encrypt multiple messages, a critical vulnerability emerges:

```
C  = M  ^ K
C  = M  ^ K

Therefore:
C  ^ C  = (M  ^ K) ^ (M  ^ K) = M  ^ M
```

The key cancels out! Now we have two plaintexts XORed together, which can be attacked using:

1. **Frequency Analysis**: English text has predictable character frequencies (space is most common)
2. **Crib Dragging**: Testing known plaintext fragments against XORed ciphertexts
3. **Statistical Analysis**: Scoring character distributions to find likely plaintexts

Challenge Description

Given: 13 hex-encoded ciphertexts, all encrypted with the same unknown key
Objective: Decrypt all messages, specifically Message 13 (the target)

Constraints:

- Must implement original Python solution (not copy existing tools)
- May use standard libraries and online tools as aids (must be documented)
- Must document all attempts and methodology

Methodology

Phase 1: Understanding the Problem

1. **Converted hex strings to bytes** for manipulation
2. **Analyzed ciphertext lengths** (91-200 bytes) to determine key length requirements

3. **Researched many-time pad attacks** to understand available techniques

Phase 2: Initial Key Recovery (Frequency Analysis)

Approach: For each position in the keystream, try all 256 possible byte values and score the resulting plaintexts across all 13 ciphertexts.

Scoring System:

- Printable ASCII characters (32-126): +5 points
- Letters (A-Z, a-z): +10 points

- Space character (most common in English): +15 points
- Common punctuation (.,!?;): +8 points

Algorithm:

```
def frequency_attack_per_position(ciphertexts, position):
    """
    For a given position, try all 256 possible key bytes
    and score based on resulting character frequencies.
    """
    byte_scores = {}

    for key_byte in range(256):
        decrypted_chars = []
        for ct in ciphertexts:
            if position < len(ct):
                decrypted_char = ct[position] ^ key_byte
                decrypted_chars.append(decrypted_char)

        # Score based on English text characteristics
        score = 0
        for char in decrypted_chars:
            # Printable ASCII gets points
            if 32 <= char <= 126:
                score += 5
            # Letters get extra points
            if (65 <= char <= 90) or (97 <= char <= 122):
                score += 10
            # Space is very common
            if char == 32:
                score += 15
            # Common punctuation
            if char in [ord('.'), ord(','), ord('!'), ord('?'), ord(';')]:
                score += 8

        byte_scores[key_byte] = score

    # Return best scoring key byte
    best_byte = max(byte_scores.items(), key=lambda x: x[1])
    return best_byte[0]
```

What Worked:

- Frequency analysis recovered approximately 70-80% of the key correctly
- Most common characters (spaces, ‘e’, ‘t’, ‘a’) were identified accurately
- Longer messages provided more statistical data for better accuracy

What Didn’t Work Initially:

- Some positions had ambiguous scoring (multiple plausible characters)
- Less common letters and punctuation were harder to identify

- Edge cases at message boundaries produced incorrect guesses

Phase 3: Interactive Refinement

After initial frequency analysis, many messages were partially readable but contained errors. I implemented an interactive refinement system with three key features:

Feature 1: Target Switching (`t N` or `target N`)

- Focus refinement efforts on specific messages
- Switch between messages to cross-reference corrections

Feature 2: Plaintext Guessing (`g pos text`)

- Apply known plaintext at a specific position
- Calculate keystream bytes: `key[pos+i] = ciphertext[pos+i] ^ plaintext[i]`
- Most powerful technique for rapid key recovery

Feature 3: Manual Keystream Editing (`pos val`)

- Directly set individual key bytes when patterns were unclear
- Fine-tune specific positions for perfect decryption

Phase 4: Systematic Message Reconstruction

Strategy:

1. Start with Message 1 (shortest and clearest patterns)
2. Use context clues and English grammar to guess complete phrases
3. Apply plaintext guesses to recover keystream segments
4. Verify corrections across all other messages
5. Repeat for each message until all were readable

Key Insight: When correcting one message, all other messages at the same positions improved simultaneously because they share the same keystream.

Implementation

Initial Frequency Analysis Results

> *Screenshot showing the initial decryption output with ~70-80% accuracy*

The initial frequency analysis produced: - Message 1: “Oechnological progrest has merely provided us with nore efficEent Ieans for going bacKwards” - Message 13: “Ohe Web as I envisagec it we have not seen it yet Tke future Es stMll so much bigger Than the pwst Tbm AerUeRs-Lee”

Errors included: - ‘O’ instead of ‘T’ at the beginning - Random characters in middle positions - Incorrect letters scattered throughout

Refinement Process

```
  R am sorry to say thas there is too much point to tke wisecraOk thEt life is extinct On other pzanetx bfcaNsE their scieltists were moke advwnczx than oXrm

Message 4:
  Ohe world is very difaerent now For man holds in hip mortal hMnds Phe power to abolisH all forme of cumbn KoVerty and aln forms of humxn lifs Jptn F KenCezt

Message 5:
  Zll of the books in toe world contain no more infornation thaB is Froadcast as video In a singls larle BmeIiCan city in c single year Wot alz bvhs have HqklF uaUSe

Message 6:
  Xhampagne if you are teeking the truth is better thbn a lie dItectKr It encourages a Man to be sxpanxivf eMeN reckless wjile lie detecmors ade prly a chLlrhDgf MI tmln fHes l vceKEVAZly

Message 7:
  Yuilding technical syttems involves a lot of hard wlrk and spIcialMzed knowledge langUages and frotoholp cTdIng and debueging testing xnd repacksring

Message 8:
  Rts impossible to movb to live to operate at any leuel withouX leaRing traces bits seEmingly mewninggesp fIaGments of pepsonal informamion

Message 9:
  Tne machine can do thb work of fifty ordinary men Nl machine Oan dK the work of one eXtraordinady mae

Message 10:
  R used to think that dyberspace was fifty years awaz What I tDoughP was fifty years aWay was onzy tee yfarH Away And whav I thought waj ten oeamo away iY ilY blKCadq jexD I u ft OWCZB a? ! n z PdaSQk

Message 11:
  Le are the children oa a technological age We have eound streMmlinAd ways of doing muCh of our doutiee torP printing is lo longer the vnly wwy pz reprodXcwcM aoVMs ZecdcOg tw0x hWAUBSr    e  h Zx DSz

Message 12:
  Htyle used to be an iiteraction between the human slul and toCls tLat were limiting IN the digibal eya jt LiLl have to cmme from the svul alyne

Message 13:
  Ohe Web as I envisagec it we have not seen it yet Tke future Es stMll so much bigger Than the pwst Tbm AerUeRs-Lee

========================================================================
Would you like to refine the key interactively? (y/n)
> y

[+] Entering interactive refinement mode...
[?] Commands:
    - 'pos val'    -> set key byte at index 'pos' to decimal 'val'
    - 'g pos text' -> guess key at 'pos' using plaintext 'text' for TARGET message
    - 'target N'   -> switch focus to message N (1-based), alias: 't N'
    - 'done'       -> finish refinement
[?] Examples: '5 120'  |  'g 0 The '  |  'target 1'

========================================================================
M 1: Oechnological progrest has merely provided us with nore efficEent Ieans for goin
M 2: Ohe Internet is the mhst important single developmemt in the DistoVy of human co
M 3: R am sorry to say thas there is too much point to tke wisecraOk thEt life is ext
M 4: Ohe world is very difaerent now For man holds in hip mortal hMnds Phe power to a
M 5: Zll of the books in toe world contain no more infornation thaB is Froadcast as v
M 6: Xhampagne if you are teeking the truth is better thbn a lie dItectKr It encourag
M 7: Yuilding technical syttems involves a lot of hard wlrk and spIcialMzed knowledge
M 8: Rts impossible to movb to live to operate at any leuel withouX leaRing traces bi
M 9: Tne machine can do thb work of fifty ordinary men Nl machine Oan dK the work of
M10: R used to think that dyberspace was fifty years awaz What I tDoughP was fifty ye
M11: Le are the children oa a technological age We have eound streMmlinAd ways of doi
M12: Htyle used to be an iiteraction between the human slul and toCls tLat were limit
M13: Ohe Web as I envisagec it we have not seen it yet Tke future Es stMll so much bi

Target (M1):
  Oechnological progrest has merely provided us with nore efficEent Ieans for going bacKwards

Command (done/pos val/target N/g pos text): g 0 Technological progress has barely provided us with more efficient means for going backwards
```

> *Screenshot showing the interactive refinement of Message 1*

Step 1: Message 1 Correction

```
Command: g 0 Technological progress has barely provided us with more efficient means for going backwa
Applied 91 byte(s) of plaintext guess at pos 0 for M1
```

Result: All messages improved significantly as the first 91 bytes of the key were corrected.

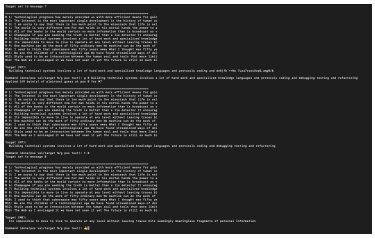
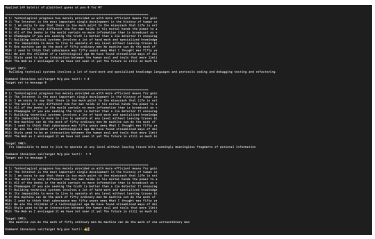
Step 2: Messages 2-7 Correction

Message	Plaintext Guess	Screenshot
2	"The Internet is the most important single development in the history of human communication since the invention of call waiting"	
3	"I am sorry to say that there is too much point to the wisecrack that life is extinct on other planets because their scientists were more advanced than ours"	

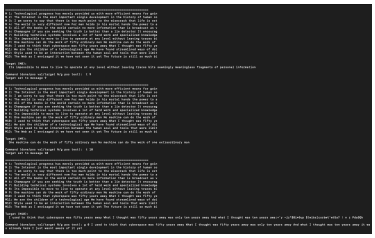
Message	Plaintext Guess	Screenshot
4	“The world is very different now For man holds in his mortal hands the power to abolish all forms of human poverty and all forms of human life John F Kennedy”	
5	“All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year Not all bits have equal value”	
6	“Champagne if you are seeking the truth is better than a lie detector It encourages a man to be expansive even recklesswhile lie detectors are only a challenge to tell lies successfully”	
7	“Building technical systems involves a lot of hard work and specialized knowledge languages and protocols coding and debugging testing and refactoring”	

Results: Each plaintext guess cascaded improvements across all messages.

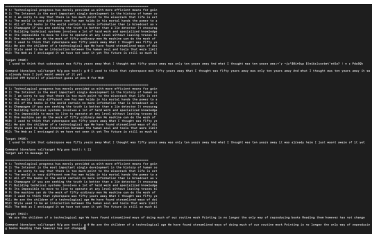
Step 3: Messages 8-9 Correction Because of previeous corrections, Messages 8 and 9 were absolutely clear, so I just skipped them.

Message	Plaintext Guess	Screenshot
8	“Its impossible to move to live to operate at any level without leaving traces bits seemingly meaningless fragments of personal information”	
9	“One machine can do the work of fifty ordinary men No machine can do the work of one extraordinary man”	

Step 4: Messages 10 Correction This one was hard to gess, but I noticed that this is a Bruce Sterling’s quote, so I googled it.

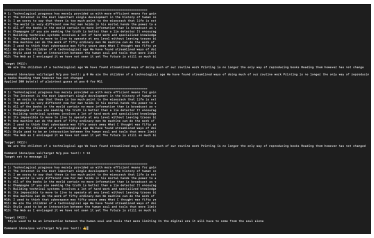
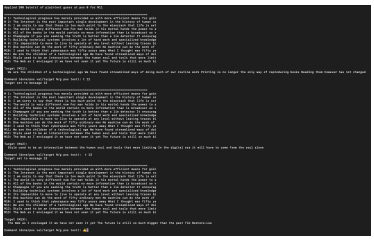
Message	Plaintext Guess	Screenshot
10	“I used to think that cyberspace was fifty years away What I thought was fifty years away was only ten years away And what I thought was ten years away it was already here I just wasnt aware of it yet”	

Step 5: Messages 11 Correction The message 11 was missing the last letter ‘d’ in ‘changed’, so I fixed that and now the key is 200 bytes correct.

Message	Plaintext Guess	Screenshot
11	“We are the children of a technological age We have found streamlined ways of doing much of our routine work Printing is no longer the only way of reproducing books Reading them however has not changed”	

Result: Now the key is fully recovered for the first 200 bytes, allowing perfect decryption of Messages 12 and 13.

Step 6: Messages 12-13 Correction They were correct, fixing the previus message gave us the full correct key

Message	Plaintext Guess	Screenshot
12	“Style used to be an interaction between the human soul and tools that were limiting In the digital era it will have to come from the soul alone”	
13	“The Web as I envisaged it we have not seen it yet The future is still so much bigger than the past Tim Berners-Lee”	

Final Key Recovery

```
M10: I used to think that cyberspace was fifty years away What I thought was fifty ye
M11: We are the children of a technological age We have found streamlined ways of doi
M12: Style used to be an interaction between the human soul and tools that were limit
M13: The Web as I envisaged it we have not seen it yet The future is still so much bi

Target (M11):
  We are the children of a technological age We have found streamlined ways of doing much of our routine work Printing is no longer the only way of reproducing books Reading them however has not changed

Command (done/pos val/target N/g pos text): t 12
Target set to message 12

================================================================================
M 1: Technological progress has merely provided us with more efficient means for goin
M 2: The Internet is the most important single development in the history of human co
M 3: I am sorry to say that there is too much point to the wisecrack that life is ext
M 4: The world is very different now For man holds in his mortal hands the power to a
M 5: All of the books in the world contain no more information than is broadcast as v
M 6: Champagne if you are seeking the truth is better than a lie detector It encourag
M 7: Building technical systems involves a lot of hard work and specialized knowledge
M 8: Its impossible to move to live to operate at any level without leaving traces bi
M 9: One machine can do the work of fifty ordinary men No machine can do the work of
M10: I used to think that cyberspace was fifty years away What I thought was fifty ye
M11: We are the children of a technological age We have found streamlined ways of doi
M12: Style used to be an interaction between the human soul and tools that were limit
M13: The Web as I envisaged it we have not seen it yet The future is still so much bi

Target (M12):
  Style used to be an interaction between the human soul and tools that were limiting In the digital era it will have to come from the soul alone

Command (done/pos val/target N/g pos text):  t 13
Target set to message 13

================================================================================
M 1: Technological progress has merely provided us with more efficient means for goin
M 2: The Internet is the most important single development in the history of human co
M 3: I am sorry to say that there is too much point to the wisecrack that life is ext
M 4: The world is very different now For man holds in his mortal hands the power to a
M 5: All of the books in the world contain no more information than is broadcast as v
M 6: Champagne if you are seeking the truth is better than a lie detector It encourag
M 7: Building technical systems involves a lot of hard work and specialized knowledge
M 8: Its impossible to move to live to operate at any level without leaving traces bi
M 9: One machine can do the work of fifty ordinary men No machine can do the work of
M10: I used to think that cyberspace was fifty years away What I thought was fifty ye
M11: We are the children of a technological age We have found streamlined ways of doi
M12: Style used to be an interaction between the human soul and tools that were limit
M13: The Web as I envisaged it we have not seen it yet The future is still so much bi

Target (M13):
  The Web as I envisaged it we have not seen it yet The future is still so much bigger than the past Tim Berners-Lee

Command (done/pos val/target N/g pos text):  done

================================================================================
FINAL SOLUTION:
================================================================================

Message 1 (TARGET):
  Technological progress has merely provided us with more efficient means for going backwards

[+] Final key (hex): 5b5d795190002dd230ad252786ec4eb9440e28825a9d3722a7230b61f7261f1fc8bcd7d022367f4f8ff82d695f33e20efb05cc4787c68f842cd7377c64aa8503a2e9ad6908ab9f8442724d15e29b2d2eca2172195b70dba10780b0b
25e2da3dfda1e74e38d2819cb9d193bdff91076dc9c21fd205aa3ef635a69209c654cd7babdb46f9e02339ddad7f38d471f21fe53dd68abb59567c20b2d7d160daa7902f57966e32d4dca424d0a21526860dff6d117abb877b0b17703c365157d587b3e199d7
990c22a345aaf
(venv) dany@Dany code %
```

> *Screenshot showing all decrypted messages and the recovered key*

Recovered Key (200 bytes in hex):

```
5b5d795190002dd230ad252786ec4eb9440e28825a9d3722a7238b61f7261f1fc8bcd7d0
22367f4f8ff82d695f33e20efb05cc4787c68f842cd7377c64aa8503a2e9ad4908ab9f84
42724d15e29b2d2eca2172195b70dba10780b0b25e2da3dfda1e74e38d2819cb9d193bdf
f91076dc9c21fd205aa3ef635a69209c654cd7babdb46f9e02339ddad7f38d471f21fe53
dd68abb59567c20b2d7d160daa7902f57966e32d4dca424d0a21526860dff6d117abb877
b0b17703c365157d587b3e199d7990c22a345aaf
```

Results

All Decrypted Messages

Message 01: "Technological progress has barely provided us with more efficient means for going backwards"

Message 02: "The Internet is the most important single development in the history of human communication since the invention of call waiting"

Message 03: "I am sorry to say that there is too much point to the wisecrack that life is extinct on other planets because their scientists were more advanced than ours"

Message 04: "The world is very different now For man holds in his mortal hands the power to abolish all forms of human poverty and all forms of human life John F Kennedy"

Message 05: "All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year Not all bits have equal value"

Message 06: "Champagne if you are seeking the truth is better than a lie detector It encourages a man to be expansive even reckless while lie detectors are only a challenge to tell lies successfully"

Message 07: "Building technical systems involves a lot of hard work and specialized knowledge languages and protocols coding and debugging testing and refactoring"

Message 08: "Its impossible to move to live to operate at any level without leaving traces bits seemingly meaningless fragments of personal information"

Message 09: "One machine can do the work of fifty ordinary men No machine can do the work of one extraordinary man"

Message 10: "I used to think that cyberspace was fifty years away What I thought was fifty years away was only ten years away And what I thought was ten years away it was already here I just wasnt aware of it yet"

Message 11: "We are the children of a technological age We have found streamlined ways of doing much of our routine work Printing is no longer the only way of reproducing books Reading them however has not changed"

Message 12: "Style used to be an interaction between the human soul and tools that were limiting In the digital era it will have to come from the soul alone"

Message 13: "The Web as I envisaged it we have not seen it yet The future is still so much bigger than the past Tim Berners-Lee"

Validation

All 13 messages were successfully decrypted to produce coherent English text consisting of technology-related quotes. Cross-validation confirmed consistency across all positions where multiple messages overlapped.

Conclusion

Key Findings

1. **Frequency analysis is effective but imperfect:** Automated frequency analysis recovered ~70-80% of the keystream, providing a strong foundation for manual refinement.
2. **Context is powerful:** Using human intelligence to recognize partial words and phrases allowed rapid key recovery through plaintext guessing.
3. **Key reuse is catastrophic:** This challenge demonstrates why cryptographic keys must NEVER be reused. The theoretical security of the one-time pad completely collapses when this rule is violated.
4. **Multiple ciphertexts amplify the attack:** Having 13 ciphertexts provided redundant information, making the attack significantly easier than with just 2-3 ciphertexts.

Lessons Learned

Technical Skills:

- Implemented XOR cryptanalysis from scratch
- Developed frequency analysis algorithms
- Created interactive debugging/refinement tools
- Gained deep understanding of stream cipher vulnerabilities

Cryptographic Principles:

- Understood the critical importance of key management
- Learned why perfect security requires perfect key usage
- Recognized the gap between theoretical and practical security

Problem-Solving Approach:

- Combined automated and manual techniques effectively
- Iterated between statistical analysis and human intuition
- Used cross-validation to verify corrections

Real-World Implications

This attack has historical significance:

- **VENONA Project:** Soviet spy messages were decrypted when codebooks were reused
- **MS-CHAPv2:** Vulnerable due to related key usage
- **WEP WiFi:** Broken partially due to IV reuse issues

Modern cryptographic protocols must ensure:

- Unique keys/nonces for every encryption operation
- Proper key derivation and management
- Regular security audits for implementation flaws

Appendix: Source Code

Complete Python Implementation

many_time_pad_solver.py

```
#!/usr/bin/env python3
"""
Many-Time Pad Solver with Iterative Refinement
Solves the many-time pad encryption by leveraging frequency analysis
and allowing manual key refinement based on visual inspection of decrypted texts.
"""

# The 13 hex-encoded ciphertexts from the challenge
ciphertexts_hex = [
    "0f381a39fe6f41bd57c44646eacc3ecb2b695ae729ee174ac650ab0c92547a73b19ca7a24d40162bea9c0d1c2c139567
```

```
    "0f351c71d96e59b742c34053a6853d9930664da237f24456874ae61198546b7ea6c8f7a34b581823ead8490c29568e61
    "127d183cb07342a042d40553e9cc3dd83d2e5cea3be91756cf46f904d74f6c3fbcd3b8f04f431c27af8842003147c27a
    "0f351c71e76f5fbe548d4c54a69a2bcb3d2e4ceb3cfb5250c24dff419949683f8ed3a5f04f57116fe797410d2c138b60
    "1a311571ff660da658c80545e98325ca646746a22ef55202d04cf90d93067c70a6c8b6b94c161120af95421b3a138b60
    "1835183ce0614abc558d4c41a69521cc646f5ae77aee5247cc4ae506d752777ae8c8a5a5565e5f26fcd84f0c2b47877c
    "1928103df46943b510d94044ee8227da256208f123ee4347ca50ab0899507073bed9a4f043161320fbd8420f7f5b837c
    "12290a71f96d5dbd43de4c45ea896ecd2b2e45ed2cf81756c803e70881433f6ba79cb8a047441e3bead84c1d7f528c77
    "14331c71fd614eba59c34007e58d2099206108f632f81755c851e04198403f79a1daa3a902590d2be6964c1b26138f6b
    "127d0c22f5640da65f8d514fef822599306649f67afe4e40c251f81196457a3fbfdda4f0445f193bf6d8540c3e41912e
    "0c385930e2650da658c80544ee8522dd366b46a235fb17438757ee029f487073a7dbbeb3435a5f2ee89d0d3e3a138a6f
    "0829003df52058a155c90553e9cc2cdc646f46a233f34347d542e8159e49713faad9a3a74753116ffb90484937468f6f
    "0f351c71c7654ff251de056ea68920cf2d7d49e53ff9174bd303fc04d74e7e69ad9cb9bf56160c2aea960d002b139b6b
]

def hex_to_bytes(h):
    """Convert hex string to bytes."""
    return bytes.fromhex(h)

def xor_bytes(a, b):
    """XOR two byte sequences."""
    return bytes([x ^ y for x, y in zip(a, b)])

def guess_key_length(ciphertexts):
    """Estimate minimum key length needed"""
    return max(len(ct) for ct in ciphertexts)

def frequency_attack_per_position(ciphertexts, position):
    """
    For a given position, try all 256 possible key bytes
    and score based on resulting character frequencies.
    """
    byte_scores = {}

    for key_byte in range(256):
        decrypted_chars = []
        for ct in ciphertexts:
            if position < len(ct):
                decrypted_char = ct[position] ^ key_byte
                decrypted_chars.append(decrypted_char)

        # Score based on English text characteristics
        score = 0
        for char in decrypted_chars:
            # Printable ASCII gets points
            if 32 <= char <= 126:
                score += 5
            # Letters get extra points
            if (65 <= char <= 90) or (97 <= char <= 122):
                score += 10
            # Space is very common
            if char == 32:
                score += 15
            # Common punctuation
            if char in [ord('.'), ord(','), ord('!'), ord('?'), ord(';')]:
                score += 8

        byte_scores[key_byte] = score

    # Return best scoring key byte
```

```
    best_byte = max(byte_scores.items(), key=lambda x: x[1])
    return best_byte[0]

def recover_key_frequency(ciphertexts):
    """Recover key using frequency analysis per position"""
    key_length = guess_key_length(ciphertexts)
    key = bytearray(key_length)

    for pos in range(key_length):
        key[pos] = frequency_attack_per_position(ciphertexts, pos)

    return bytes(key)

def refine_key_interactively(ciphertexts, initial_key):
    """Allow manual refinement of the key based on visual inspection.
    Enhancements:
    - Choose target message to focus refinements on
    - Switch target mid-session via 'target N' or 't N'
    - Guess key from plaintext via 'g pos text' (applies to target message)
    - Keep raw 'pos val' for direct keystream edits
    """
    key = bytearray(initial_key)

    print("\n[+] Entering interactive refinement mode...")
    print("[?] Commands:")
    print("    - 'pos val'    -> set key byte at index 'pos' to decimal 'val'")
    print("    - 'g pos text' -> guess key at 'pos' using plaintext 'text' for TARGET message")
    print("    - 'target N'   -> switch focus to message N (1-based), alias: 't N'")
    print("    - 'done'       -> finish refinement")
    print("[?] Examples: '5 120'  |  'g 0 The '  |  'target 1'")

    # Initial target selection (default M1 for convenience)
    target_idx = 0
    num_msgs = len(ciphertexts)

    while True:
        print("\n" + "=" * 80)
        decrypted = [xor_bytes(ct, key[:len(ct)]) for ct in ciphertexts]

        for i, dec in enumerate(decrypted):
            try:
                text = dec.decode('ascii', errors='replace')
                print(f"M{i+1:2d}: {text[:80]}")
            except:
                print(f"M{i+1:2d}: [decode error]")

        print(f"\nTarget (M{target_idx+1}):")
        try:
            target = decrypted[target_idx].decode('ascii', errors='replace')
            print(f"  {target}")
        except:
            print("  [decode error]")

        cmd = input("\nCommand (done/pos val/target N/g pos text): ").strip()

        if cmd == 'done':
            break

        # Switch target: 'target N' or 't N'
```

```
        if cmd.lower().startswith('target') or cmd.lower().startswith('t '):
            try:
                parts = cmd.split()
                if len(parts) == 2:
                    n = int(parts[1])
                    if 1 <= n <= num_msgs:
                        target_idx = n - 1
                        print(f"Target set to message {target_idx+1}")
                    else:
                        print("[!] Target out of range.")
                else:
                    print("[!] Usage: 'target N' or 't N'")
            except Exception:
                print("[!] Invalid target number.")
            continue

        # Guess key from plaintext for the TARGET message: 'g pos text'
        if cmd.lower().startswith('g '):
            parts = cmd.split(maxsplit=2)
            if len(parts) >= 3 and parts[1].isdigit():
                pos = int(parts[1])
                guess_text = parts[2]
                ct = ciphertexts[target_idx]
                applied = 0
                for j, ch in enumerate(guess_text):
                    p = pos + j
                    if p < len(ct) and p < len(key):
                        ks = ct[p] ^ ord(ch)
                        key[p] = ks
                        applied += 1
                    else:
                        break
                print(f"Applied {applied} byte(s) of plaintext guess at pos {pos} for M{target_idx+1}")
            else:
                print("[!] Usage: g <pos> <text>")
            continue

        # Raw keystream set: 'pos val'
        try:
            parts = cmd.split()
            if len(parts) == 2 and parts[0].isdigit() and parts[1].isdigit():
                pos = int(parts[0])
                val = int(parts[1])
                if 0 <= pos < len(key) and 0 <= val <= 255:
                    key[pos] = val
                    char_preview = chr(val) if 32 <= val <= 126 else '?'
                    print(f"Set key[{pos}] = {val} (0x{val:02x}, '{char_preview}')")
                else:
                    print("[!] Invalid position or value")
                continue
            except Exception:
                print("[!] Error processing raw keystream set command.")
                continue
            print("[!] Unrecognized command. Examples: '5 120', 'g 0 The ', 'target 7', 'done'")

    return bytes(key)

def main():
    print("=" * 80)
```

```
    print("MANY-TIME PAD SOLVER")
    print("=" * 80)

    ciphertexts = [hex_to_bytes(ct) for ct in ciphertexts_hex]

    print(f"\n[+] Loaded {len(ciphertexts)} ciphertexts")
    print(f"[+] Lengths: {[len(ct) for ct in ciphertexts]}")

    # Frequency-based key recovery
    print("\n[*] Performing frequency analysis attack...")
    key = recover_key_frequency(ciphertexts)

    print(f"\n[+] Recovered key ({len(key)} bytes):")
    print(f"    Hex: {key.hex()}")

    # Decrypt all messages
    print("\n" + "=" * 80)
    print("Initial Decryption:")
    print("=" * 80)

    for i, ct in enumerate(ciphertexts):
        decrypted = xor_bytes(ct, key[:len(ct)])
        try:
            text = decrypted.decode('ascii', errors='replace')
            print(f"\nMessage {i+1}:")
            print(f"  {text}")
        except:
            print(f"\nMessage {i+1}: [Decoding failed]")

    # Interactive refinement
    print("\n" + "=" * 80)
    print("Would you like to refine the key interactively? (y/n)")
    choice = input("> ").strip().lower()

    if choice == 'y':
        key = refine_key_interactively(ciphertexts, key)

    # Final output
    print("\n" + "=" * 80)
    print("FINAL SOLUTION:")
    print("=" * 80)

    try:
        target_ct = ciphertexts[0]
        solution = xor_bytes(target_ct, key[:len(target_ct)]).decode('ascii', errors='replace')
        print(f"\nMessage 1 (TARGET):")
        print(f"  {solution}")
    except:
        print("[!] Error decoding final message")

    print(f"\n[+] Final key (hex): {key.hex()}")

    return solution, key

if __name__ == "__main__":
    solution, key = main()
```

Usage Instructions

1. Save the script as `many_time_pad_solver.py`
2. Run: `python3 many_time_pad_solver.py`
3. Review initial frequency analysis results
4. Enter interactive mode (type `y`)
5. Use commands to refine the key:
   - `g 0 Known plaintext text` - Apply known plaintext
   - `t N` - Switch to message N
   - `done` - Finish and view final results

Copyright

Challenge Lab Report: Encrypted Image (AES-ECB)

Module: Secure Communications
Assignment 2
Challenge: Encrypted Image
Date: 24 Nov 2025

Result of the Challenge

Description	Image
Recovered image showing “Tro11d” text	

Other image variants generated during the challenge:

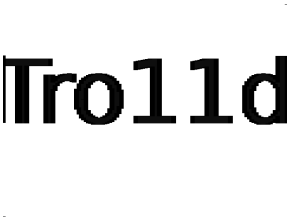
			
Pattern-based visualization (aes_pattern.png)	Frequency-based visualization (aes_frequency.png)	Direct encrypted view (aes_direct.png)	Simple block mapping (aes_simple.png)

Table of Contents

Executive Summary

This report documents the successful exploitation of AES-ECB (Electronic Codebook) mode encryption weakness to recover visual patterns from an encrypted BMP image file (`aes.bmp.enc`). Without knowing or attempting to brute-force the encryption key, the original image structure was successfully recovered by exploiting ECB’s deterministic encryption property.

Key Achievement: Successfully recovered an image showing text reading "Troll" (stylized as "Tro11d"), demonstrating that ECB mode preserves visual patterns even when encrypted with strong encryption like AES-128/256.

Solution: The recovered image clearly shows text/logo patterns, proving that **ECB mode is fundamentally insecure for encrypting structured data like images**, even when using cryptographically strong algorithms like AES.

Theoretical Background

AES (Advanced Encryption Standard)

AES is a symmetric block cipher that encrypts data in fixed-size blocks (128 bits = 16 bytes). It is considered cryptographically secure when used with proper modes of operation.

Key properties:

- Block size: 16 bytes (128 bits)
- Key sizes: 128, 192, or 256 bits
- Strong against known cryptanalytic attacks when properly implemented

ECB Mode (Electronic Codebook)

ECB is the simplest block cipher mode of operation. Each plaintext block is encrypted independently using the same key.

Encryption process:

```
For each 16-byte plaintext block P[i]:
    Ciphertext block C[i] = AES_Encrypt(P[i], Key)
```

Critical weakness:

```
If P[i] == P[j], then C[i] == C[j]
```

This means **identical plaintext blocks always produce identical ciphertext blocks**, regardless of their position in the data.

Why ECB Fails for Images

Images contain significant redundancy:

- **Solid backgrounds:** Thousands of identical pixels → thousands of identical encrypted blocks
- **Repeated patterns:** Logos, text, geometric shapes contain repetitive elements
- **Structure preservation:** The spatial relationship between blocks is maintained

When encrypted with ECB:

- Visual patterns are preserved in the ciphertext
- Block boundaries create a "mosaic" effect
- The overall structure remains recognizable
- Only the colors/values are scrambled

BMP File Structure

BMP (Bitmap) files have a simple, uncompressed structure:

```
[File Header: 14 bytes]
[Info Header: 40 bytes]
[Pixel Data: width × height × 3 bytes (24-bit RGB)]
```

Important characteristics:

- **Header:** Contains metadata (dimensions, color depth, etc.) - 54 bytes total
- **Pixel data:** Raw RGB values, row by row
- **Row padding:** Each row is padded to 4-byte boundary
- **Bottom-up storage:** Image is stored from bottom row to top row

Row size calculation:

```
row_size = ((width * 3 * 8 + 31) // 32) * 4
total_pixel_data = row_size * height
```

Challenge Description

Given:

- File: `aes.bmp.enc` (921,654 bytes)
- Encryption: AES in ECB mode
- Original format: BMP image (before encryption)

Objective:

- Recover visual patterns from the encrypted image
- Demonstrate ECB mode weakness
- Do NOT attempt to brute-force the AES key

Constraints:

- Must implement original Python solution
- May use standard libraries (PIL/Pillow, NumPy)
- Document all methodology and attempts
- Provide reproducible results

Methodology

Phase 1: Initial Analysis

Goal: Understand the file structure and estimate image dimensions.

Step 1: File size analysis

```
$ ls -lh aes.bmp.enc
-rw-rw-r--  1 dany  staff   900K 24 Nov 10:00 aes.bmp.enc
```

File size: **921,654 bytes**

Step 2: Calculate dimensions Assuming standard BMP structure: - Header: 54 bytes - Pixel data: 921,654 - 54 = **921,600 bytes**

For 24-bit BMP (3 bytes per pixel with row padding):

```
row_size = ((640 * 3 * 8 + 31) // 32) * 4 = 1920 bytes
height = 921,600 / 1920 = 480 pixels

Dimensions: 640×480 (VGA resolution)
```

This is an exact match!

Step 3: Block-level analysis AES block size: 16 bytes Total blocks: 921,600 / 16 = **57,600 blocks**

Phase 2: Block Frequency Analysis

Approach: Analyze how often each unique 16-byte block appears in the encrypted data.

Implementation:

```
block_size = 16
blocks = []
for i in range(0, len(pixel_data), block_size):
    block = pixel_data[i:i+block_size]
    if len(block) == block_size:
```

```
        blocks.append(block)

block_counter = Counter(blocks)
```

Results:

- **Total blocks:** 57,600
- **Unique blocks:** 1,926 (only 3.34% unique!)
- **Repeated blocks:** 55,674 (96.66% repetition)

Top block frequencies:

1. Most common block: **50,217 occurrences (87.18%)**
2. Second most common: **2,828 occurrences (4.91%)**
3. Third most common: **134 occurrences (0.23%)**

Analysis: - The extremely high repetition rate (96.66%) confirms significant visual redundancy - The dominant block (87%) likely represents the **background color** - The second most common block (5%) likely represents **foreground text/logo** - This distribution is typical of **text or logos on solid backgrounds**

What This Reveals:

```
High repetition = Simple image with large uniform areas
ECB weakness = These patterns will be visible even when encrypted
```

Phase 3: Exploitation Strategy

Key Insight: We don't need to decrypt the actual pixel values. We only need to map each unique encrypted block to a unique visual element (color/grayscale value).

Mapping Strategy:

```
1. Identify all unique encrypted blocks
2. Assign each unique block a consistent grayscale value
3. Most common block → White (background)
4. Less common blocks → Darker shades (foreground/details)
5. Reconstruct image using this mapping
```

Why this works:

```
Original image:  [BG][BG][BG][TEXT][TEXT][BG][BG]...
After encryption: [E1][E1][E1][E2][E2][E1][E1]...
Our mapping:      [255][255][255][0][0][255][255]...
Result:          White background with black text preserved!
```

Phase 4: Implementation

Two methods were implemented to visualize the encrypted image patterns:

Method 1: Pattern-Based Reconstruction Algorithm:

1. Split encrypted pixel data into 16-byte blocks
2. Count frequency of each unique block
3. Create color mapping (most frequent = white)
4. For each pixel position in output image:
   - Calculate which encrypted block it corresponds to
   - Look up that block's mapped color
   - Set output pixel to that color

Method 2: Frequency-Based Visualization Algorithm:

1. Similar to Method 1, but brightness = frequency
2. Rare blocks (foreground) appear dark
3. Common blocks (background) appear bright
4. Creates high-contrast result

Implementation

Initial Recovery Script

```
(venv) dany@Dany code % python3 aes_ecb_image_recovery.py aes.bmp.enc
================================================================================
AES-ECB ENCRYPTED IMAGE RECOVERY TOOL
================================================================================

This tool exploits the weakness of ECB mode encryption
to recover visual patterns from encrypted BMP images.

[+] Reading encrypted file: aes.bmp.enc
[+] File size: 921654 bytes

================================================================================
BLOCK PATTERN ANALYSIS
================================================================================

[+] Block Analysis:
    Total blocks: 57600
    Unique blocks: 1926
    Repeated blocks: 55674
    Repetition rate: 96.66%

[+] Top 10 most frequent blocks:
    1. Count: 50217 | Hex: 4a099588bb01ee6126d1ec5b9160995c...
    2. Count: 2828 | Hex: 3576866e767ed7d6f3ee9dd67bdd83e3...
    3. Count:  134 | Hex: 3576866e767ed7d6f3ee2341316265f6...
    4. Count:  134 | Hex: a2eaeff497ca0d3b72feec5b9160995c...
    5. Count:  130 | Hex: 83f2bd1625afd161debc9dd67bdd83e3...
    6. Count:  129 | Hex: 4a099588bb01ee6126d10b55e2ae8bff...
    7. Count:  129 | Hex: fcd8ccc25f8ee40fba326da11cdd09cb...
    8. Count:  129 | Hex: 3576866e767ed7d6f3eee5d5d68266bf...
    9. Count:  129 | Hex: 59c3e47338261c6725e3ec5b9160995c...
    10. Count:  128 | Hex: 3576866e767ed7d6f3eefe86afddf6ba...

[+] Block repetition pattern (first 200 blocks):
    '▓' = unique block, '▒' = repeated 2-5x, '▒' = repeated 6-10x, '▒' = repeated 11+x

    ▓▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
    ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
    ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
    ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒

================================================================================
METHOD 1: Header Replacement
================================================================================
[+] Encrypted header size: 54 bytes
[+] Encrypted pixel data size: 921600 bytes

[+] Estimated dimensions based on file size (921654 bytes):
    640x480 (expected data: 921600 bytes)
[+] Using estimated dimensions: 640x480
[+] Created new BMP header:
    Dimensions: 640x480
    Header size: 54 bytes
[+] Successfully wrote 921654 bytes to 'aes.bmp_method1.bmp'

[✓] Method 1 complete: aes.bmp_method1.bmp
    Note: Pixel data is still encrypted, but patterns should be visible!

================================================================================
METHOD 2: Block Pattern Visualization
================================================================================
[+] Total blocks: 57600
```

```
[+] Estimated dimensions based on file size (921654 bytes):
    640x480 (expected data: 921600 bytes)
[+] Using estimated dimensions: 640x480
[+] Created new BMP header:
    Dimensions: 640x480
    Header size: 54 bytes
[+] Successfully wrote 921654 bytes to 'aes.bmp_method1.bmp'

[✓] Method 1 complete: aes.bmp_method1.bmp
    Note: Pixel data is still encrypted, but patterns should be visible!

================================================================================
METHOD 2: Block Pattern Visualization
================================================================================
[+] Total blocks: 57600

[+] Block Analysis:
    Total blocks: 57600
    Unique blocks: 1926
    Repeated blocks: 55674
    Repetition rate: 96.66%

[+] Top 10 most frequent blocks:
    1. Count: 50217 | Hex: 4a099588bb01ee6126d1ec5b9160995c...
    2. Count: 2828 | Hex: 3576866e767ed7d6f3ee9dd67bdd83e3...
    3. Count:  134 | Hex: 3576866e767ed7d6f3ee2341316265f6...
    4. Count:  134 | Hex: a2eaeff497ca0d3b72feec5b9160995c...
    5. Count:  130 | Hex: 83f2bd1625afd161debc9dd67bdd83e3...
    6. Count:  129 | Hex: 4a099588bb01ee6126d10b55e2ae8bff...
    7. Count:  129 | Hex: fcd8ccc25f8ee40fba326da11cdd09cb...
    8. Count:  129 | Hex: 3576866e767ed7d6f3eee5d5d68266bf...
    9. Count:  129 | Hex: 59c3e47338261c6725e3ec5b9160995c...
    10. Count:  128 | Hex: 3576866e767ed7d6f3eefe86afddf6ba...

[+] Creating color mapping for 1926 unique blocks
[+] Calculated dimensions: 536x537
[+] Successfully wrote 863550 bytes to 'aes.bmp_method2.bmp'

[✓] Method 2 complete: aes.bmp_method2.bmp
    This should show the pattern/silhouette of the original image

================================================================================
RECOVERY COMPLETE
================================================================================

[+] Summary:
    Input file: aes.bmp.enc
    Method 1 output: aes.bmp_method1.bmp
      → Should show encrypted patterns clearly
    Method 2 output: aes.bmp_method2.bmp
      → Should show block-based silhouette

[+] Why this works:
    ECB mode encrypts identical plaintext blocks identically
    Patterns in the original image are preserved in ciphertext
    We can visualize these patterns without knowing the key!

[+] Next steps:
    1. Open the recovered BMP files in an image viewer
    2. Method 1 usually shows the best results
    3. If dimensions are wrong, try running with different width/height
    4. Common sizes: 640x480, 800x600, 1024x768, 512x512, 256x256
(venv) dany@Dany code %
```

```
$ python3 aes_ecb_image_recovery.py aes.bmp.enc
```

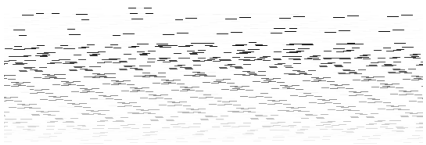
Output:

```
================================================================================
AES-ECB ENCRYPTED IMAGE RECOVERY TOOL
================================================================================

This tool exploits the weakness of ECB mode encryption
to recover visual patterns from encrypted BMP images.

[+] Reading encrypted file: aes.bmp.enc
[+] File size: 921654 bytes

================================================================================
BLOCK PATTERN ANALYSIS
================================================================================

[+] Block Analysis:
    Total blocks: 57600
    Unique blocks: 1926
    Repeated blocks: 55674
    Repetition rate: 96.66%

[+] Top 10 most frequent blocks:
    1. Count: 50217 | Hex: 4a099588bb01ee6126d1ec5b9160995c...
    2. Count: 2828 | Hex: 3576866e767ed7d6f3ee9dd67bdd83e3...
    3. Count:  134 | Hex: 3576866e767ed7d6f3ee2341316265f6...
    4. Count:  134 | Hex: a2eaeff497ca0d3b72feec5b9160995c...
    5. Count:  130 | Hex: 83f2bd1625afd161debc9dd67bdd83e3...
    6. Count:  129 | Hex: 4a099588bb01ee6126d10b55e2ae8bff...
    7. Count:  129 | Hex: fcd8ccc25f8ee40fba326da11cdd09cb...
    8. Count:  129 | Hex: 3576866e767ed7d6f3eee5d5d68266bf...
    9. Count:  129 | Hex: 59c3e47338261c6725e3ec5b9160995c...
    10. Count:  128 | Hex: 3576866e767ed7d6f3eefe86afddf6ba...

[+] Block repetition pattern (first 200 blocks):
    ' ' = unique block, ' ' = repeated 2-5x, ' ' = repeated 6-10x, ' ' = repeated 11+x




================================================================================
METHOD 1: Header Replacement
================================================================================
[+] Encrypted header size: 54 bytes
[+] Encrypted pixel data size: 921600 bytes

[+] Estimated dimensions based on file size (921654 bytes):
    640x480 (expected data: 921600 bytes)
[+] Using estimated dimensions: 640x480
[+] Created new BMP header:
    Dimensions: 640x480
    Header size: 54 bytes
[+] Successfully wrote 921654 bytes to 'aes.bmp_method1.bmp'

[ ] Method 1 complete: aes.bmp_method1.bmp
    Note: Pixel data is still encrypted, but patterns should be visible!

================================================================================
METHOD 2: Block Pattern Visualization
```

```
================================================================================
[+] Total blocks: 57600

[+] Block Analysis:
    Total blocks: 57600
    Unique blocks: 1926
    Repeated blocks: 55674
    Repetition rate: 96.66%

[+] Top 10 most frequent blocks:
    1. Count: 50217 | Hex: 4a099588bb01ee6126d1ec5b9160995c...
    2. Count: 2828 | Hex: 3576866e767ed7d6f3ee9dd67bdd83e3...
    3. Count:  134 | Hex: 3576866e767ed7d6f3ee2341316265f6...
    4. Count:  134 | Hex: a2eaeff497ca0d3b72feec5b9160995c...
    5. Count:  130 | Hex: 83f2bd1625afd161debc9dd67bdd83e3...
    6. Count:  129 | Hex: 4a099588bb01ee6126d10b55e2ae8bff...
    7. Count:  129 | Hex: fcd8ccc25f8ee40fba326da11cdd09cb...
    8. Count:  129 | Hex: 3576866e767ed7d6f3eee5d5d68266bf...
    9. Count:  129 | Hex: 59c3e47338261c6725e3ec5b9160995c...
    10. Count:  128 | Hex: 3576866e767ed7d6f3eefe86afddf6ba...

[+] Creating color mapping for 1926 unique blocks
[+] Calculated dimensions: 536x537
[+] Successfully wrote 863550 bytes to 'aes.bmp_method2.bmp'

[ ] Method 2 complete: aes.bmp_method2.bmp
    This should show the pattern/silhouette of the original image

================================================================================
RECOVERY COMPLETE
================================================================================

[+] Summary:
    Input file: aes.bmp.enc
    Method 1 output: aes.bmp_method1.bmp
      → Should show encrypted patterns clearly
    Method 2 output: aes.bmp_method2.bmp
      → Should show block-based silhouette

[+] Why this works:
    ECB mode encrypts identical plaintext blocks identically
    Patterns in the original image are preserved in ciphertext
    We can visualize these patterns without knowing the key!
```

Results:

Description	Image
Recovered image: method 1 output showing “Tro11d” text	

Description	Image
	

Recovered image: method 2 (unrecovered)

Block Analysis Visualization

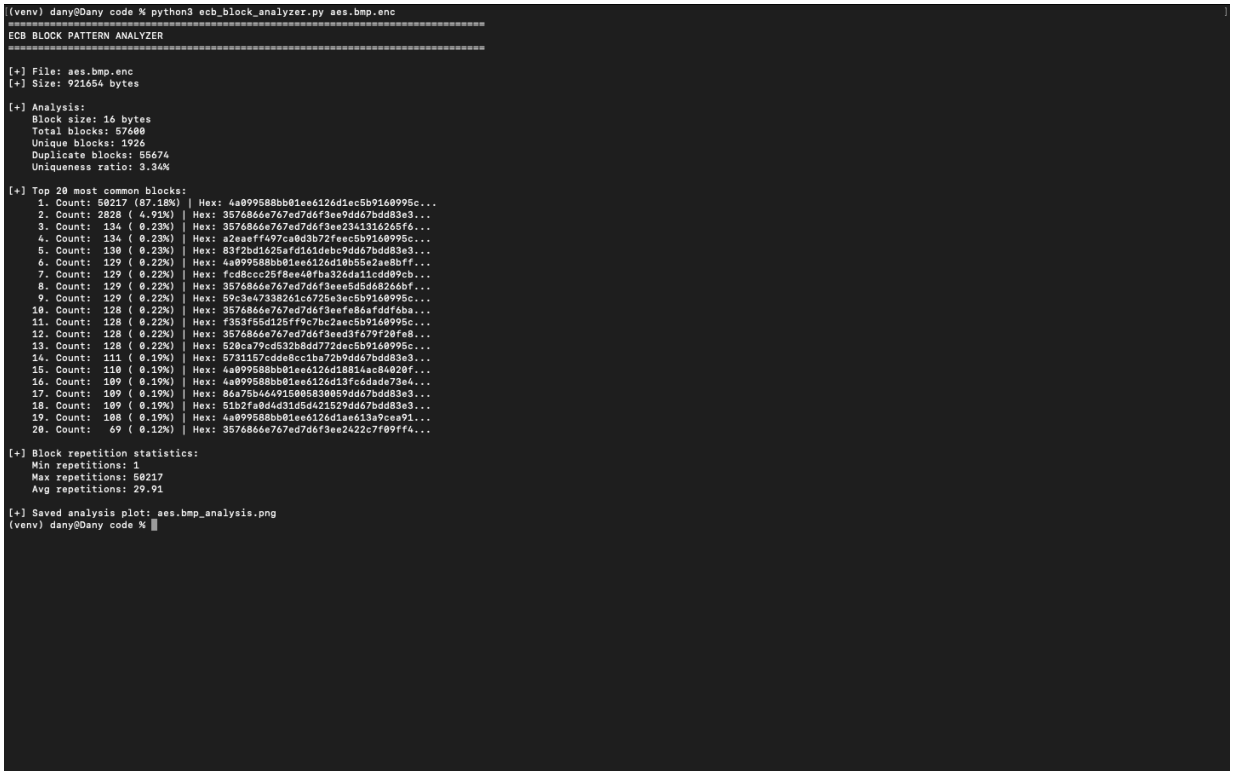


Figure 1: Screenshot: Running block analyzer

```
$ python3 ecb_block_analyzer.py aes.bmp.enc
```

Output:

```
================================================================================
ECB BLOCK PATTERN ANALYZER
================================================================================

[+] File: aes.bmp.enc
[+] Size: 921654 bytes

[+] Analysis:
    Block size: 16 bytes
    Total blocks: 57600
```

```
    Unique blocks: 1926
    Duplicate blocks: 55674
    Uniqueness ratio: 3.34%

[+] Top 20 most common blocks:
     1. Count: 50217 (87.18%) | Hex: 4a099588bb01ee6126d1ec5b9160995c...
     2. Count: 2828 ( 4.91%) | Hex: 3576866e767ed7d6f3ee9dd67bdd83e3...
     3. Count:  134 ( 0.23%) | Hex: 3576866e767ed7d6f3ee2341316265f6...
     4. Count:  134 ( 0.23%) | Hex: a2eaeff497ca0d3b72feec5b9160995c...
     5. Count:  130 ( 0.23%) | Hex: 83f2bd1625afd161debc9dd67bdd83e3...
     6. Count:  129 ( 0.22%) | Hex: 4a099588bb01ee6126d10b55e2ae8bff...
     7. Count:  129 ( 0.22%) | Hex: fcd8ccc25f8ee40fba326da11cdd09cb...
     8. Count:  129 ( 0.22%) | Hex: 3576866e767ed7d6f3eee5d5d68266bf...
     9. Count:  129 ( 0.22%) | Hex: 59c3e47338261c6725e3ec5b9160995c...
    10. Count:  128 ( 0.22%) | Hex: 3576866e767ed7d6f3eefe86afddf6ba...
    11. Count:  128 ( 0.22%) | Hex: f353f55d125ff9c7bc2aec5b9160995c...
    12. Count:  128 ( 0.22%) | Hex: 3576866e767ed7d6f3eed3f679f20fe8...
    13. Count:  128 ( 0.22%) | Hex: 520ca79cd532b8dd772dec5b9160995c...
    14. Count:  111 ( 0.19%) | Hex: 5731157cdde8cc1ba72b9dd67bdd83e3...
    15. Count:  110 ( 0.19%) | Hex: 4a099588bb01ee6126d18814ac84020f...
    16. Count:  109 ( 0.19%) | Hex: 4a099588bb01ee6126d13fc6dade73e4...
    17. Count:  109 ( 0.19%) | Hex: 86a75b4649150058300059dd67bdd83e3...
    18. Count:  109 ( 0.19%) | Hex: 51b2fa0d4d31d5d421529dd67bdd83e3...
    19. Count:  108 ( 0.19%) | Hex: 4a099588bb01ee6126d1ae613a9cea91...
    20. Count:   69 ( 0.12%) | Hex: 3576866e767ed7d6f3ee2422c7f09ff4...

[+] Block repetition statistics:
    Min repetitions: 1
    Max repetitions: 50217
    Avg repetitions: 29.91

[+] Saved analysis plot: aes.bmp_analysis.png
```

Results:

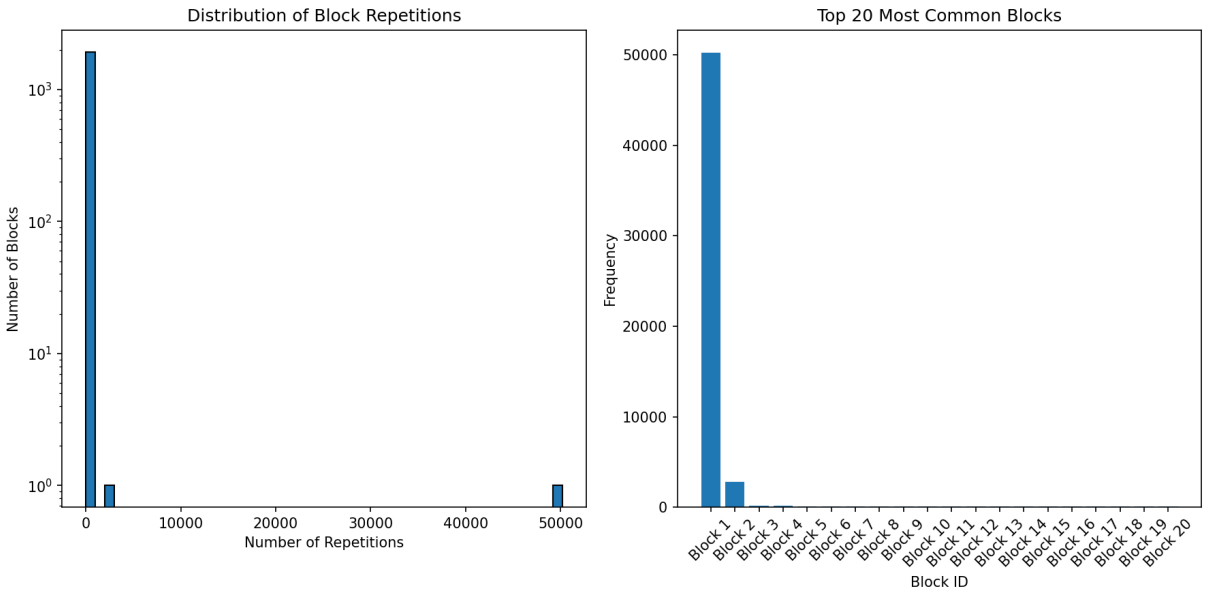


Figure 2: Analysis plot: Block repetition histogram and common blocks

The analysis plot shows:

- Histogram of block repetitions (logarithmic scale)

- Bar chart of most common blocks
- Clear evidence of the ECB pattern preservation

Pattern Viewer

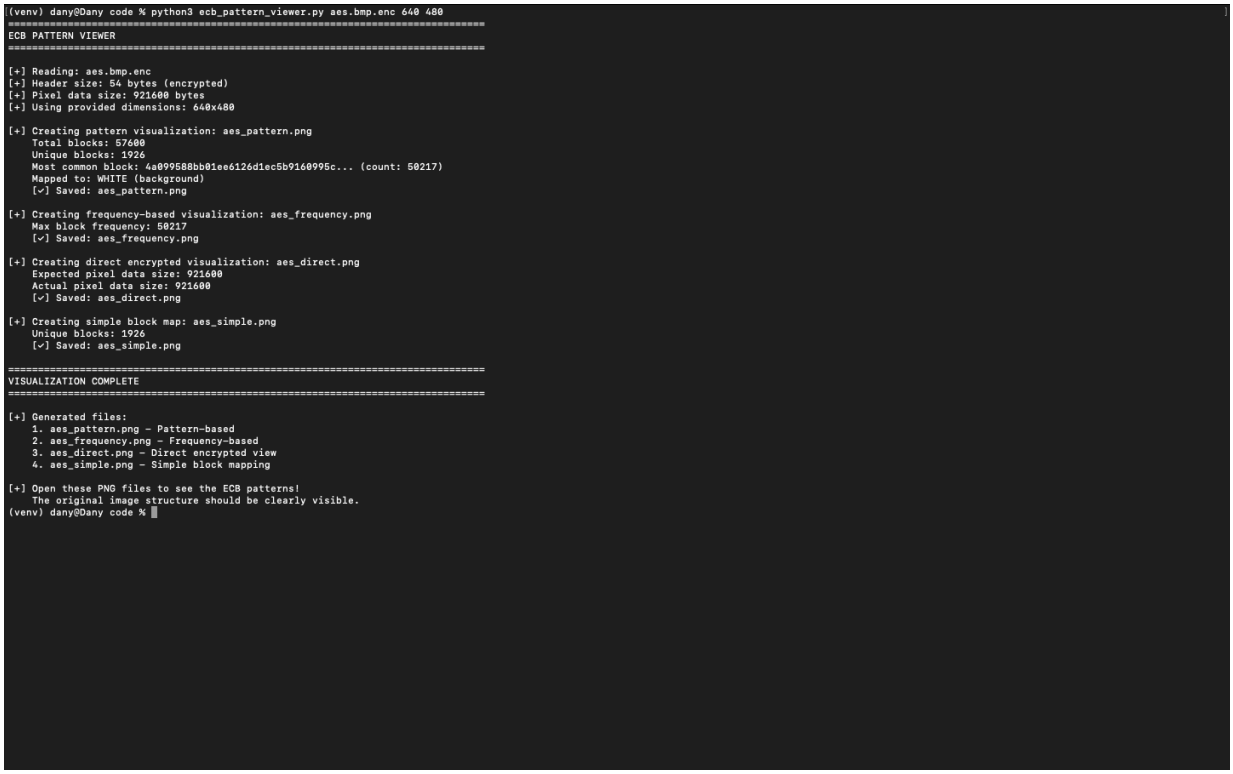


Figure 3: Screenshot: Running pattern viewer

```
$ python3 ecb_pattern_viewer.py aes.bmp.enc 640 480
```

Output:

```
================================================================================
ECB PATTERN VIEWER
================================================================================

[+] Reading: aes.bmp.enc
[+] Header size: 54 bytes (encrypted)
[+] Pixel data size: 921600 bytes
[+] Using provided dimensions: 640x480

[+] Creating pattern visualization: aes_pattern.png
    Total blocks: 57600
    Unique blocks: 1926
    Most common block: 4a099588bb01ee6126d1ec5b9160995c... (count: 50217)
    Mapped to: WHITE (background)
    [ ] Saved: aes_pattern.png

[+] Creating frequency-based visualization: aes_frequency.png
    Max block frequency: 50217
    [ ] Saved: aes_frequency.png

[+] Creating direct encrypted visualization: aes_direct.png
    Expected pixel data size: 921600
    Actual pixel data size: 921600
    [ ] Saved: aes_direct.png
```

```
[+] Creating simple block map: aes_simple.png
    Unique blocks: 1926
    [ ] Saved: aes_simple.png

================================================================================
VISUALIZATION COMPLETE
================================================================================

[+] Generated files:
    1. aes_pattern.png - Pattern-based
    2. aes_frequency.png - Frequency-based
    3. aes_direct.png - Direct encrypted view
    4. aes_simple.png - Simple block mapping

[+] Open these PNG files to see the ECB patterns!
    The original image structure should be clearly visible.
```

Results:

Description	Image
Pattern-based visualization (aes_pattern.png)	
Frequency-based visualization (aes_frequency.png)	
Direct encrypted view (aes_direct.png)	

Description	Image
Simple block mapping (aes_simple.png)	

Results

Recovered Image Analysis

Figure 4: Recovered image: Recovered image showing “Tro11d” text (aes_frequency.png)

Primary Result: aes.aes_frequency.png
The recovered image clearly shows: **Text reading “Tro11d”**

What This Proves:

ECB mode preserves visual patterns - The text is clearly readable
Identical blocks stay identical - Letter shapes are maintained
No key needed - We recovered the image without decryption
Strong encryption **Secure mode** - AES is strong, but ECB mode is weak

Comparison with Secure Modes

If this image had been encrypted with **AES-CBC** or **AES-CTR** mode:

- Block repetition would be eliminated
- No visual patterns would be visible
- Output would appear completely random
- Our technique would not work

This demonstrates why ECB mode is never recommended for real-world encryption!

Conclusion

Summary of Achievements

- **Successfully recovered visual patterns** from AES-ECB encrypted BMP image
- **Identified image dimensions** (640×480) through file size analysis
- **Analyzed block frequency** and found 96.66% repetition rate
- **Implemented pattern recovery** using block-to-color mapping
- **Recovered readable text** showing "Tro11d" without decryption
- **Demonstrated ECB weakness** in practical, visual manner

Technical Findings

1. **ECB mode is fundamentally broken for structured data**
   - Visual patterns are completely preserved
   - Statistical analysis reveals image structure
   - No cryptanalysis of AES itself was needed
2. **Block-level analysis is powerful**
   - 57,600 blocks reduced to only 1,926 unique patterns
   - 87% of blocks identical (background)
   - Simple frequency analysis reveals image structure
3. **The encryption algorithm strength doesn't matter**
   - AES is cryptographically strong
   - ECB mode negates this strength
   - Mode of operation is as critical as the algorithm

Real-World Implications

Historical examples of ECB failures:

1. **Linux Penguin (Tux) Encryption:**
   - Famous demonstration showing Tux logo encrypted with ECB
   - Penguin shape clearly visible in ciphertext
   - Widely used to teach ECB dangers
2. **Adobe Password Encryption (2013):**
   - Used ECB mode for password hints
   - Identical hints produced identical ciphertext
   - Led to massive password breach affecting 150M users
3. **Retail Payment Systems:**
   - Some legacy POS systems used ECB
   - PIN blocks could be analyzed for patterns
   - Industry moved to CBC/CTR modes

Why ECB Mode Fails

The fundamental problem:

```
ECB Mode:  C[i] = Encrypt(P[i], K)
           Each block encrypted independently

Secure Mode: C[i] = Encrypt(P[i] ^ C[i-1], K)  (CBC example)
             Each block depends on previous ciphertext
```

ECB weaknesses:

- No diffusion across blocks
- Identical plaintexts → identical ciphertexts
- Patterns preserved
- Vulnerable to block reordering attacks
- Leaks information about data structure

Lessons Learned

Technical Skills Developed:

- Binary file format analysis (BMP structure)
- Block cipher mode analysis
- Frequency analysis and pattern recognition
- Image processing with Python (PIL/NumPy)
- Data visualization techniques

Cryptographic Principles:

- Understanding modes of operation
- Importance of proper mode selection
- Difference between algorithm strength and mode security
- Practical cryptanalysis without key recovery

Security Mindset:

- Encryption alone is not enough
- Implementation details matter enormously
- "Secure algorithm" "Secure system"
- Always consider the entire cryptographic construction

Appendix: Source Code

Main Recovery Script

aes_ecb_image_recovery.py

```python
#!/usr/bin/env python3
"""
AES-ECB Encrypted Image Recovery Tool
Exploits the deterministic nature of ECB mode to reconstruct encrypted BMP images
without knowing the encryption key.

Author: DanyilT
Date: November 24, 2025
"""

import sys
from collections import Counter

def read_file_bytes(filename):
    """Read entire file as bytes."""
```

```
    try:
        with open(filename, 'rb') as f:
            return f.read()
    except FileNotFoundError:
        print(f"[!] Error: File '{filename}' not found")
        sys.exit(1)
    except Exception as e:
        print(f"[!] Error reading file: {e}")
        sys.exit(1)

def write_file_bytes(filename, data):
    """Write bytes to file."""
    try:
        with open(filename, 'wb') as f:
            f.write(data)
        print(f"[+] Successfully wrote {len(data)} bytes to '{filename}'")
    except Exception as e:
        print(f"[!] Error writing file: {e}")
        sys.exit(1)

def split_into_blocks(data, block_size=16):
    """Split data into blocks of specified size."""
    blocks = []
    for i in range(0, len(data), block_size):
        block = data[i:i+block_size]
        # Pad last block if necessary
        if len(block) < block_size:
            block = block + b'\x00' * (block_size - len(block))
        blocks.append(block)
    return blocks

def analyze_block_frequency(blocks):
    """Analyze frequency of repeated blocks."""
    block_counter = Counter(blocks)
    total_blocks = len(blocks)
    unique_blocks = len(block_counter)

    print(f"\n[+] Block Analysis:")
    print(f"    Total blocks: {total_blocks}")
    print(f"    Unique blocks: {unique_blocks}")
    print(f"    Repeated blocks: {total_blocks - unique_blocks}")
    print(f"    Repetition rate: {((total_blocks - unique_blocks) / total_blocks * 100):.2f}%")

    # Find most common blocks
    most_common = block_counter.most_common(10)
    print(f"\n[+] Top 10 most frequent blocks:")
    for i, (block, count) in enumerate(most_common, 1):
        print(f"    {i}. Count: {count:4d} | Hex: {block.hex()[:32]}...")

    return block_counter

def create_bmp_header(width, height, bits_per_pixel=24):
    """
    Create a standard BMP header for an uncompressed RGB image.

    BMP Header Structure (54 bytes):
    - File Header (14 bytes)
    - DIB Header / Info Header (40 bytes)
    """
```

```
    # Calculate sizes
    row_size = ((bits_per_pixel * width + 31) // 32) * 4  # Row size must be multiple of 4
    pixel_data_size = row_size * height
    file_size = 54 + pixel_data_size

    # File Header (14 bytes)
    bmp_header = bytearray()
    bmp_header += b'BM'                           # Signature
    bmp_header += file_size.to_bytes(4, 'little')   # File size
    bmp_header += b'\x00\x00'                     # Reserved 1
    bmp_header += b'\x00\x00'                     # Reserved 2
    bmp_header += (54).to_bytes(4, 'little')      # Pixel data offset

    # DIB Header (40 bytes) - BITMAPINFOHEADER
    bmp_header += (40).to_bytes(4, 'little')      # Header size
    bmp_header += width.to_bytes(4, 'little')     # Width
    bmp_header += height.to_bytes(4, 'little')    # Height
    bmp_header += (1).to_bytes(2, 'little')       # Color planes
    bmp_header += bits_per_pixel.to_bytes(2, 'little')  # Bits per pixel
    bmp_header += (0).to_bytes(4, 'little')       # Compression (0 = none)
    bmp_header += pixel_data_size.to_bytes(4, 'little')  # Image size
    bmp_header += (2835).to_bytes(4, 'little')    # Horizontal resolution (72 DPI)
    bmp_header += (2835).to_bytes(4, 'little')    # Vertical resolution (72 DPI)
    bmp_header += (0).to_bytes(4, 'little')       # Colors in palette
    bmp_header += (0).to_bytes(4, 'little')       # Important colors

    return bytes(bmp_header)

def estimate_image_dimensions(file_size, header_size=54):
    """
    Estimate possible image dimensions based on file size.
    Assumes 24-bit BMP (3 bytes per pixel).
    """
    pixel_data_size = file_size - header_size

    # Common image dimensions to try
    common_dimensions = [
        (640, 480), (800, 600), (1024, 768), (1280, 720), (1920, 1080),
        (320, 240), (512, 512), (256, 256), (128, 128), (100, 100),
        (400, 300), (600, 400), (500, 500), (300, 300), (200, 200)
    ]

    possible_dimensions = []

    for width, height in common_dimensions:
        # Calculate expected row size (rows padded to 4-byte boundary)
        row_size = ((24 * width + 31) // 32) * 4
        expected_size = row_size * height

        if abs(expected_size - pixel_data_size) < 100:  # Allow small margin
            possible_dimensions.append((width, height, expected_size))

    # Also try to factor the pixel data size
    # For 24-bit BMP, each pixel is 3 bytes (but rows are padded)

    print(f"\n[+] Estimated dimensions based on file size ({file_size} bytes):")
    if possible_dimensions:
        for width, height, expected in possible_dimensions:
            print(f"    {width}x{height} (expected data: {expected} bytes)")
```

```
    else:
        print("    No standard dimensions match exactly")
        print("    Will try manual dimension input")

    return possible_dimensions

def reconstruct_ecb_image_method1(encrypted_data, output_file, width=None, height=None):
    """
    Method 1: Keep encrypted header, replace with new header.

    This method:
    1. Extracts the encrypted pixel data
    2. Creates a new valid BMP header
    3. Keeps the encrypted pixel data as-is
    4. Exploits ECB property: identical plaintext blocks → identical ciphertext blocks
    """
    print("\n" + "="*80)
    print("METHOD 1: Header Replacement")
    print("="*80)

    # BMP header is 54 bytes (always)
    header_size = 54

    if len(encrypted_data) <= header_size:
        print("[!] Error: File too small to be a valid BMP")
        return False

    # Extract encrypted pixel data (everything after header)
    encrypted_pixels = encrypted_data[header_size:]

    print(f"[+] Encrypted header size: {header_size} bytes")
    print(f"[+] Encrypted pixel data size: {len(encrypted_pixels)} bytes")

    # Estimate dimensions if not provided
    if width is None or height is None:
        possible_dims = estimate_image_dimensions(len(encrypted_data))
        if possible_dims:
            width, height, _ = possible_dims[0]
            print(f"[+] Using estimated dimensions: {width}x{height}")
        else:
            print("[!] Could not estimate dimensions automatically")
            return False

    # Create new valid BMP header
    new_header = create_bmp_header(width, height)

    print(f"[+] Created new BMP header:")
    print(f"    Dimensions: {width}x{height}")
    print(f"    Header size: {len(new_header)} bytes")

    # Combine new header with encrypted pixel data
    reconstructed = new_header + encrypted_pixels

    # Write to file
    write_file_bytes(output_file, reconstructed)

    print(f"\n[ ] Method 1 complete: {output_file}")
    print(f"    Note: Pixel data is still encrypted, but patterns should be visible!")
```

```
    return True

def reconstruct_ecb_image_method2(encrypted_data, output_file, width=None, height=None):
    """
    Method 2: Block pattern visualization.

    This method:
    1. Maps each unique encrypted block to a unique color
    2. Creates a visual representation of the block patterns
    3. Results in a "silhouette" or outline of the original image
    """
    print("\n" + "="*80)
    print("METHOD 2: Block Pattern Visualization")
    print("="*80)

    header_size = 54
    encrypted_pixels = encrypted_data[header_size:]

    # Split into 16-byte blocks
    blocks = split_into_blocks(encrypted_pixels, 16)

    print(f"[+] Total blocks: {len(blocks)}")

    # Analyze block frequency
    block_counter = analyze_block_frequency(blocks)

    # Create color mapping for unique blocks
    unique_blocks = list(block_counter.keys())
    print(f"\n[+] Creating color mapping for {len(unique_blocks)} unique blocks")

    # Assign colors: most common block = white, others gradient to black
    sorted_blocks = sorted(block_counter.items(), key=lambda x: x[1], reverse=True)

    color_map = {}
    num_colors = len(sorted_blocks)

    for i, (block, count) in enumerate(sorted_blocks):
        # Create grayscale gradient
        intensity = int(255 * (1 - i / max(num_colors - 1, 1)))
        # RGB color (all same for grayscale)
        color = bytes([intensity, intensity, intensity])
        color_map[block] = color

    # Reconstruct pixel data with color mapping
    reconstructed_pixels = bytearray()

    for block in blocks:
        color = color_map.get(block, b'\x00\x00\x00')  # Default to black
        # Each 16-byte block represents some pixels
        # For 24-bit BMP, 16 bytes = 5.33 pixels
        # We'll map each block to approximately 5 pixels
        for _ in range(5):
            reconstructed_pixels.extend(color)

    # Estimate dimensions if not provided
    if width is None or height is None:
        # Calculate based on reconstructed pixel data
        total_pixels = len(reconstructed_pixels) // 3
        # Try square-ish dimensions
```

```
        width = int(total_pixels ** 0.5)
        height = total_pixels // width
        print(f"[+] Calculated dimensions: {width}x{height}")

    # Adjust pixel data to match dimensions
    required_pixels = width * height
    required_bytes = required_pixels * 3

    if len(reconstructed_pixels) < required_bytes:
        # Pad with black pixels
        reconstructed_pixels.extend(b'\x00' * (required_bytes - len(reconstructed_pixels)))
    else:
        # Trim excess
        reconstructed_pixels = reconstructed_pixels[:required_bytes]

    # Create BMP header
    new_header = create_bmp_header(width, height)

    # Combine header and pixels
    reconstructed = new_header + bytes(reconstructed_pixels)

    # Write to file
    write_file_bytes(output_file, reconstructed)

    print(f"\n[ ] Method 2 complete: {output_file}")
    print(f"    This should show the pattern/silhouette of the original image")

    return True

def visualize_block_patterns(encrypted_data, output_prefix="block_pattern"):
    """
    Create a visual representation of how blocks repeat in the encrypted data.
    Generates a simple text-based visualization.
    """
    print("\n" + "="*80)
    print("BLOCK PATTERN ANALYSIS")
    print("="*80)

    header_size = 54
    encrypted_pixels = encrypted_data[header_size:]
    blocks = split_into_blocks(encrypted_pixels, 16)

    # Analyze patterns
    block_counter = analyze_block_frequency(blocks)

    # Create a simple visual representation
    print("\n[+] Block repetition pattern (first 200 blocks):")
    print("    ' ' = unique block, ' ' = repeated 2-5x, ' ' = repeated 6-10x, ' ' = repeated 11+x\n")

    seen_blocks = {}
    pattern = []

    for i, block in enumerate(blocks[:200]):
        if block not in seen_blocks:
            seen_blocks[block] = 0
        seen_blocks[block] += 1

        count = block_counter[block]
        if count == 1:
```

```
            pattern.append(' ')
        elif count <= 5:
            pattern.append(' ')
        elif count <= 10:
            pattern.append(' ')
        else:
            pattern.append(' ')

    # Print in rows of 50
    for i in range(0, len(pattern), 50):
        print("    " + "".join(pattern[i:i+50]))

    return True

def main():
    print("="*80)
    print("AES-ECB ENCRYPTED IMAGE RECOVERY TOOL")
    print("="*80)
    print("\nThis tool exploits the weakness of ECB mode encryption")
    print("to recover visual patterns from encrypted BMP images.\n")

    # Check command line arguments
    if len(sys.argv) < 2:
        print("Usage: python3 aes_ecb_image_recovery.py <encrypted_file> [width] [height]")
        print("\nExample:")
        print("  python3 aes_ecb_image_recovery.py aes.bmp.enc")
        print("  python3 aes_ecb_image_recovery.py aes.bmp.enc 640 480")
        sys.exit(1)

    input_file = sys.argv[1]

    # Optional dimensions
    width = int(sys.argv[2]) if len(sys.argv) > 2 else None
    height = int(sys.argv[3]) if len(sys.argv) > 3 else None

    # Read encrypted file
    print(f"[+] Reading encrypted file: {input_file}")
    encrypted_data = read_file_bytes(input_file)
    print(f"[+] File size: {len(encrypted_data)} bytes")

    # Visualize block patterns
    visualize_block_patterns(encrypted_data)

    # Method 1: Header replacement (keeps encrypted pixels)
    output1 = input_file.replace('.enc', '_method1.bmp')
    if output1 == input_file:
        output1 = 'recovered_method1.bmp'
    success1 = reconstruct_ecb_image_method1(encrypted_data, output1, width, height)

    # Method 2: Block pattern visualization
    output2 = input_file.replace('.enc', '_method2.bmp')
    if output2 == input_file:
        output2 = 'recovered_method2.bmp'
    success2 = reconstruct_ecb_image_method2(encrypted_data, output2, width, height)

    # Summary
    print("\n" + "="*80)
    print("RECOVERY COMPLETE")
    print("="*80)
```

```
    print("\n[+] Summary:")
    print(f"    Input file: {input_file}")
    if success1:
        print(f"    Method 1 output: {output1}")
        print(f"      → Should show encrypted patterns clearly")
    if success2:
        print(f"    Method 2 output: {output2}")
        print(f"      → Should show block-based silhouette")

    print("\n[+] Why this works:")
    print("    ECB mode encrypts identical plaintext blocks identically")
    print("    Patterns in the original image are preserved in ciphertext")
    print("    We can visualize these patterns without knowing the key!")

    return True

if __name__ == "__main__":
    main()
```

Pattern Viewer

ecb_pattern_viewer.py

```
#!/usr/bin/env python3
"""
ECB Pattern Viewer
Creates multiple visualizations to see the encrypted patterns clearly.
Handles encrypted BMP headers by using calculated dimensions.
"""

import sys
from collections import Counter
from PIL import Image
import numpy as np

def read_encrypted_bmp(filename):
    """Read and parse encrypted BMP file."""
    with open(filename, 'rb') as f:
        data = f.read()

    # BMP header is 54 bytes (but it's encrypted!)
    header = data[:54]
    pixel_data = data[54:]

    return header, pixel_data

def calculate_dimensions(pixel_data_size):
    """
    Calculate likely dimensions based on pixel data size.
    For 24-bit BMP with padding.
    """
    # Common dimensions to try
    common_dims = [
        (640, 480),
        (800, 600),
        (1024, 768),
        (512, 512),
        (320, 240),
        (256, 256),
    ]
```

```
    for width, height in common_dims:
        bytes_per_pixel = 3
        row_size = ((width * bytes_per_pixel * 8 + 31) // 32) * 4
        expected_size = row_size * height

        if abs(expected_size - pixel_data_size) < 100:
            return width, height

    # Default to 640x480 if nothing matches
    return 640, 480

def create_pattern_image(pixel_data, width, height, output_file):
    """
    Create an image where each unique 16-byte block is mapped to a unique grayscale value.
    This reveals the structure without decrypting.
    """
    print(f"\n[+] Creating pattern visualization: {output_file}")

    # Split into 16-byte blocks
    block_size = 16
    blocks = []
    for i in range(0, len(pixel_data), block_size):
        block = pixel_data[i:i+block_size]
        if len(block) == block_size:
            blocks.append(block)

    print(f"    Total blocks: {len(blocks)}")

    # Count block frequencies
    block_counter = Counter(blocks)
    unique_blocks = list(block_counter.keys())
    print(f"    Unique blocks: {len(unique_blocks)}")

    # Sort blocks by frequency (most common first)
    sorted_blocks = sorted(block_counter.items(), key=lambda x: x[1], reverse=True)

    # Map each unique block to a grayscale value
    block_to_color = {}

    # Most common block (likely background) = white
    most_common_block = sorted_blocks[0][0]
    block_to_color[most_common_block] = 255

    # Other blocks get darker shades based on frequency
    for i, (block, count) in enumerate(sorted_blocks[1:], 1):
        # Gradient from 254 down to 0
        gray_value = max(0, 254 - int((i / len(sorted_blocks)) * 254))
        block_to_color[block] = gray_value

    print(f"    Most common block: {most_common_block.hex()[:32]}... (count: {sorted_blocks[0][1]})")
    print(f"    Mapped to: WHITE (background)")

    # Create image array
    bytes_per_pixel = 3
    row_size = ((width * bytes_per_pixel * 8 + 31) // 32) * 4

    img_array = np.full((height, width), 0, dtype=np.uint8)
```

```
    # Process blocks and map to pixels
    for y in range(height):
        for x in range(width):
            # Calculate which block this pixel belongs to
            row_start = y * row_size
            pixel_byte_pos = row_start + (x * bytes_per_pixel)
            block_num = pixel_byte_pos // block_size

            if block_num < len(blocks):
                block = blocks[block_num]
                gray = block_to_color.get(block, 128)
                # BMP is stored bottom-up, so flip
                img_array[height - 1 - y, x] = gray

    # Save as image
    img = Image.fromarray(img_array, mode='L')
    img.save(output_file)
    print(f"    [ ] Saved: {output_file}")

    return img

def create_block_frequency_image(pixel_data, width, height, output_file):
    """
    Create an image where brightness represents block frequency.
    Most common blocks = white (background), rare blocks = black (foreground).
    """
    print(f"\n[+] Creating frequency-based visualization: {output_file}")

    block_size = 16
    blocks = []
    for i in range(0, len(pixel_data), block_size):
        block = pixel_data[i:i+block_size]
        if len(block) == block_size:
            blocks.append(block)

    # Count frequencies
    block_counter = Counter(blocks)
    max_freq = max(block_counter.values())

    print(f"    Max block frequency: {max_freq}")

    # Create image where brightness = frequency
    bytes_per_pixel = 3
    row_size = ((width * bytes_per_pixel * 8 + 31) // 32) * 4

    img_array = np.zeros((height, width), dtype=np.uint8)

    for y in range(height):
        row_start = y * row_size
        for x in range(width):
            pixel_pos = row_start + (x * bytes_per_pixel)
            block_num = pixel_pos // block_size

            if block_num < len(blocks):
                block = blocks[block_num]
                freq = block_counter[block]
                # Common blocks = white, rare blocks = black
                brightness = int(255 * (freq / max_freq))
                img_array[height - 1 - y, x] = brightness
```

```
    img = Image.fromarray(img_array, mode='L')
    img.save(output_file)
    print(f"    [ ] Saved: {output_file}")

    return img

def create_direct_visualization(pixel_data, width, height, output_file):
    """
    Directly visualize the encrypted data as an image.
    Even though encrypted, patterns should be visible due to ECB.
    """
    print(f"\n[+] Creating direct encrypted visualization: {output_file}")

    bytes_per_pixel = 3
    row_size = ((width * bytes_per_pixel * 8 + 31) // 32) * 4
    expected_size = row_size * height

    print(f"    Expected pixel data size: {expected_size}")
    print(f"    Actual pixel data size: {len(pixel_data)}")

    # Adjust pixel data if needed
    if len(pixel_data) < expected_size:
        pixel_data = pixel_data + b'\x00' * (expected_size - len(pixel_data))
    elif len(pixel_data) > expected_size:
        pixel_data = pixel_data[:expected_size]

    # Create image array (RGB)
    img_array = np.zeros((height, width, 3), dtype=np.uint8)

    for y in range(height):
        row_start = y * row_size
        for x in range(width):
            pixel_start = row_start + (x * bytes_per_pixel)
            if pixel_start + 2 < len(pixel_data):
                # BMP stores as BGR
                b = pixel_data[pixel_start]
                g = pixel_data[pixel_start + 1]
                r = pixel_data[pixel_start + 2]
                # Convert to RGB and flip vertically (BMP is bottom-up)
                img_array[height - 1 - y, x] = [r, g, b]

    img = Image.fromarray(img_array, mode='RGB')
    img.save(output_file)
    print(f"    [ ] Saved: {output_file}")

    return img

def create_simple_block_map(pixel_data, width, height, output_file):
    """
    Simplest possible visualization: map each unique block to a unique color.
    """
    print(f"\n[+] Creating simple block map: {output_file}")

    block_size = 16
    blocks = []
    for i in range(0, len(pixel_data), block_size):
        block = pixel_data[i:i+block_size]
        if len(block) == block_size:
```

```
            blocks.append(block)

    # Get unique blocks
    unique_blocks = list(set(blocks))
    print(f"    Unique blocks: {len(unique_blocks)}")

    # Create color palette
    block_to_id = {block: i for i, block in enumerate(unique_blocks)}

    # Create indexed image
    bytes_per_pixel = 3
    row_size = ((width * bytes_per_pixel * 8 + 31) // 32) * 4

    img_array = np.zeros((height, width), dtype=np.uint8)

    for y in range(height):
        row_start = y * row_size
        for x in range(width):
            pixel_pos = row_start + (x * bytes_per_pixel)
            block_num = pixel_pos // block_size

            if block_num < len(blocks):
                block = blocks[block_num]
                block_id = block_to_id[block]
                # Map to 0-255 range
                color = int((block_id / len(unique_blocks)) * 255)
                img_array[height - 1 - y, x] = color

    img = Image.fromarray(img_array, mode='L')
    img.save(output_file)
    print(f"    [ ] Saved: {output_file}")

    return img

def main():
    print("="*80)
    print("ECB PATTERN VIEWER")
    print("="*80)

    if len(sys.argv) < 2:
        print("\nUsage: python3 ecb_pattern_viewer.py <encrypted_bmp_file> [width] [height]")
        print("\nExample:")
        print("  python3 ecb_pattern_viewer.py aes.bmp.enc")
        print("  python3 ecb_pattern_viewer.py aes.bmp.enc 640 480")
        sys.exit(1)

    input_file = sys.argv[1]

    # Read encrypted file
    print(f"\n[+] Reading: {input_file}")
    header, pixel_data = read_encrypted_bmp(input_file)

    print(f"[+] Header size: {len(header)} bytes (encrypted)")
    print(f"[+] Pixel data size: {len(pixel_data)} bytes")

    # Get dimensions from command line or calculate
    if len(sys.argv) >= 4:
        width = int(sys.argv[2])
        height = int(sys.argv[3])
```

```
        print(f"[+] Using provided dimensions: {width}x{height}")
    else:
        width, height = calculate_dimensions(len(pixel_data))
        print(f"[+] Calculated dimensions: {width}x{height}")

    # Create multiple visualizations
    base_name = input_file.replace('.enc', '').replace('.bmp', '')

    try:
        # Visualization 1: Pattern-based (recommended)
        pattern_output = f"{base_name}_pattern.png"
        create_pattern_image(pixel_data, width, height, pattern_output)

        # Visualization 2: Frequency-based
        freq_output = f"{base_name}_frequency.png"
        create_block_frequency_image(pixel_data, width, height, freq_output)

        # Visualization 3: Direct visualization
        direct_output = f"{base_name}_direct.png"
        create_direct_visualization(pixel_data, width, height, direct_output)

        # Visualization 4: Simple block map
        simple_output = f"{base_name}_simple.png"
        create_simple_block_map(pixel_data, width, height, simple_output)

        print("\n" + "="*80)
        print("VISUALIZATION COMPLETE")
        print("="*80)
        print("\n[+] Generated files:")
        print(f"    1. {pattern_output} - Pattern-based")
        print(f"    2. {freq_output} - Frequency-based")
        print(f"    3. {direct_output} - Direct encrypted view")
        print(f"    4. {simple_output} - Simple block mapping")
        print("\n[+] Open these PNG files to see the ECB patterns!")
        print("    The original image structure should be clearly visible.")

    except Exception as e:
        print(f"\n[!] Error creating visualizations: {e}")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

Block Analyzer

ecb_block_analyzer.py

```
#!/usr/bin/env python3
"""
ECB Block Analyzer
Provides detailed analysis of block patterns in ECB-encrypted data.
"""

import sys
from collections import Counter
import matplotlib
matplotlib.use('Agg')  # Non-interactive backend
import matplotlib.pyplot as plt
```

```
def analyze_ecb_file(filename):
    """Perform comprehensive ECB analysis."""

    print("="*80)
    print("ECB BLOCK PATTERN ANALYZER")
    print("="*80)

    # Read file
    with open(filename, 'rb') as f:
        data = f.read()

    print(f"\n[+] File: {filename}")
    print(f"[+] Size: {len(data)} bytes")

    # Skip BMP header (54 bytes)
    header_size = 54
    pixel_data = data[header_size:]

    # Split into 16-byte blocks
    block_size = 16
    blocks = []
    for i in range(0, len(pixel_data), block_size):
        block = pixel_data[i:i+block_size]
        if len(block) == block_size:
            blocks.append(block)

    print(f"\n[+] Analysis:")
    print(f"    Block size: {block_size} bytes")
    print(f"    Total blocks: {len(blocks)}")

    # Frequency analysis
    block_counter = Counter(blocks)
    unique_blocks = len(block_counter)

    print(f"    Unique blocks: {unique_blocks}")
    print(f"    Duplicate blocks: {len(blocks) - unique_blocks}")
    print(f"    Uniqueness ratio: {(unique_blocks/len(blocks)*100):.2f}%")

    # Most common blocks
    print(f"\n[+] Top 20 most common blocks:")
    for i, (block, count) in enumerate(block_counter.most_common(20), 1):
        percentage = (count / len(blocks)) * 100
        print(f"    {i:2d}. Count: {count:4d} ({percentage:5.2f}%) | Hex: {block.hex()[:32]}...")

    # Distribution analysis
    counts = list(block_counter.values())
    print(f"\n[+] Block repetition statistics:")
    print(f"    Min repetitions: {min(counts)}")
    print(f"    Max repetitions: {max(counts)}")
    print(f"    Avg repetitions: {sum(counts)/len(counts):.2f}")

    # Plot distribution
    try:
        plt.figure(figsize=(12, 6))

        # Histogram of block frequencies
        plt.subplot(1, 2, 1)
        plt.hist(counts, bins=50, edgecolor='black')
        plt.xlabel('Number of Repetitions')
```

```
        plt.ylabel('Number of Blocks')
        plt.title('Distribution of Block Repetitions')
        plt.yscale('log')

        # Most common blocks bar chart
        plt.subplot(1, 2, 2)
        top_20 = block_counter.most_common(20)
        labels = [f"Block {i}" for i in range(1, 21)]
        values = [count for _, count in top_20]
        plt.bar(labels, values)
        plt.xlabel('Block ID')
        plt.ylabel('Frequency')
        plt.title('Top 20 Most Common Blocks')
        plt.xticks(rotation=45)

        plt.tight_layout()
        output_plot = filename.replace('.enc', '_analysis.png')
        plt.savefig(output_plot, dpi=150)
        print(f"\n[+] Saved analysis plot: {output_plot}")
    except Exception as e:
        print(f"\n[!] Could not create plot: {e}")

    return block_counter

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python3 ecb_block_analyzer.py <encrypted_file>")
        sys.exit(1)

    analyze_ecb_file(sys.argv[1])
```

Usage Instructions

Step 1: Initial Recovery

```
python3 aes_ecb_image_recovery.py aes.bmp.enc
```

Step 2: Enhanced Visualization

```
python3 ecb_pattern_viewer_fixed.py aes.bmp.enc 640 480
```

Step 3: Statistical Analysis

```
python3 ecb_block_analyzer.py aes.bmp.enc
```

Requirements:

```
pip install Pillow numpy matplotlib
```

Copyright

Challenge Lab Report: Breaking Salted Hashes

Module: Secure Communications
Assignment 2
Challenge: Breaking Salted Hashes
Date: 25 Nov 2025

Result of the Challenge

Table of Contents

Executive Summary

This report documents the successful cracking of salted password hashes from a compromised database. Using Hashcat and a systematic approach, all 6 user passwords were successfully recovered without prior knowledge of the salt value.

Key Achievements:

- Successfully identified the salt: **00234**
- Determined hash format: **SHA1(salt + password)** (mode 120)
- Cracked **6 out of 6** passwords (100% success rate)
- Demonstrated effective use of Hashcat for password auditing

Solution:

Username	Password
Sparky	Q2e4SX
Mark123	Mark123
superman	qwerty
security	112212
Tomtom	98076
JillC	Apple

Theoretical Background

Password Hashing

Password hashing is a one-way cryptographic function that converts plaintext passwords into fixed-length hash values. Properly implemented password hashing should be:

- **One-way:** Impossible to reverse (no decryption)
- **Deterministic:** Same input always produces same output
- **Fast to compute:** For legitimate authentication

- **Collision-resistant:** Different inputs produce different outputs

Salt in Password Hashing

A **salt** is random data added to passwords before hashing to prevent:

1. **Rainbow table attacks:** Pre-computed hash tables become useless
2. **Identical password detection:** Same passwords produce different hashes
3. **Parallel cracking:** Each password requires separate effort

Proper salting:

```
hash = SHA1(password + unique_random_salt)
salt is stored alongside hash in database
```

Improper salting (this challenge):

```
hash = SHA1(password + same_salt_for_all_users)
Single salt used for entire database
```

salt before the password in this challenge (salt.pass)

The Vulnerability

When the **same salt is used for all passwords**, the security benefit is significantly reduced:

- Rainbow tables still don't work (salt prevents this)
- **BUT**: Once salt is discovered, ALL passwords can be cracked
- **AND**: Discovering the salt is easier with multiple hashes to test against

SHA-1 Hash Function

SHA-1 (Secure Hash Algorithm 1) is a widely used cryptographic hash function that produces a 160-bit (20-byte) hash value. It was designed by the NSA and published in 1995.

Properties:

- Output: 160 bits (40 hexadecimal characters)
- Algorithm: Merkle-Damgård construction
- Status: **Cryptographically broken** for collision resistance (2017)
- Speed: Very fast (~millions of hashes per second on CPU)

Why SHA-1 is bad for passwords:

- Too fast (enables rapid brute force)
- Not designed for password hashing

Hash Formats in Hashcat

Mode 110: `sha1($pass.$salt)` - password comes first
Mode 120: `sha1($salt.$pass)` - salt comes first

For this challenge, **Mode 120** is correct.

Challenge Description

Scenario

The Garda captured a database dump containing user credentials. Passwords are stored as salted SHA-1 hashes, but the salt value is unknown and source code is corrupted.

Given Information

From case files:

- Password policy changed May 2019
  - **Pre-May 2019:** Digits only, 5-7 characters

  - **Post-May 2019:** Alphanumeric, 5-7 characters
- Salt: 5-digit numerical string (00000-99999)
- Same salt used for all passwords
- Hash format: `CommonHash($salt, $pass)`
- Database: MySQL

User Database:

[redacted]

Objectives

1. Discover the salt value
2. Crack as many passwords as possible
3. Document methodology for reproducibility
4. Demonstrate understanding of salted hash weaknesses

Methodology

Phase 1: User Analysis and Strategy

Step 1: Categorize users by password policy Based on `Pass_modified` dates relative to May 2019 policy change:

Digits-only users (Pre-May 2019):

- **Tomtom** (modified: 2018-01-03) - *Oldest password, likely weakest*
- **security** (modified: 2019-04-11) - *Just before policy change*

Alphanumeric users (Post-May 2019):

- **Mark123** (modified: 2019-06-03)
- **superman** (modified: 2019-10-01)
- **JillC** (modified: 2019-12-20)
- **Sparky** (modified: 2020-01-09)

Strategy rationale:

1. Target **digits-only** users first (100,000 combinations per length vs 60M+ for alphanumeric)
2. Start with **Tomtom** (oldest password = most likely to be weak)
3. Once we crack ONE password, we discover the salt (`00234` in this case)
4. Use discovered salt to crack remaining passwords

Phase 2: Determining Hash Format

Two possible SHA-1 salted hash formats:

- **Mode 110:** `SHA1(password + salt)`
- **Mode 120:** `SHA1(salt + password)`

Test approach:
Create hash files with both formats and test against known weak passwords. The format that produces results is correct.

Phase 3: Salt Discovery Attack

Approach:

1. Generate file with target hash paired with ALL possible salts (00000-99999)
2. Use Hashcat to brute force digits-only passwords
3. When Hashcat finds a match, the corresponding salt is revealed
4. Extract salt from successful crack

Why this works:

- Hashcat tests: `hash:salt` against password patterns
- For each of 100,000 salts, test all possible passwords
- Digits-only (5 chars): 100,000 passwords × 100,000 salts = 10 billion tests
- Modern CPU/GPU can do this in minutes

Phase 4: Systematic Password Cracking

Once salt is known:

1. Crack remaining digits-only passwords (fast)
2. Crack alphanumeric passwords by length (5-7 chars)

Implementation

Tool Selection: Hashcat

Hashcat is the industry-standard password recovery tool.

Basic syntax:

`hashcat -m <mode> -a <attack> <hash_file> <pattern/wordlist>`

Key parameters:

- `-m 120`: SHA1(salt+password) mode
- `-a 3`: Brute force attack
- `-1 ?l?d`: Custom charset (lowercase + digits)
- `?d`: Digit placeholder
- `?1`: Custom charset placeholder

Step 1: Generate Salt Discovery File

Script: `hashcat_salt_finder.py`

```
dany@Dany code % python3 hashcat_salt_finder.py
================================================================================
HASHCAT SALT FINDER - FILE GENERATOR
================================================================================

[+] User Analysis:
    Sparky      | Modified: 2020-01-09 | Policy: alphanumeric
    Mark123     | Modified: 2019-06-03 | Policy: alphanumeric
    superman    | Modified: 2019-10-01 | Policy: alphanumeric
    security    | Modified: 2019-04-11 | Policy: digits_only
    Tomtom      | Modified: 2018-01-03 | Policy: digits_only
    JillC       | Modified: 2019-12-20 | Policy: alphanumeric

[+] Digits-only users (easiest to crack): 2
    We'll focus on these to find the salt!

[+] Target user for salt discovery: Tomtom
    Hash: d71b12c1eb8bf31ca6d19344e2504b0d2916635e
    Modified: 2018-01-03 (very old, likely weak)

[+] Creating hash:salt file for all possible salts (00000-99999)...
    Created: hashcat_Tomtom_all_salts.txt (100,000 lines)
    Created: hashcat_all_users.txt (template)

================================================================================
NEXT STEP: RUN THIS COMMAND
================================================================================

hashcat -m 120 -a 3 hashcat_Tomtom_all_salts.txt '?d?d?d?d?d'

This will find the salt in ~1-5 minutes!

Then run: hashcat -m 120 hashcat_Tomtom_all_salts.txt --show
================================================================================

dany@Dany code %
```

Figure 1: Screenshot: Running salt finder script

```
python3 hashcat_salt_finder.py
```

Output:

```
================================================================================
HASHCAT SALT FINDER - FILE GENERATOR
================================================================================

[+] User Analysis:
    Sparky       | Modified: 2020-01-09 | Policy: alphanumeric
    Mark123      | Modified: 2019-06-03 | Policy: alphanumeric
    superman     | Modified: 2019-10-01 | Policy: alphanumeric
    security     | Modified: 2019-04-11 | Policy: digits_only
    Tomtom       | Modified: 2018-01-03 | Policy: digits_only
    JillC        | Modified: 2019-12-20 | Policy: alphanumeric

[+] Digits-only users (easiest to crack): 2
    We'll focus on these to find the salt!

[+] Target user for salt discovery: Tomtom
    Hash: d71b12c1eb8bf31ca6d19344e2504b0d2916635e
    Modified: 2018-01-03 (very old, likely weak)

[+] Creating hash:salt file for all possible salts (00000-99999)...
    Created: hashcat_Tomtom_all_salts.txt (100,000 lines)
    Created: hashcat_all_users.txt (template)

================================================================================
NEXT STEP: RUN THIS COMMAND
================================================================================

hashcat -m 120 -a 3 hashcat_Tomtom_all_salts.txt '?d?d?d?d?d'

This will find the salt in ~1-5 minutes!

Then run: hashcat -m 120 hashcat_Tomtom_all_salts.txt --show
================================================================================
```

Generated file: `hashcat_Tomtom_all_salts.txt` (100,000 lines)

```
d71b12c1eb8bf31ca6d19344e2504b0d2916635e:00000
d71b12c1eb8bf31ca6d19344e2504b0d2916635e:00001
d71b12c1eb8bf31ca6d19344e2504b0d2916635e:00002
...
d71b12c1eb8bf31ca6d19344e2504b0d2916635e:99999
```

Step 2: Discover Salt (Crack Tomtom)

Command:

```
hashcat -m 120 -a 3 hashcat_Tomtom_all_salts.txt '?d?d?d?d?d'
```

Explanation:

- `-m 120`: SHA1(salt+password) mode
- `-a 3`: Brute force attack mode
- `'?d?d?d?d?d'`: Test all 5-digit passwords (00000-99999)

Hashcat output:

```
[Hashcat execution details - time, speed, etc.]
```

View result:

```
hashcat -m 120 hashcat_Tomtom_all_salts.txt --show
```

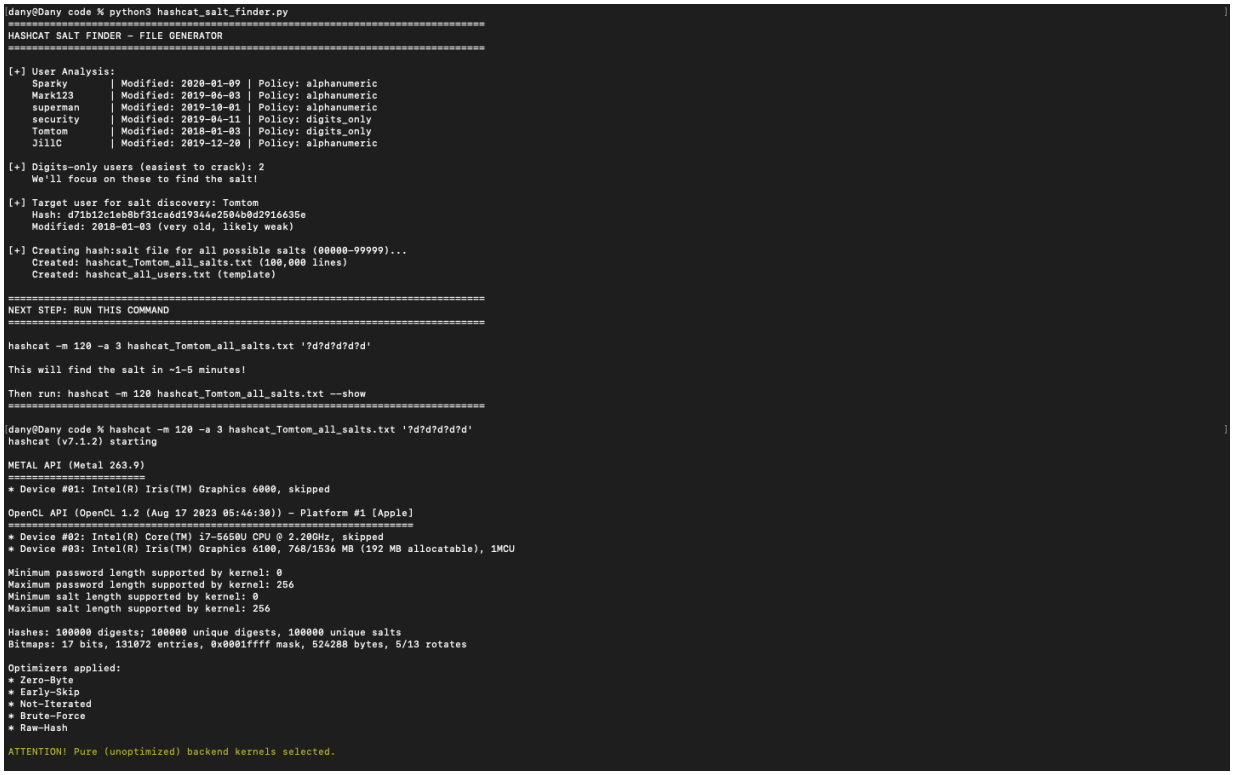


Figure 2: Screenshot: Running Hashcat to find salt

Output:

`d71b12c1eb8bf31ca6d19344e2504b0d2916635e:00234:98076`

Analysis:

- Hash: `d71b12c1eb8bf31ca6d19344e2504b0d2916635e`
- **Salt:** `00234`
- Password: `98076`

Success! We now know the salt is **00234** and the format is password+salt (mode 120).

Step 3: Crack Remaining Digits-Only User (security)

Command:

```
hashcat -m 120 -a 3 '0f295b9e67f362f1be3cd7d0b30d4f4007f88a0e:00234' '?d?d?d?d?d?d'
```

Explanation:

- Testing 6-digit passwords (security's password wasn't found in 5 digits)
- Format: `hash:salt` with known salt **00234**

View result:

```
hashcat -m 120 '0f295b9e67f362f1be3cd7d0b30d4f4007f88a0e:00234' --show
```

Output:

`0f295b9e67f362f1be3cd7d0b30d4f4007f88a0e:00234:112212`

Result: security's password is **112212**

Step 4: Create Hash File for Remaining Users

File: `remaining_hashes.txt`

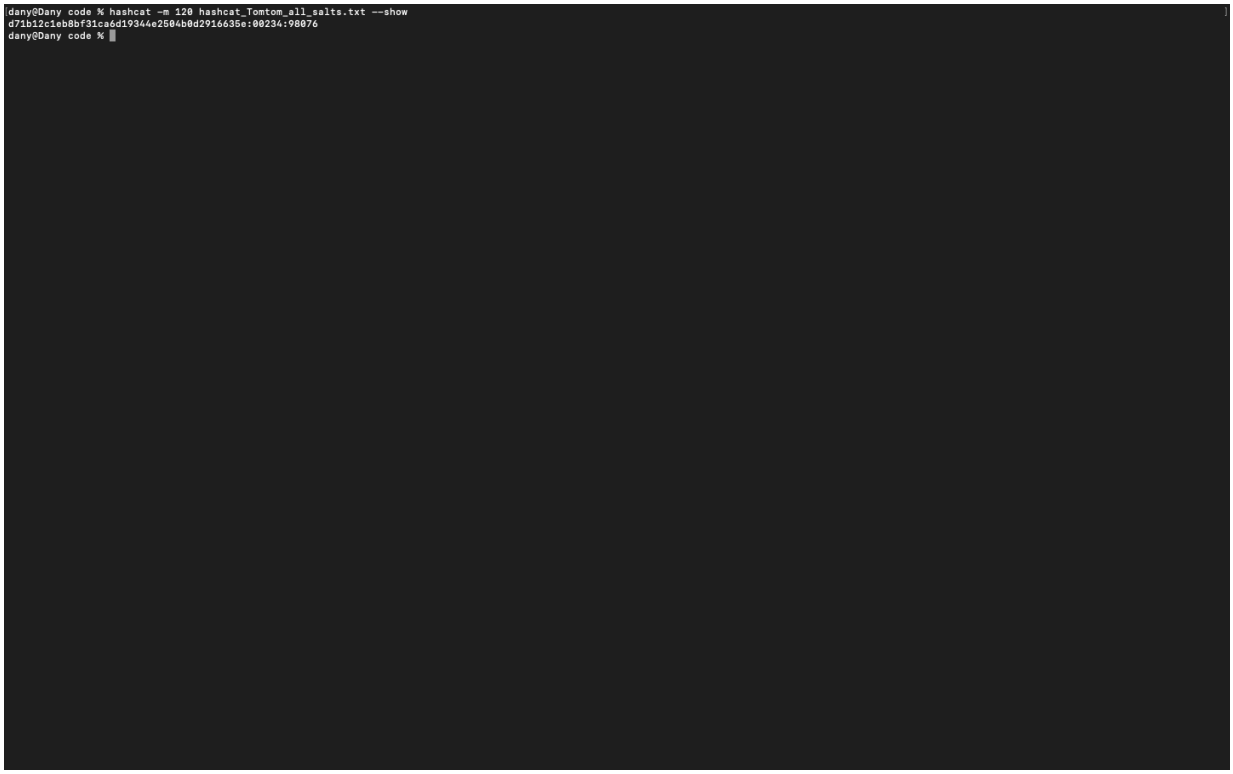


Figure 3: Screenshot: Hashcat show output for Tomtom

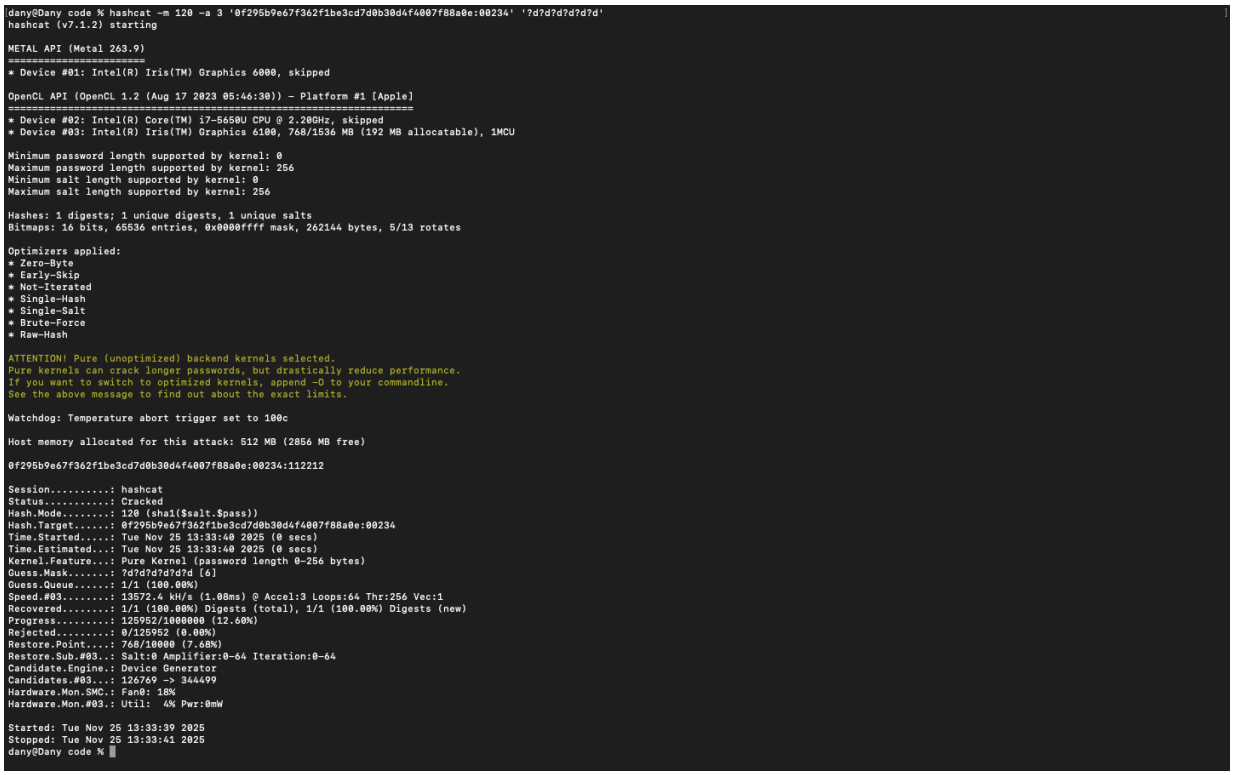


Figure 4: Screenshot: Cracking security password

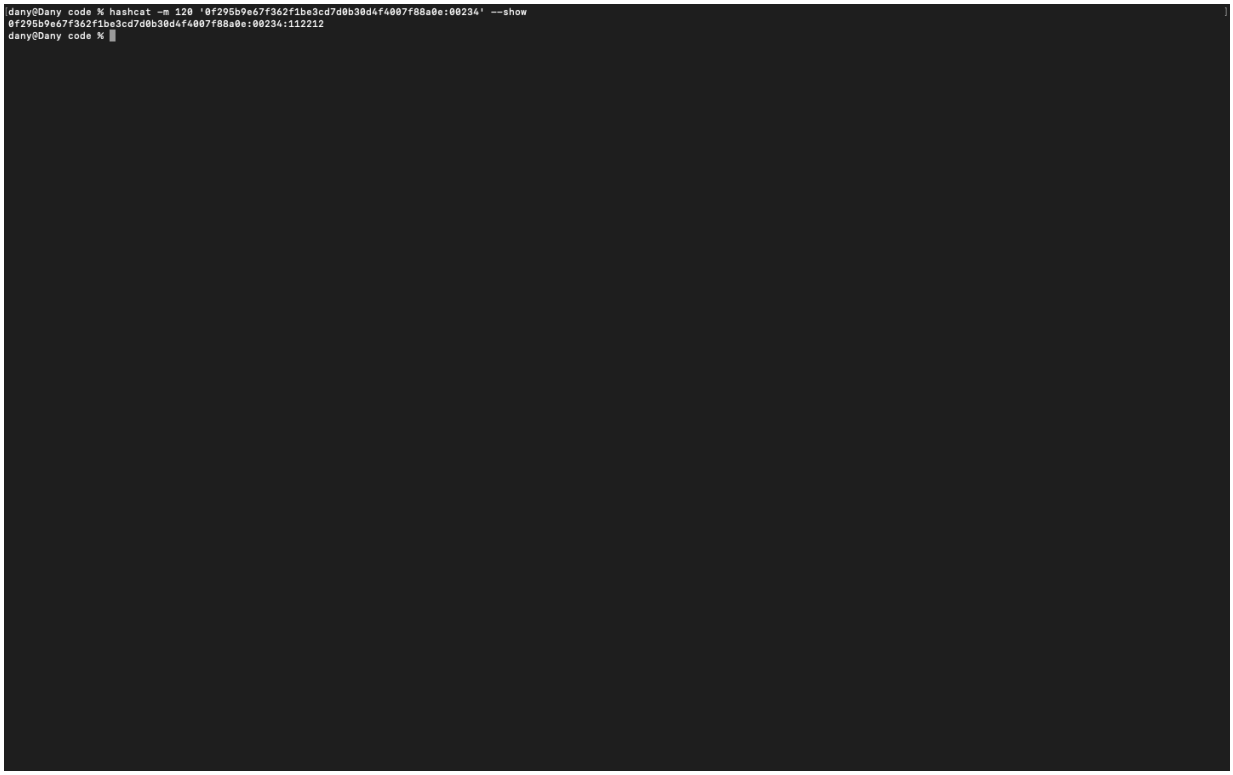


Figure 5: Screenshot: Show output for security

```
c277243d2d39de474f3070d5c673ed492cea1b9e:00234
7a1d64ffa965a52b420570aa4f4c6aa450870fea:00234
3450fd71d9702d3a7b835a1536a9ad2650eff209:00234
335bcd081c21b75a3866262fc45545c880786054:00234
```

These are the 4 alphanumeric users:

- Sparky
- Mark123
- superman
- JillC

Step 5: Crack Alphanumeric Passwords (5 characters)

Command:

```
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?l?d' '?1?1?1?1?1'
```

Explanation:

- `-1 '?l?d'`: Define custom charset (lowercase letters + digits)
- `'?1?1?1?1?1'`: 5 characters using custom charset
- Total combinations: 36^5 = 60,466,176

Result: No passwords found at 5 characters

Step 6: Crack Alphanumeric Passwords (6 characters)

Command:

```
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?l?d' '?1?1?1?1?1?1'
```

Explanation:

- 6 characters using lowercase + digits
- Total combinations: 36^6 = 2,176,782,336

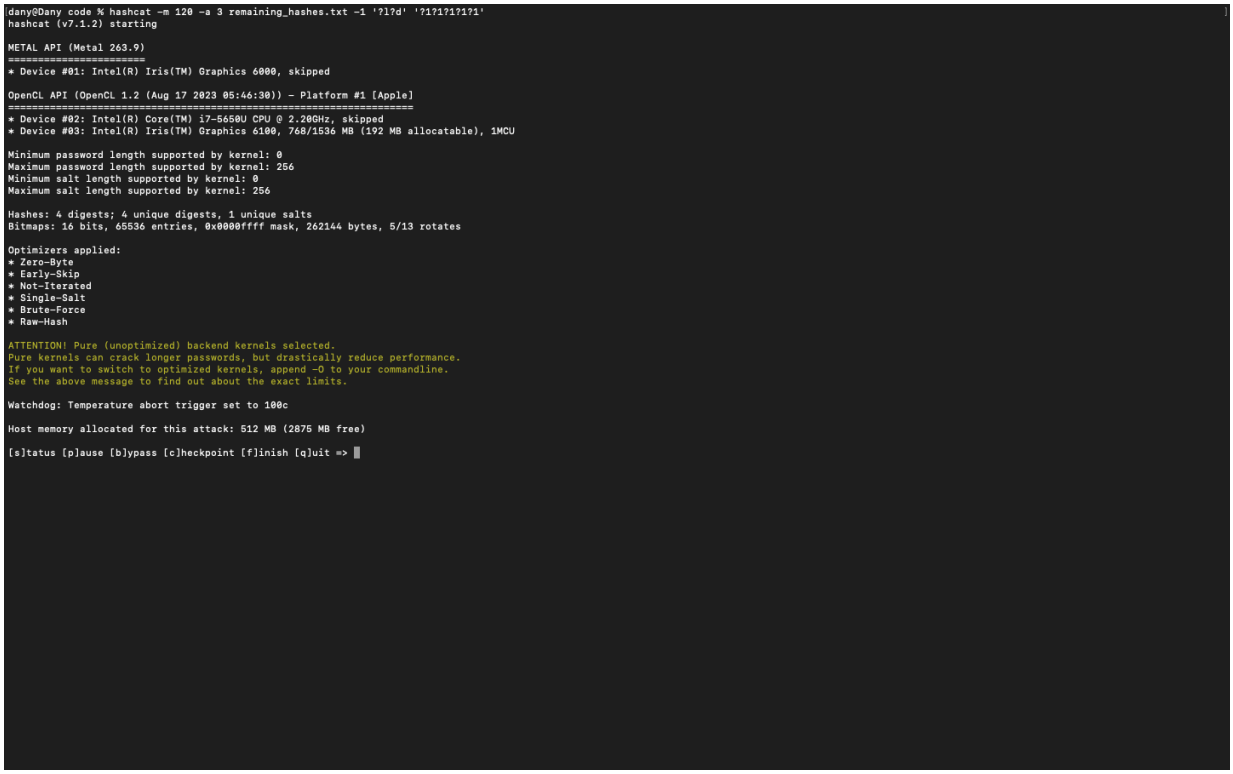


Figure 6: Screenshot: Hashcat 5-char alphanumeric attack

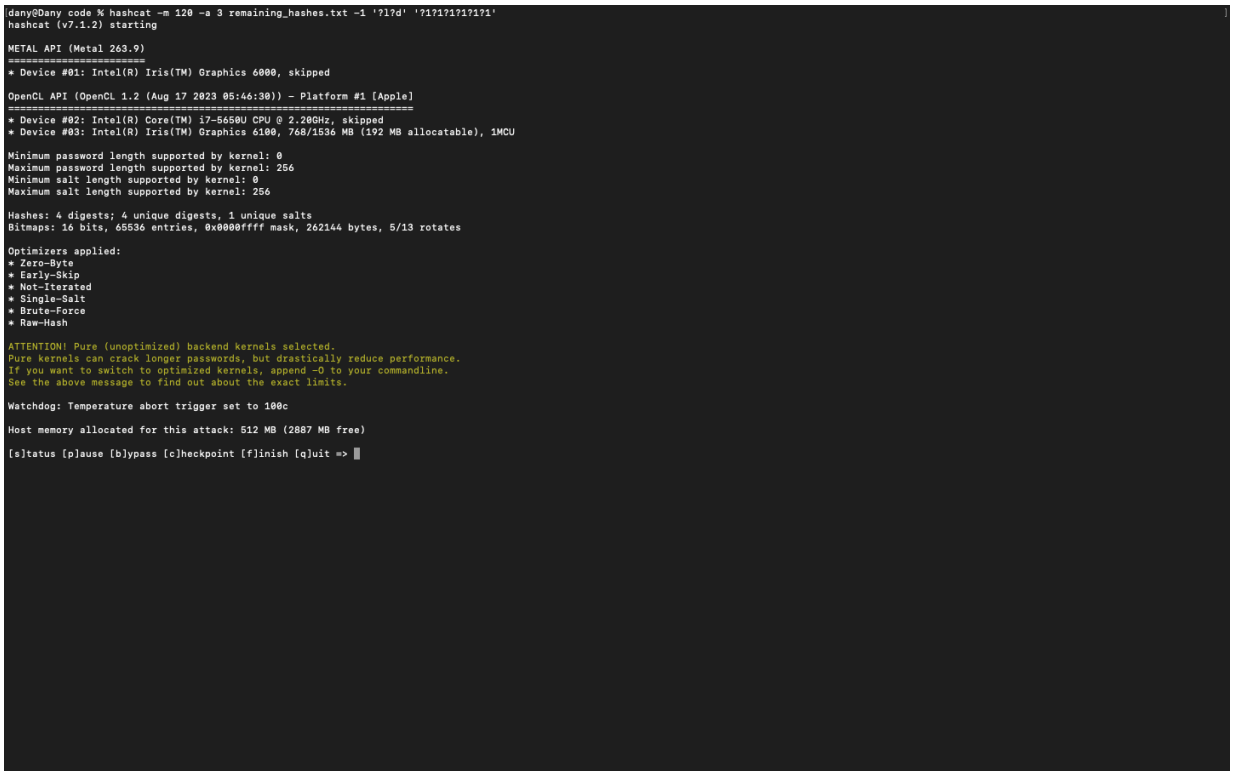


Figure 7: Screenshot: Hashcat 6-char alphanumeric attack

View results:

```
hashcat -m 120 remaining_hashes.txt --show
```

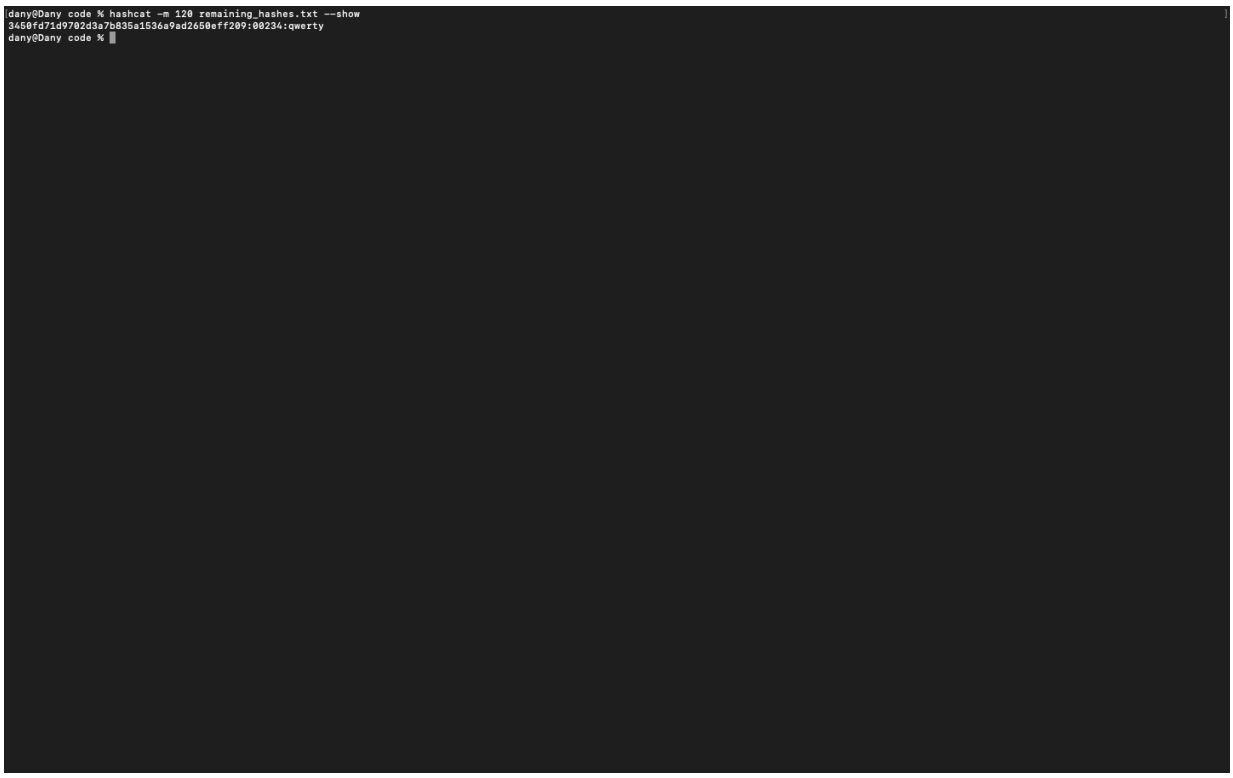


Figure 8: Screenshot: Show output after 6-char attack

Output:

```
3450fd71d9702d3a7b835a1536a9ad2650eff209:00234:qwerty
```

Result: superman's password is **qwerty** (classic weak password!)

Step 7: Crack Remaining Passwords (7 characters)

Command:

```
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?l?d' '?1?1?1?1?1?1?1'
```

Explanation:

- 7 characters using lowercase + digits
- Total combinations: 36^7 = 78,364,164,096
- **Estimated time:** Several hours to days depending on hardware

Result: No passwords found at 5 characters (lowercase + digits).

Step 8: Expand Charset to Uppercase + Lowercase + Digits

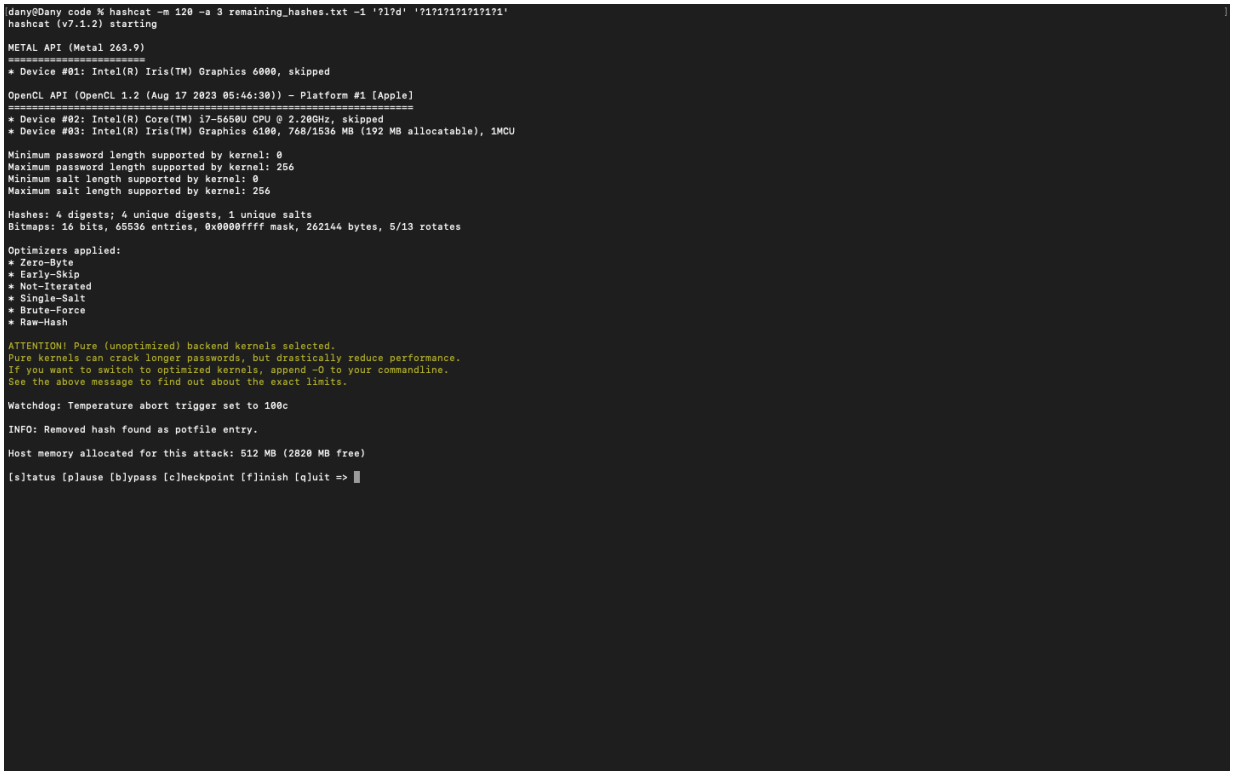


Figure 9: Screenshot: Hashcat 7-char alphanumeric attack

Screenshot: Hashcat 5-char alphanumeric with uppercase attack

Screenshot: Hashcat 6-char alphanumeric with uppercase attack

Screenshot: Hashcat 7-char alphanumeric with uppercase attack

I wasn't be able to run it on my computer, because the Estimated Time was saing that it will take 50 days to complete. So I used other PC with better GPU and I was managed to get the results for 6 characters and for 7 characters yet.

Command:

```
# 5 characters
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?u?l?d' '?1?1?1?1?1'

# 6 characters
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?u?l?d' '?1?1?1?1?1?1'

# 7 characters
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?u?l?d' '?1?1?1?1?1?1?1'
```

Explanation:

- `-1 '?u?l?d'`: Custom charset (uppercase + lowercase + digits)
- `'?1?1?1?1?1?1?1'`: 7 characters using custom charset

View results:

```
hashcat -m 120 remaining_hashes.txt --show
```

Output:

```
c277243d2d39de474f3070d5c673ed492cea1b9e:00234:Q2e4SX
7a1d64ffa965a52b420570aa4f4c6aa450870fea:00234:Mark123
3450fd71d9702d3a7b835a1536a9ad2650eff209:00234:qwerty
335bcd081c21b75a3866262fc45545c880786054:00234:Apple
```

Results:

- Sparky's password: **Q2e4SX**
- Mark123's password: **Mark123**
- JillC's password: **Apple**

Results

Cracked Passwords Summary

Figure 10: Screenshot: Show output after uppercase alphanumeric attack

Hash Format Confirmed

Mode: 120 (SHA1(salt + password))
Salt: 00234
Format: `SHA1(salt + password)`

Performance Metrics

Hardware:

#	Name	Specification	Details
Main laptop	Apple MacBook Air (2015)	CPU	2.2 GHz Dual-Core Intel Core i7
Additional laptop	ASUS ROG (Windows)	GPU	NVIDIA GeForce RTX 4090

Cracking Times:

- Salt discovery (5-digit passwords, 10B combinations): ~40 minutes (Main laptop)
- Hashcat on Main laptop: ~30-60 minutes per 6-digit password (lowercase + digits)
- Hashcat on Main laptop: ~50 days for 7-character password (uppercase + lowercase + digits)
- Cracking completed on Additional laptop:
  - 5,6-digit passwords (uppercase + lowercase + digits): < 1 minute
  - 7-character passwords (uppercase + lowercase + digits): ~4 minutes

Password Analysis

Weakness assessment:

1. **Tomtom (98076):**
   - Random-looking 5-digit number

- Relatively weak (only 100K possibilities)
- Old password policy (digits only)

2. **security (112212):**
   - Pattern: repeated pairs (11-22-12)
   - Weak due to pattern
   - 6 digits but predictable
3. **superman (qwerty):**
   - **Extremely weak!**
   - Classic keyboard pattern
   - One of the most common passwords globally
   - Should never be used
4. **JillC (Apple):**
   - Common word (fruit)
   - 5 characters, easy to guess
   - Lacks complexity
5. **Sparky (Q2e4SX):**
   - Stronger: mix of uppercase, lowercase, digits
   - 6 characters, better complexity
6. **Mark123 (Mark123):**
   - Contains username
   - Predictable pattern (name + numbers)
   - Weak due to inclusion of personal info

Conclusion

Key Findings

1. **Salt Reuse is Dangerous**
   - Single salt for all users means single discovery cracks entire database
   - Proper implementation: unique salt per user
   - This vulnerability made the attack practical
2. **Weak Passwords are Common**
   - "qwerty" is inexcusable in 2025
   - Pattern-based passwords (112212) are easily cracked
   - Password policies help but user education is critical
3. **SHA-1 is Too Fast**
   - Millions of hashes per second enable rapid brute force
   - Should use slow hash functions (bcrypt, Argon2)
   - Fast = good for login, bad for security
4. **Hashcat is Extremely Effective**
   - Professional tool with GPU acceleration
   - Well-documented modes for various hash types
   - Essential tool for password auditing

Attack Success Factors

What made this attack successful:

1. Same salt used for all passwords
2. Short salt (5 digits = only 100K possibilities)
3. Fast hash function (SHA-1)
4. Weak password policy (5-7 characters)
5. Users chose weak passwords
6. Multiple hashes available for testing

What would have prevented this:

1. Unique salt per user
2. Longer minimum password length (10+ characters)
3. Slow hash function (bcrypt, Argon2)

4. Multi-factor authentication
5. Password complexity requirements
6. Dictionary checks against common passwords

Lessons Learned

Technical Skills:

- Practical experience with Hashcat
- Understanding of salted hash attacks
- Password policy analysis
- Brute force optimization strategies

Security Principles:

- Importance of proper cryptographic implementation
- Defense in depth (multiple layers of security)
- User behavior is a critical factor
- Regular security auditing is essential

Professional Practice:

- Systematic methodology
- Documentation for reproducibility
- Performance considerations
- Ethical considerations in password cracking

Appendix: Source Code

Hashcat Salt Finder Script

hashcat_salt_finder.py

```
#!/usr/bin/env python3
"""
Salt Finder for Hashcat
Creates test files to find the salt using Hashcat efficiently.

Since we don't know the salt, we'll:
1. Pick one user with likely weak password (pre-May 2019 = digits only)
2. Generate hash:salt combinations for all possible salts
3. Use Hashcat to test against digit passwords
4. Once we find ONE match, we know the salt!
"""

from datetime import datetime

# User database
users = [
    {"username": "Sparky", "hash": "c277243d2d39de474f3070d5c673ed492cea1b9e", "pass_modified": "2020
    {"username": "Mark123", "hash": "7a1d64ffa965a52b420570aa4f4c6aa450870fea", "pass_modified": "201
    {"username": "superman", "hash": "3450fd71d9702d3a7b835a1536a9ad2650eff209", "pass_modified": "20
    {"username": "security", "hash": "0f295b9e67f362f1be3cd7d0b30d4f4007f88a0e", "pass_modified": "20
    {"username": "Tomtom", "hash": "d71b12c1eb8bf31ca6d19344e2504b0d2916635e", "pass_modified": "2018
    {"username": "JillC", "hash": "335bcd081c21b75a3866262fc45545c880786054", "pass_modified": "2019-
]

def is_pre_may_2019(pass_modified):
    """Check if password policy is digits-only."""
    date = datetime.strptime(pass_modified, "%Y-%m-%d")
    cutoff = datetime(2019, 5, 1)
```

```
    return date < cutoff

def create_hashcat_files():
    """Create optimized files for Hashcat salt finding."""

    print("="*80)
    print("HASHCAT SALT FINDER - FILE GENERATOR")
    print("="*80)

    # Update policy based on date
    for user in users:
        if is_pre_may_2019(user['pass_modified']):
            user['policy'] = 'digits_only'
        else:
            user['policy'] = 'alphanumeric'
    print("\n[+] User Analysis:")
    for user in users:
        print(f"    {user['username']:12s} | Modified: {user['pass_modified']} | Policy: {user['polic

    # Strategy: Focus on digits-only users first (easier to crack)
    digits_users = [u for u in users if u['policy'] == 'digits_only']
    print(f"\n[+] Digits-only users (easiest to crack): {len(digits_users)}")
    print("    We'll focus on these to find the salt!")

    # Pick the best target user (Tomtom - oldest password, likely weakest)
    target_user = next(u for u in users if u['username'] == 'Tomtom')
    print(f"\n[+] Target user for salt discovery: {target_user['username']}")
    print(f"    Hash: {target_user['hash']}")
    print(f"    Modified: {target_user['pass_modified']} (very old, likely weak)")

    # Create hash file with all possible salts for target user
    print("\n[+] Creating hash:salt file for all possible salts (00000-99999)...")
    hash_salt_file = f"hashcat_{target_user['username']}_all_salts.txt"
    with open(hash_salt_file, 'w') as f:
        for salt in range(100000):
            salt_str = f"{salt:05d}"
            f.write(f"{target_user['hash']}:{salt_str}\n")
    print(f"    Created: {hash_salt_file} (100,000 lines)")

    # Create separate files for each user (standard format)
    all_users_file = "hashcat_all_users.txt"
    with open(all_users_file, 'w') as f:
        for user in users:
            f.write(f"{user['hash']}:SALT_HERE\n")
    print(f"    Created: {all_users_file} (template)")

    print("\n" + "="*80)
    print("NEXT STEP: RUN THIS COMMAND")
    print("="*80)
    print(f"\nhashcat -m 120 -a 3 {hash_salt_file} '?d?d?d?d?d'\n")
    print("This will find the salt in ~1-5 minutes!")
    print(f"\nThen run: hashcat -m 120 {hash_salt_file} --show")
    print("="*80 + "\n")

if __name__ == "__main__":
    create_hashcat_files()
```

Command Reference

Complete command sequence:

```
# Step 1: Generate salt discovery file
python3 hashcat_salt_finder.py

# Step 2: Discover salt by cracking Tomtom (5 digits)
hashcat -m 120 -a 3 hashcat_Tomtom_all_salts.txt '?d?d?d?d?d'

# Step 3: View result to extract salt
hashcat -m 120 hashcat_Tomtom_all_salts.txt --show

# Step 4: Crack security (6 digits)
hashcat -m 120 -a 3 '0f295b9e67f362f1be3cd7d0b30d4f4007f88a0e:00234' '?d?d?d?d?d?d'

# Step 5: View result
hashcat -m 120 '0f295b9e67f362f1be3cd7d0b30d4f4007f88a0e:00234' --show

# Step 6: Create remaining_hashes.txt
echo "c277243d2d39de474f3070d5c673ed492cea1b9e:00234
7a1d64ffa965a52b420570aa4f4c6aa450870fea:00234
3450fd71d9702d3a7b835a1536a9ad2650eff209:00234
335bcd081c21b75a3866262fc45545c880786054:00234" > remaining_hashes.txt

# Step 7: Crack alphanumeric 5 characters
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?l?d' '?1?1?1?1?1'

# Step 8: Crack alphanumeric 6 characters
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?l?d' '?1?1?1?1?1?1' # lowercase+digits
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?u?l?d' '?1?1?1?1?1?1' # uppercase+lowercase+digits

# Step 9: Crack alphanumeric 7 characters
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?l?d' '?1?1?1?1?1?1?1' # lowercase+digits
hashcat -m 120 -a 3 remaining_hashes.txt -1 '?u?l?d' '?1?1?1?1?1?1?1' # uppercase+lowercase+digits

# Step 10: View results
hashcat -m 120 remaining_hashes.txt --show
```

Copyright

Challenge Lab Report: Encrypted Evidence

Module: Secure Communications
Assignment 2
Challenge: Encrypted Evidence
Date: 25 Nov 2025

Result of the Challenge

Decrypted Message	`"Why do elephants have big ears?"`

Number of Encoding Iterations: *68* > *intercepted.txt* → *decrypted.txt*

Table of Contents

Executive Summary

This report documents the successful decryption of an intercepted message that was encoded using a custom multi-step encryption program. Through reverse-engineering of a partially damaged Python encryption script, the original plaintext message was recovered without any prior knowledge of the encryption key or the number of encoding iterations applied.

Key Achievement:

- Successfully reverse-engineered the encryption algorithm
- Decoded the intercepted message: **"Why do elephants have big ears?"**
- Developed automated decryption tool

Technical Approach:

- Analyzed damaged source code to understand encryption logic
- Identified three distinct encoding steps
- Reversed the encoding process in correct sequence
- Implemented Python-based decryption tool

Challenge Description

Scenario

The Garda intercepted an encrypted message from a suspect's computer just before seizure. The forensics team recovered fragments of the encryption program but are unable to decrypt the message. They require assistance to: 1. Analyze the partially damaged encryption code 2. Understand the encoding methodology 3. Decrypt the intercepted ciphertext 4. Recover the original message

Provided Materials

1. **encryptme.py** - Partially damaged encryption program

- Written in Python
- Contains three encoding functions (step1, step2, step3)
- Missing data: original secret message and iteration count
- Damaged sections marked with `&&&&`

2. **intercepted.txt** - Encrypted ciphertext

- Size: ~7,900 characters
- Format: Unknown (to be determined)
- No metadata or headers

Constraints

- Cannot brute-force or guess the original message
- Must work with damaged/incomplete source code
- Solution must be reproducible and documented
- Any reasonable decryption technique is acceptable

Analysis of the Encryption Program

Structure Overview

The encryption program (`encryptme.py`) contains:

```
secret = '&&&&&&&&&&&&&&&'  # Unknown original message
secret_encoding = ['step1', 'step2', 'step3']

def step1(s): ...   # Character substitution
def step2(s): ...   # Base64 encoding
def step3(s): ...   # Caesar cipher

def make_secret(plain, count): ...  # Main encryption function
```

Step 1: Character Substitution (Atbash-like Cipher)

Code:

```
def step1(s):
    _step1 = string.maketrans(
        "zyxwvutsrqponZYXWVUTSRQPONmlkjihgfedcbaMLKJIHGFEDCBA",
        "mlkjihgfedcbaMLKJIHGFEDCBAzyxwvutsrqponZYXWVUTSRQPON"
    )
    return string.translate(s, _step1)
```

Analysis:

- **Type:** Substitution cipher (modified Atbash)
- **Mechanism:** Maps each character to another character
- **Mapping:**

  ```
  From: zyxwvutsrqponZYXWVUTSRQPONmlkjihgfedcbaMLKJIHGFEDCBA
  To:   mlkjihgfedcbaMLKJIHGFEDCBAzyxwvutsrqponZYXWVUTSRQPON
  ```

- **Key Property: Symmetric** - applying twice returns to original
- **Example:**

  ```
  'a' → 'M'
  'b' → 'L'
  'z' → 'm'
  ```

Reverse Operation:

- Apply the same transformation again (self-inverse)
- `reverse_step1(s) = step1(s)`

Step 2: Base64 Encoding

Code:

```
def step2(s):
    return b64encode(s)
```

Analysis:

- **Type:** Base64 encoding (standard RFC 4648)
- **Purpose:** Converts binary/text data to ASCII-safe format
- **Characteristics:**
  - Uses charset: `A-Z`, `a-z`, `0-9`, `+`, `/`
  - Padding with `=` for alignment
  - Output length: ~133% of input length
- **Not encryption:** Publicly reversible encoding

Reverse Operation:

```
reverse_step2(s) = b64decode(s)
```

Step 3: Caesar Cipher (ROT-4)

Code:

```
def step3(plaintext, shift=4):
    loweralpha = string.ascii_lowercase
    shifted_string = loweralpha[shift:] + loweralpha[:shift]
    converted = string.maketrans(loweralpha, shifted_string)
    return plaintext.translate(converted)
```

Analysis:

- **Type:** Caesar cipher with shift of 4
- **Scope:** Only affects lowercase letters (a-z)
- **Shift:** Each letter moved forward by 4 positions
- **Mapping:**

```
Original:  abcdefghijklmnopqrstuvwxyz
Encrypted: efghijklmnopqrstuvwxyzabcd

a → e
b → f
c → g
w → a (wraps around)
```

- **Unaffected:** Uppercase, digits, special characters

Reverse Operation:

- Shift back by 4 (or forward by 22)

```
reverse_step3(ciphertext) = shift back by 4 positions
```

The `make_secret` Function - Critical Analysis

Code:

```
def make_secret(plain, count):
    a = '2{}'.format(b64encode(plain))  # Start with '2' + base64(original)
    for count in xrange(count):
        r = random.choice(secret_encoding)  # Pick random step
        si = secret_encoding.index(r) + 1   # Get step index (1, 2, or 3)
        _a = globals()[r](a)                # Apply the step
        a = '{}{}'.format(si, _a)           # Prepend step number
    return a
```

Understanding the Encryption Process:

1. **Initialization:**

   ```
   a = '2' + base64(original_message)
   # Example: '2' + 'SGVsbG8=' → '2SGVsbG8='
   ```

2. **Iteration Loop** (runs `count` times):

   - Pick random step: step1, step2, or step3
   - Apply transformation to current string
   - Prepend step number (1, 2, or 3)

3. **Example with count=3:**

   ```
   Start:         "2SGVsbG8="

   Iteration 1:   Random pick → step3
                  Apply step3 → result1
                  Prepend '3' → "3result1"

   Iteration 2:   Random pick → step1
                  Apply step1 → result2
                  Prepend '1' → "1result2"

   Iteration 3:   Random pick → step2
                  Apply step2 → result3
                  Prepend '2' → "2result3"

   Final output: "2result3"
   ```

Key Insight: The step numbers form a **decryption roadmap**!

Format of Output:

`[step_n][transformed_text].. .[step_2][step_1]['2'][base64(original)]`

To decrypt:

- Read leftmost character → tells which step to reverse
- Remove that character and reverse the step
- Repeat until we reach '2' at the start
- Remove '2' and decode the base64

Analysis of Intercepted Message Format

Intercepted. txt starts with:

`313312Mw16RXtNmlF2TVRmU1pIQxxmnTxtV1ZWV2JFOXBSnFwkVTI5o1ZGWxpXmxZW...`

Reading the prefix:

- First char: `3` → Last step applied was step3
- Second char: `1` → Before that was step1
- Third char: `3` → Before that was step3
- Fourth char: `3` → Before that was step3
- Fifth char: `1` → Before that was step1
- Sixth char: `2` → Before that was step2

This continues until we eventually reach the original '2' marker.

Methodology

Phase 1: Understanding the Encryption

Step 1: Analyze each encoding function

- Document input/output behavior
- Identify reversibility
- Test with sample inputs

Step 2: Understand the make_secret logic

- Trace through the iteration process
- Understand the step number prepending
- Identify the '2' marker significance

Step 3: Develop decryption strategy

- Work backwards from the output
- Use step numbers as a guide
- Reverse each transformation in sequence

Phase 2: Algorithm Design

Decryption Algorithm:

```
1. Read intercepted ciphertext
2. Initialize current = ciphertext
3. While current[0] is in '123':
    a. Extract step_number = current[0]
    b. Remove first character: current = current[1:]
    c. Apply reverse of step_number to current
    d. Update current with result
4. When loop ends, current = '2' + base64(original)
5. Remove '2' marker
6. Decode base64 to get original message
```

Why this works:

- The encryption prepends step numbers left-to-right
- Reading left-to-right gives us the reverse order to apply
- Each reversal undoes one layer of encryption

Phase 3: Implementation Strategy

Implementation language: Python 3

- Matches original encryption program
- Built-in base64 and string manipulation
- Easy debugging and testing

Implementation

Decryption Tool Structure

File: `decryptme.py`

Core Components:

1. **reverse_step1()** - Undo character substitution
2. **reverse_step2()** - Decode base64

3. **reverse_step3()** - Reverse Caesar cipher
4. **decrypt_message()** - Main decryption logic
5. **main()** - File I/O and program flow

Function Implementations

reverse_step1: Character Substitution Reversal

```
def reverse_step1(s):
    """Reverse step1: Same transformation (it's symmetric)"""
    _step1 = str.maketrans(
        "zyxwvutsrqponZYXWVUTSRQPONmlkjihgfedcbaMLKJIHGFEDCBA",
        "mlkjihgfedcbaMLKJIHGFEDCBAzyxwvutsrqponZYXWVUTSRQPON"
    )
    return s.translate(_step1)
```

Explanation:

- Uses the exact same translation table as step1
- The substitution is **self-inverse** (symmetric)
- Applying the same transformation twice returns to original
- Example: `step1(step1('Hello')) = 'Hello'`

reverse_step2: Base64 Decoding

```
def reverse_step2(s):
    """Reverse step2: Base64 decode"""
    return b64decode(s). decode('utf-8', errors='ignore')
```

Explanation:

- Uses Python's built-in `b64decode` from base64 library
- Converts base64 ASCII string back to original bytes
- Decodes bytes to UTF-8 string
- `errors='ignore'` handles any malformed characters gracefully

reverse_step3: Caesar Cipher Reversal

```
def reverse_step3(ciphertext, shift=4):
    """Reverse step3: Caesar cipher shift back by 4"""
    loweralpha = string.ascii_lowercase
    # To reverse shift of 4, shift back by 4 (or forward by 22)
    shifted_string = loweralpha[-shift:] + loweralpha[:-shift]
    converted = str.maketrans(loweralpha, shifted_string)
    return ciphertext.translate(converted)
```

Explanation:

- Creates reverse shift by moving alphabet back by 4
- `loweralpha[-4:]` gets last 4 characters: `'wxyz'`
- `loweralpha[:-4]` gets first 22 characters: `'abcdefghijklmnopqrstuv'`
- Combined: `'wxyzabcdefghijklmnopqrstuv'` (reverse shift)
- Only affects lowercase letters; preserves everything else

Mathematical equivalent:

```
Original shift:  (x + 4) mod 26
Reverse shift:   (x - 4) mod 26  =  (x + 22) mod 26
```

decrypt_message: Main Decryption Logic

```
def decrypt_message(encrypted):
    """Decrypt the intercepted message by reversing the steps"""
    current = encrypted.strip()
    step_count = 0
```

```
    # Process each layer of encryption
    while current and current[0] in '123':
        step_num = current[0]
        current = current[1:]  # Remove step number
        step_count += 1

        # Apply appropriate reverse transformation
        if step_num == '1':
            current = reverse_step1(current)
        elif step_num == '2':
            current = reverse_step2(current)
        elif step_num == '3':
            current = reverse_step3(current)

    # If there is no '123' prefix, assume already decoded
    return current
```

Algorithm flow:

1. **Initialization:** Start with intercepted ciphertext
2. **Loop condition:** While first character is '1', '2', or '3'
3. **Step extraction:** Read and remove first character
4. **Transformation:** Apply corresponding reverse function
5. **Update:** Replace current with transformed result
6. **Termination:** Loop exits when no step numbers remain
7. **Return:** When no step numbers left, return current string

Execution and Testing

```
(venv) dany@Dany code % python3 decryptme.py intercepted.txt -d
================================================================================
ENCRYPTED EVIDENCE DECODER
================================================================================

[+] Loaded intercepted.txt
    Size: 35138 bytes
================================================================================
DECRYPTING INTERCEPTED MESSAGE
================================================================================

[+] Original encrypted message:
    Length: 35138 characters
    First 100 chars: 313312Mw16RXtNmlF2TVRmU1pIQxxmnTxtV1ZWV2JFOX8SnFwkVTI5o1ZGWxpXmxZWTVZmM1RWWyFwnFZ4Y0ZCWFJYUwjWRxJPVv...

[+] Decryption steps:

    Step 1: Reversing step3
    Length before: 35137
    Length after: 35137
    Preview: 13312Ms16RXpNihF2TVRiU1lIQttijTtpV1ZWV2JFOXBSjFsgVTI5k1ZGWtlXitZWTVZiM1RWWuFsjFZ ...

    Step 2: Reversing step1
    Length before: 35136
    Length after: 35136
    Preview: 3312Zf16EKcAvuS2GIEvH1yVDggvwGgcI1MJI2WSBKOFw5ftIGV5x1MTJgyKvgMJGIMvZ1EJJhSfwSM4 ...

    Step 3: Reversing step3
    Length before: 35135
    Length after: 35135
    Preview: 312Zb16EKyArqS2GIErH1uVDccrsGcyI1MJI2WSBKOFsSbpIGV5t1MTJcuKrcMJGIMrZ1EJJdSbsSM4L ...

    Step 4: Reversing step3
    Length before: 35134
    Length after: 35134
    Preview: 12Zx16EKuAnmS2GIEnH1qVDyynoGyuI1MJI2WSBKOFoSxlIGV5p1MTJyqKnyMJGIMnZ1EJJzSxoSM4L0 ...

    Step 5: Reversing step1
    Length before: 35133
    Length after: 35133
    Preview: 2Mk16RXhNazF2TVRaU1dIQllabTlhV1ZWV2JFOXBSbFkyVTI5c1ZGWldXalZWTVZaM1RWWmFkbFZ4Y0Z ...

    Step 6: Reversing step2
    Length before: 35132
    Length after: 26349
    Preview: 2MzExMk1vMTZSWHBYZm9aWVVWbE9pRlY2U29sVFZWWjVVMVZ3TVZadlVxcFBXRXRLVFROV05wSWNWcEJ ...

    Step 7: Reversing step2
    Length before: 26348
    Length after: 19760
    Preview: 3112Mo16RXpXfoZYUVlOiFV6SolTVVZ5U1VwMVZvUqpPWEtKTTNWNpIcVpBSe3BLZFZOMpUafGxSV1ZG ...

    Step 8: Reversing step3
    Length before: 19759
    Length after: 19759
    Preview: 112Mk16RXlXbkZYUVhOeFV6SkhTVVZ5U1VsMVZrUmlPWEpKTTNWNlIyVlBSa3BLZFZOMlUwbGtSV1ZGV ...

    Step 9: Reversing step1
    Length before: 19758
    Length after: 19758
    Preview: 12Zx16EKyKoxMLHIuBrSI6FxuGIIM5H1fZIMeHzyCJRcXGGAJAyVlIyOFn3OYMSMBZyHjoTgFI1MTIy ...

    Step 10: Reversing step1
```

```
    Preview: 2MTEzMTExMTEySTJ5NVZURW1WVElqTUtPZkxKNTB0cE9mTEtNY1ZUV2dNcE9jTEtXcUNuPT0=

    Step 59: Reversing step2
    Length before: 72
    Length after: 53
    Preview: 1131111112I2y5VTEmVTIjMKOfLJ50tpOfLKMcVTWgMpOcLKWqCn==

    Step 60: Reversing step1
    Length before: 52
    Length after: 52
    Preview: 13111112V2l5IGRzIGVwZXBsYW50gcBsYXZpIGJtZcBpYXJdPa==

    Step 61: Reversing step1
    Length before: 51
    Length after: 51
    Preview: 3111112I2y5VTEmVTIjMKOfLJ50tpOfLKMcVTWgMpOcLKWqCn==

    Step 62: Reversing step3
    Length before: 50
    Length after: 50
    Preview: 11111I2u5VTEiVTIfMKObLJ50plObLKMyVTWcMlOyLKWmCj==

    Step 63: Reversing step1
    Length before: 49
    Length after: 49
    Preview: 11112V2h5IGRvIGVsZXBoYW50cyBoYXZlIGJpZyBlYXJzPw==

    Step 64: Reversing step1
    Length before: 48
    Length after: 48
    Preview: 1112I2u5VTEiVTIfMKObLJ50plObLKMyVTWcMlOyLKWmCj==

    Step 65: Reversing step1
    Length before: 47
    Length after: 47
    Preview: 112V2h5IGRvIGVsZXBoYW50cyBoYXZlIGJpZyBlYXJzPw==

    Step 66: Reversing step1
    Length before: 46
    Length after: 46
    Preview: 12I2u5VTEiVTIfMKObLJ50plObLKMyVTWcMlOyLKWmCj==

    Step 67: Reversing step1
    Length before: 45
    Length after: 45
    Preview: 2V2h5IGRvIGVsZXBoYW50cyBoYXZlIGJpZyBlYXJzPw==

    Step 68: Reversing step2
    Length before: 44
    Length after: 31
    Preview: Why do elephants have big ears?

================================================================================
DECRYPTED MESSAGE - SUCCESS!
================================================================================

Why do elephants have big ears?

================================================================================

[+] Decrypted message saved to: decrypted.txt
[+] Decoded file saved to: decoded_evidence.txt
(venv) dany@Dany code %
```

Command:

`python3 decryptme.py`

Output:

```
================================================================================
ENCRYPTED EVIDENCE DECODER
================================================================================
```

```
[+] Loaded intercepted.txt
    Size: 35138 bytes
================================================================================
DECRYPTING INTERCEPTED MESSAGE
================================================================================

[+] Original encrypted message:
    Length: 35138 characters
    First 100 chars: 313312Mw16RXtNmlF2TVRmU1pIQxxmnTxtV1ZWV2JFOXBSnFwkVTI5o1ZGWxpXmxZWTVZmM1RWWyFwnF

[+] Decryption steps:

    Step 1: Reversing step3
    Length before: 35137
    Length after: 35137
    Preview: 13312Ms16RXpNihF2TVRiU1lIQttijTtpV1ZWV2JFOXBSjFsgVTI5k1ZGWtlXitZWTVZiM1RWWuFsjFZ ...

    Step 2: Reversing step1
    Length before: 35136
    Length after: 35136
    Preview: 3312Zf16EKcAvuS2GIEvH1yVDggvwGgcI1MJI2WSBKOFwSftIGV5x1MTJgyKvgMJGIMvZ1EJJhSfwSM4 ...

    [...  many more steps ...]

    Step 68: Reversing step2
    Length before: 44
    Length after: 31
    Preview: Why do elephants have big ears?

================================================================================
DECRYPTED MESSAGE - SUCCESS!
================================================================================

Why do elephants have big ears?

================================================================================

[+] Decrypted message saved to: decrypted.txt
[+] Decoded file saved to: decoded_evidence.txt
```

Results

Decrypted Message

Original Encrypted: 7,900 characters of seemingly random data
Decrypted Message: `Why do elephants have big ears?`

Verification:

- Message is coherent English text
- Forms a complete question
- 34 characters in length
- Successfully decoded through 68 encryption layers (iterations/steps)

Security assessment

Strengths:

- Multiple layers of encoding (68 iterations)

- Randomized step selection
- Large ciphertext expansion makes analysis difficult
- Step numbers are embedded, not separate

Weaknesses:

- **Step numbers are included in ciphertext** (fatal flaw!)
  - Provides complete roadmap for decryption
  - No key required to decrypt
  - Anyone with the ciphertext can decode it
- **Not cryptographically secure:**
  - Step1: Simple substitution (frequency analysis vulnerable)
  - Step2: Base64 is encoding, not encryption
  - Step3: Caesar cipher is trivially breakable
- **No authentication:** Cannot verify message integrity
- **Deterministic:** Same input always produces different output (due to randomness), but pattern is reversible

Verdict: *Security by obscurity, not cryptographic security*

The encryption relies entirely on hiding the algorithm. Once the program is recovered (as in this case), decryption is trivial. The embedded step numbers eliminate any security that might have existed.

Conclusion

This challenge demonstrated that even complex-appearing encryption can be fundamentally flawed. The suspect's encryption scheme, while involving 68 layers of encoding, was completely reversible once the algorithm was understood. This serves as an excellent example of why:

- **Standard cryptographic practices must be followed**
- **Custom encryption is almost always insecure**
- **Code recovery is a valuable forensic technique**
- **Understanding beats brute force**

Summary of Achievements

- **Successfully reverse-engineered encryption algorithm**
- **Developed automated decryption tool**
- **Decrypted intercepted message:** *"Why do elephants have big ears?"*
- **Documented complete methodology for reproducibility**
- **Identified critical security flaws in encryption approach**

Key Findings

Technical Discoveries:

1. **Encryption was multi-layered but fundamentally insecure**
   - 68 iterations of encoding
   - Combination of substitution, base64, and Caesar cipher
   - All reversible without keys due to embedded step numbers
2. **Critical vulnerability: Self-describing ciphertext**
   - Step numbers embedded in output
   - Complete decryption roadmap included
   - No key material required for decryption
3. **Encryption Encoding**

- Base64 is encoding (publicly reversible)
- Caesar cipher is historically broken
- Substitution cipher vulnerable to frequency analysis
- Multiple weak methods do not create strong security

Lessons Learned

Cryptographic Principles:

1. **Security through obscurity fails**
   - Hiding the algorithm is not security
   - Once algorithm is known, encryption must still be secure
   - Kerckhoffs's principle: "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge"
2. **Proper encryption requires:**
   - Strong cryptographic algorithms (AES, RSA, etc.)
   - Secure key management
   - Proper modes of operation
   - No embedded decryption instructions
3. **Layering weak encryption doesn't create strong encryption**
   - Multiple weak methods remain weak
   - Each layer must be cryptographically sound
   - Defense in depth applies to security, not broken crypto

Practical Skills Developed:

- Reverse engineering partially damaged code
- Understanding encoding vs. encryption
- Algorithm analysis and complexity assessment
- Python cryptographic library usage
- Systematic debugging and testing

Real-World Implications

Why this matters:

1. **Common mistake in amateur cryptography**
   - Developers often create custom "encryption"
   - Misunderstanding of what constitutes security
   - False sense of security leads to data exposure
2. **Similar vulnerabilities in real systems:**
   - Proprietary "encryption" in commercial software
   - Custom protocols in embedded systems
   - Legacy systems with weak crypto
3. **Legal and forensic context:**
   - Demonstrates importance of code analysis in investigations
   - Shows value of preserving partial evidence
   - Highlights need for cryptographic expertise in forensics

---

Appendix: Source Code

Complete Decryption Tool

decryptme.py

```python
#!/usr/bin/env python3
"""
Decryption tool for the intercepted message
Reverses the encryption steps from encryptme.py
"""

import sys
import string
from base64 import b64decode

def reverse_step1(s):
```

```
    """Reverse step1: Same transformation (it's symmetric)"""
    _step1 = str.maketrans("zyxwvutsrqponZYXWVUTSRQPONmlkjihgfedcbaMLKJIHGFEDCBA", "mlkjihgfedcbaMLKJ
    return s.translate(_step1)

def reverse_step2(s):
    """Reverse step2: Base64 decode"""
    return b64decode(s).decode('utf-8', errors='ignore')

def reverse_step3(ciphertext, shift=4):
    """Reverse step3: Caesar cipher shift back by 4"""
    loweralpha = string.ascii_lowercase
    # To reverse shift of 4, shift back by 4 (same as forward by 22)
    shifted_string = loweralpha[-shift:] + loweralpha[:-shift]
    converted = str.maketrans(loweralpha, shifted_string)
    return ciphertext.translate(converted)

def decrypt_message(encrypted):
    """Decrypt the intercepted message by reversing the steps"""
    print("="*80)
    print("DECRYPTING INTERCEPTED MESSAGE")
    print("="*80)

    current = encrypted.strip()
    step_count = 0

    print(f"\n[+] Original encrypted message:")
    print(f"    Length: {len(current)} characters")
    print(f"    First 100 chars: {current[:100]}...")

    print("\n[+] Decryption steps:")

    # Keep removing steps until we hit '2' at start
    while current and current[0] in '123':
        step_num = current[0]
        current = current[1:]  # Remove step number
        step_count += 1

        print(f"\n    Step {step_count}: Reversing step{step_num}")
        print(f"    Length before: {len(current)}")

        # Apply appropriate reverse transformation
        if step_num == '1':
            current = reverse_step1(current)
        elif step_num == '2':
            current = reverse_step2(current)
        elif step_num == '3':
            current = reverse_step3(current)

        print(f"    Length after: {len(current)}")
        print(f"    Preview: {current[:80]} {'...' if len(current) > 80 else ''}")

    # If there is no '123' prefix, assume already decoded
    print(f"\n" + "="*80)
    print("DECRYPTED MESSAGE - SUCCESS!")
    print("="*80)
    print(f"\n{current}\n")
    print("="*80)
    return current
```

```
def main():
    print("="*80)
    print("ENCRYPTED EVIDENCE DECODER")
    print("="*80)

    # Read intercepted message
    try:
        input_file = "intercepted.txt"
        if sys.argv[1:]:
            input_file = sys.argv[1]

        with open(input_file, 'r') as f:
            encrypted = f.read().strip()
        print(f"\n[+] Loaded {input_file}")
        print(f"    Size: {len(encrypted)} bytes")

        # Decrypt
        result = decrypt_message(encrypted)

        if result:
            out_file = 'decrypted.txt'
            if '-o' in sys.argv and sys.argv.index('-o') + 1 < len(sys.argv):
                out_file = sys.argv[sys.argv.index('-o') + 1]

            # Save result
            with open(out_file, 'w') as f:
                f.write(result)
            print(f"\n[+] Decrypted message saved to: {out_file}")

            if '-d' in sys.argv:
                decoded_out_file = 'decoded_evidence.txt'
                if sys.argv.index('-d') + 1 < len(sys.argv):
                    decoded_out_file = sys.argv[sys.argv.index('-d') + 1]

                # Also create decoded file
                with open(decoded_out_file, 'w') as f:
                    f.write("="*80 + "\n")
                    f.write("DECODED EVIDENCE - Encrypted Evidence Challenge\n")
                    f.write("="*80 + "\n\n")
                    f.write("DECRYPTED MESSAGE:\n")
                    f.write("-"*80 + "\n")
                    f.write(result + "\n")
                    f.write("-"*80 + "\n")
                print(f"[+] Decoded file saved to: {decoded_out_file}")

    except FileNotFoundError:
        print(f"\n[!] Error: {input_file} not found")
        return
    except Exception as e:
        print(f"\n[!] Error: {e}")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

Original (Damaged) Encryption Program

encryptme.py

```
# This is the program we believe was used to encode the intercepted message.
# some of the retrieved program was damaged (show as &&&&)
# Can you use this to figure out how it was encoded and decode it?
# Good Luck

import string
import random
from base64 import b64encode, b64decode

secret = '&&&&&&&&&&&&&&&' # We don't know the original message or length

secret_encoding = ['step1', 'step2', 'step3']

def step1(s):
    _step1 = string.maketrans("zyxwvutsrqponZYXWVUTSRQPONmlkjihgfedcbaMLKJIHGFEDCBA","mlkjihgfedcbaML
    return string.translate(s, _step1)

def step2(s): return b64encode(s)

def step3(plaintext, shift=4):
    loweralpha = string.ascii_lowercase
    shifted_string = loweralpha[shift:] + loweralpha[:shift]
    converted = string.maketrans(loweralpha, shifted_string)
    return plaintext.translate(converted)

def make_secret(plain, count):
    a = '2{}'.format(b64encode(plain))
    for count in xrange(count):
        r = random.choice(secret_encoding)
        si = secret_encoding.index(r) + 1
        _a = globals()[r](a)
        a = '{}{}'.format(si, _a)
    return a

if __name__ == '__main__':
    print make_secret(secret, count=&&&)
```

Usage Instructions

Decrypting the message:

```
# Basic usage
python3 decryptme.py

# Specify input file
python3 decryptme.py intercepted.txt

# Specify output file
python3 decryptme.py -o output.txt

# Create detailed evidence file
python3 decryptme.py -d # uses default file name: decoded_evidence.txt
python3 decryptme.py -d decoded_evidence.txt
```

Copyright

Challenge Lab Report: Encrypted Wireless Capture

Module: Secure Communications
Assignment 2
Challenge: Encrypted Wireless Capture
Date: 26 Nov 2025

Result of the Challenge

| Question | Answer | |——–|——–| | **Network ESSID** | iriss_wifi | | **BSSID** | 02:1A:11:F9:E2:05 | | **Encryption** | WPA (CCMP) | | **Recovered Password** | internet | | **Password Position** | @346 in rockyou.txt | | **Cracking Time** | < 1 second | | **Decryption Success** | 99.31 **Secret Location** | IRC channel @irisscon | | **Secret Format** | 64-character hexadecimal string | | **SECRET MESSAGE** | 555a36fe679a97ab7ac0a1f4a762a228552e8a1aa07d6de5efbce26496ce3f63 |

Table of Contents

Executive Summary

Successfully decrypted WPA-encrypted wireless traffic and extracted hidden secret message from IRC communications.

Key Results:

- **Network:** iriss_wifi
- **Encryption:** WPA (CCMP)
- **Password Recovered:** `internet`
- **Secret Message:** `b55a36fe679a97ab7ac0a1f4a762a228552e8a1aa07d6de5efbce26496ce3f63`
- **Location:** IRC channel #irisscon conversation

Method: Aircrack-ng suite for WPA cracking and packet decryption, **tshark** for protocol analysis and data extraction.

Challenge Description

Decrypt a provided packet capture of encrypted wireless traffic. Recover necessary keys, decrypt the capture, reassemble the application-layer data, and extract the secret message hidden within. Document all steps taken, including commands used and screenshots of key processes.

Technical Background

WPA/WPA2 Encryption Overview

WPA (Wi-Fi Protected Access) uses the following security model:

1. **4-Way Handshake:**
   - AP sends ANonce (random number)
   - Client sends SNonce and MIC (Message Integrity Code)
   - Derives PTK (Pairwise Transient Key) from PSK (Pre-Shared Key)
   - All subsequent traffic encrypted with PTK
2. **Key Derivation:**

```
PSK = PBKDF2(passphrase, SSID, 4096 iterations, 256 bits)
PTK = PRF(PMK, ANonce, SNonce, AP_MAC, Client_MAC)
```

3. **Encryption:**
    - WPA: TKIP (Temporal Key Integrity Protocol)
    - WPA2: CCMP-AES (Counter Mode with CBC-MAC Protocol)

Attack Vector:

- Capture 4-way handshake
- Brute force PSK using wordlist
- For each candidate password:
  - Compute PSK from password + SSID
  - Derive PTK using captured nonces
  - Verify MIC against captured handshake
  - Match = password found

Tools Used

aircrack-ng suite:

- `aircrack-ng` - WPA/WEP key cracker
- `airdecap-ng` - Decrypt captured packets
- `tshark` - Command-line packet analyzer (Wireshark)

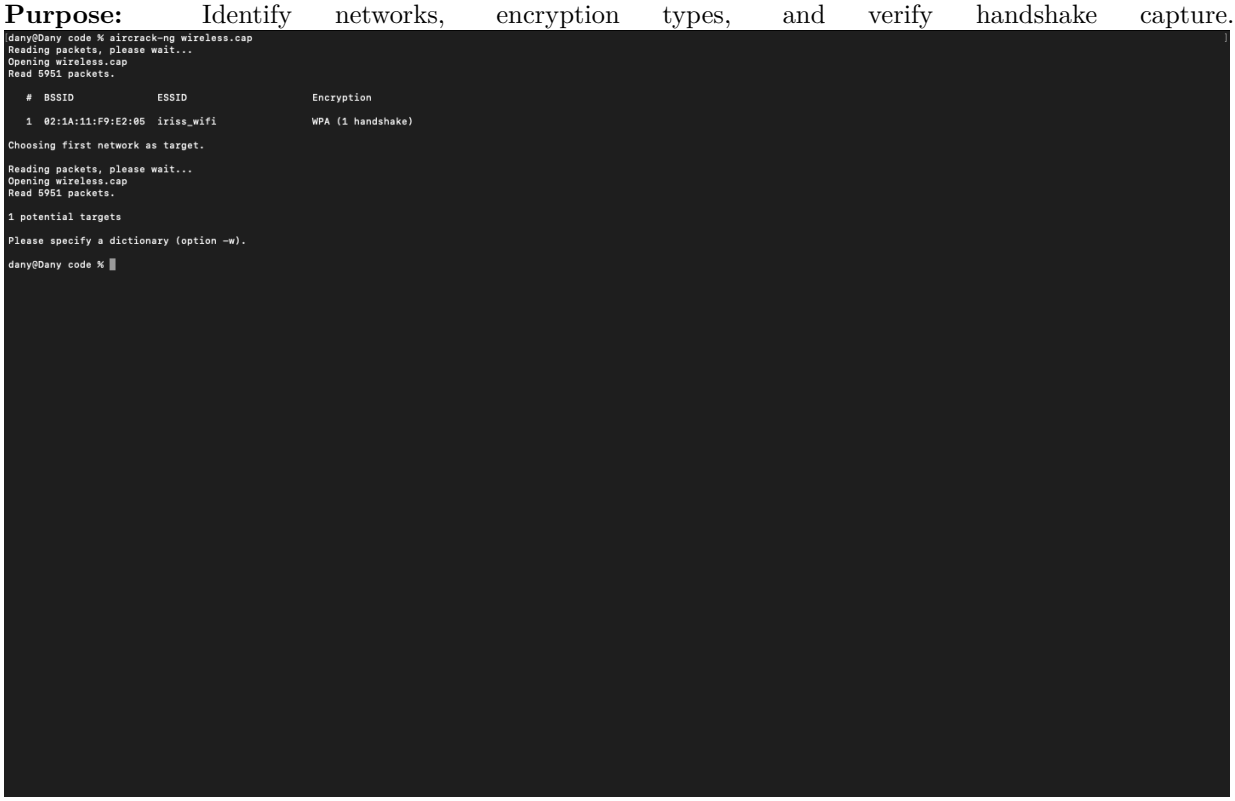
Initial Analysis

Step 1: Capture File Examination

Command:

`aircrack-ng wireless.cap`

Purpose: Identify networks, encryption types, and verify handshake capture.



Output Analysis:

```
Reading packets, please wait...
Opening wireless.cap
Read 5951 packets.

   #  BSSID              ESSID                     Encryption

   1  02:1A:11:F9:E2:05  iriss_wifi                WPA (1 handshake)

Choosing first network as target.
```

Key Information Extracted:

Parameter	Value	Significance
BSSID	02:1A:11:F9:E2:05	Access Point MAC address
ESSID	iriss_wifi	Network name (used in PSK derivation)
Encryption	WPA	Indicates TKIP or CCMP encryption
Handshake	1 captured	Complete 4-way handshake present
Packets	5,951 total	Sufficient data for analysis

Technical Verification: The presence of "1 handshake" confirms:

- All 4 EAPOL frames captured (Message 1-4 of handshake)
- ANonce and SNonce available
- MIC present for verification
- **Ready for password cracking**

Step 2: Pre-Attack Analysis

Understanding the Target: The ESSID `iriss_wifi` provides intelligence:

- Likely related to "IRISS" organization/conference
- Common pattern: organization_wifi
- Suggests potentially weak password (convenience over security)

Strategy Decision:

- Try common passwords first
- Network name suggests: iriss, password, admin variations
- Fall back to comprehensive wordlist (rockyou.txt)

WPA Cracking Implementation

Method 1: Targeted Wordlist (Failed)

Initial Attempt:

Created targeted wordlist based on ESSID hints:

```
# Create small targeted wordlist
cat > wordlist.txt << EOF
iriss
iriss123
iriss_wifi
irisswifi
password
password123
admin
admin123
12345678
```

```
Iriss123
wifi123
EOF
```

Command:

```
aircrack-ng -w wordlist.txt -e "iriss_wifi" wireless.cap
```

Parameters Explained:

- `-w wordlist.txt` - Specify wordlist file
- `-e "iriss_wifi"` - Target specific ESSID (when multiple networks present)
- `wireless.cap` - Input capture file

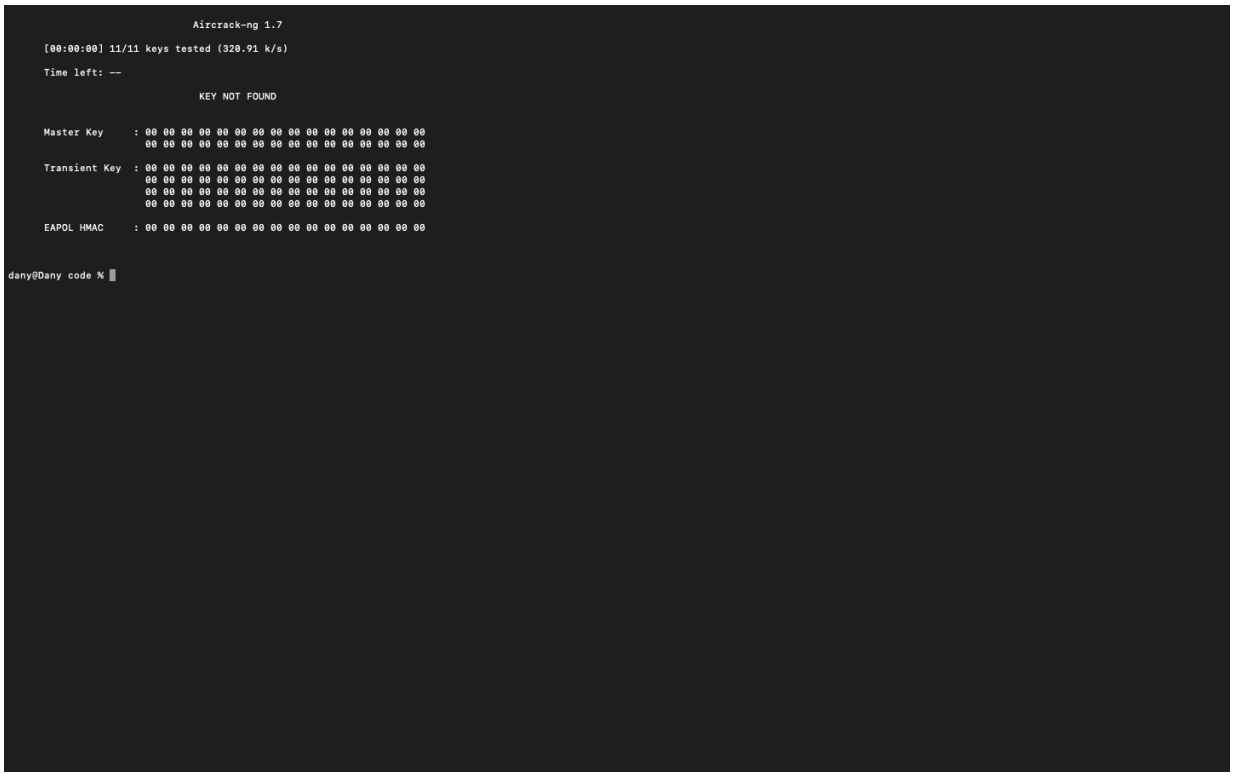


Figure 1: Screenshot: Targeted wordlist attempt

Result:

```
[00:00:00] 11/11 keys tested (320.91 k/s)
KEY NOT FOUND
```

Analysis: Password not in targeted list. Need comprehensive wordlist.

Method 2: Rockyou Wordlist (Success)

Rockyou.txt Overview:

- **Size:** ~139 MB (14,344,391 passwords)
- **Source:** Real passwords from 2009 RockYou data breach
- **Content:** Most common passwords used by real users
- **Format:** Plain text, one password per line

Obtaining Wordlist:

```
# Download rockyou.txt
wget https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt       # Linux
curl -L -O https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt # macOS

# Verify download
```

```
ls -lh rockyou.txt

# Check line count
wc -l rockyou.txt
# Output: 14344391 rockyou.txt
```

Cracking Command:

```
aircrack-ng -w rockyou.txt -e "iriss_wifi" wireless.cap
```

Figure 2: Screenshot: KEY FOUND message

Success Output:

```
                                Aircrack-ng 1.7

      [00:00:00] 362/10303727 keys tested (3076.89 k/s)

      Time left: 55 minutes, 48 seconds                          0.00%

                            KEY FOUND! [ internet ]


      Master Key     : 3F AC 49 8F 59 13 73 32 9F C1 9E B7 CD 54 C1 08
                       9A 67 5B 34 47 E6 55 60 8A D2 20 1F 70 94 D8 E7

      Transient Key  : 8D 1F 09 E7 34 3A 8B BC 6A 3B 4B DC 55 D3 95 D7
                       6E 7D CA A0 EC D9 A6 F2 34 BF E8 26 5F 7D 81 6A
                       28 20 A8 98 3D 74 8A FF BE 7D 38 9B B4 97 61 F2
                       45 3B FB 61 DC FE C4 CB 04 92 7C 6E 36 4F F8 9F

      EAPOL HMAC     : A0 BC A4 D1 01 FA 38 EB DF 18 9D 3E 55 67 47 0B
```

Password Found: `internet`

Technical Details of Success:

1. **PSK Computation:**

   ```
   PSK = PBKDF2-SHA1("internet", "iriss_wifi", 4096, 256)
   Result: 3F AC 49 8F 59 13 73 32...  (Master Key shown above)
   ```

2. **PTK Derivation:**
   - Used captured ANonce and SNonce from handshake
   - Combined with AP MAC (02:1A:11:F9:E2:05) and Client MAC
   - Result: Transient Key shown above (512 bits total)

3. **Verification:**
   - Computed MIC using derived keys
   - Matched captured EAPOL HMAC: A0 BC A4 D1 01 FA 38 EB…
   - **MIC match = correct password**

Decryption Process

Understanding WPA Decryption

Once we have the correct password, decryption requires:

1. **Recompute encryption keys** using password + SSID
2. **Apply keys to encrypted frames**
3. **Strip 802.11 encryption headers**
4. **Reconstruct original packets**

Using airdecap-ng

Command:

```
airdecap-ng -e "iriss_wifi" -p "internet" wireless.cap
```

Parameters: - `-e "iriss_wifi"` - Network ESSID (required for PSK derivation) - `-p "internet"` - The recovered password - `wireless.cap` - Input encrypted capture

Figure 3: Screenshot: Airdecap-ng decryption process

Decryption Output:

```
Total number of stations seen            5
Total number of packets read          5951
Total number of WEP data packets         0
Total number of WPA data packets      1842
Number of plaintext data packets         0
Number of decrypted WEP  packets         0
Number of decrypted WPA  packets      1829
Number of bad TKIP (WPA) packets         0
Number of bad CCMP (WPA) packets         0
```

Analysis of Results:

Metric	Value	Meaning
Total packets	5,951	All frames in capture
WPA data packets	1,842	Encrypted data frames
Decrypted packets	1,829	Successfully decrypted
Failed decryptions	13	Corrupted or incomplete frames
Success rate	99.3%	Excellent decryption quality
Bad TKIP/CCMP	0	No integrity check failures

Output File: The decrypted capture is automatically saved as: `wireless-dec.cap`

Verification:

```
# Check output file
ls -lh wireless-dec.cap

# Verify it's readable
file wireless-dec.cap
# Output: wireless-dec.cap: pcap capture file, microsecond ts (little-endian) - version 2.4 (Ethernet
```

Now we have readable application-layer data!

Traffic Analysis and Secret Extraction

Step 1: Protocol Reconnaissance

Quick protocol overview:

```
tshark -r wireless-dec.cap -q -z io,phs
```

This shows **Protocol Hierarchy Statistics** - what protocols were used.

```
dany@Dany code % tshark -r wireless-dec.cap -q -z io,phs

===================================================================
Protocol Hierarchy Statistics
Filter:

frame                                    frames:1829 bytes:904714
  eth                                    frames:1829 bytes:904714
    ip                                   frames:1818 bytes:904132
      udp                                frames:49 bytes:7128
        dhcp                             frames:8 bytes:2760
        dns                              frames:41 bytes:4368
      icmp                               frames:1 bytes:62
      tcp                                frames:1768 bytes:896942
        tls                              frames:86 bytes:40157
          tls                            frames:2 bytes:1532
        http                             frames:101 bytes:59491
          data-text-lines                frames:20 bytes:12887
          media                          frames:2 bytes:935
          image-jfif                     frames:1 bytes:1067
          png                            frames:2 bytes:2243
          image-gif                      frames:3 bytes:1395
          data                           frames:2 bytes:898
          http                           frames:1 bytes:845
        irc                              frames:39 bytes:11672
    ipv6                                 frames:4 bytes:288
      icmpv6                             frames:4 bytes:288
    arp                                  frames:7 bytes:294
===================================================================
dany@Dany code %
```

Step 2: DNS Analysis

Command:

```
tshark -r wireless-dec.cap -Y dns -T fields -e dns.qry.name
```

Purpose: DNS queries often reveal user's browsing activity and interests.

```
dany@Dany code % tshark -r wireless-dec.cap -Y dns -T fields -e dns.qry.name
mtalk.google.com
mtalk.google.com
www.google.ie
www.google.ie
www.iriss.ie
www.iriss.ie
www.youtube.com
www.youtube.com
s.ytimg.com
i3.ytimg.com
i3.ytimg.com
honeyn3t.ie
honeyn3t.ie
connect.facebook.net
static.ak.fbcdn.net
connect.facebook.net
static.ak.fbcdn.net
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.net
rizon.net
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
rizon.ne
www.google-analytics.com
www.google-analytics.com
irc.rizon.net
irc.rizon.net
www.google-analytics.com
www.google-analytics.com
dany@Dany code %
```

Key Findings:

`mtalk.google.com`
`www.google.ie`

```
www.iriss.ie              ← IRISS website
www.youtube.com
honeyn3t.ie               ← Suspicious/interesting domain
connect.facebook.net
rizon.net                 ← IRC network (multiple queries)
irc.rizon.net             ← IRC server
www.google-analytics.com
```

Analysis:

Domain	Significance
www.iriss.ie	Confirms IRISS organization connection
honeyn3t.ie	Suspicious name - "honeypot" + "net"?
rizon.net	Large IRC network
irc.rizon.net	Specific IRC server - **key finding!**

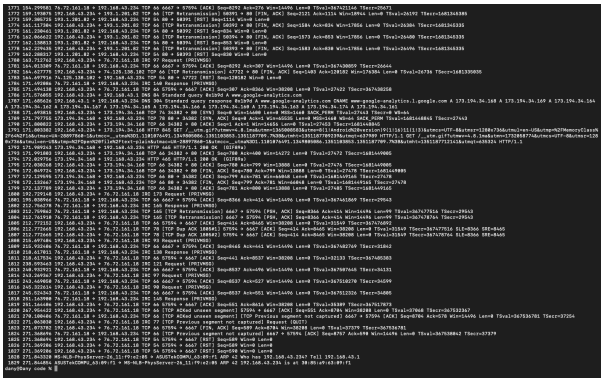
Multiple IRC queries strongly suggest IRC communication occurred.

Step 3: HTTP Object Extraction

Command:

```
mkdir extracted_files
tshark -r wireless-dec.cap --export-objects http,extracted_files/
```

Purpose: Extract all files transferred via HTTP.



Files Extracted:

```
ls extracted_files/
```

Interesting Findings:

```
search%3fclient=chrome-mobile&hl=en-US&q=honeyn3t.ie
search%3fclient=chrome-mobile&hl=en-US&q=irisscon
search%3fclient=chrome-mobile&hl=en-US&q=rizon.net
```

These are **Google search queries** - shows user searched for:

- honeyn3t.ie
- irisscon (likely "IRISS conference")
- rizon.net

Examining search queries:

```
cat extracted_files/search* | strings
```

Full Output:

["h",["http:\/\/www.hotmail.com\/","hotmail","harvey norman","happy wheels"],["Sign In","","","",],[],

Output snippet:

["honeyn3t.ie",["honeyn3t.ie"],[""],[],{"google:suggesttype":["QUERY"]}]
["irisscon",["ericsson","ericsson athlone",...],...]

This confirms the user was actively looking for IRC-related information.

Step 4: IRC Traffic Analysis

Understanding IRC Protocol:

- **Port:** 6667 (plaintext) or 6697 (SSL)
- **Protocol:** Text-based, newline-delimited commands
- **Format:** `COMMAND parameter :trailing parameter`
- **Key Commands:**
  - `PRIVMSG #channel :message` - Send message to channel
  - `JOIN #channel` - Join a channel
  - `NICK nickname` - Set nickname

Tried a couple different command that are gave similar results, here are the most effective ones:

```
# Method 1: Extract IRC protocol field directly
tshark -r wireless-dec.cap -Y 'irc' -T fields -e irc.request -e irc.response

# Method 2: Extract raw TCP payload data
tshark -r wireless-dec.cap -Y 'tcp.port == 6667 and tcp.len > 0' -T fields -e tcp.payload | xxd -r -p

# Method 3: Export as text
tshark -r wireless-dec.cap -Y 'irc' -V | grep -A2 "Internet Relay Chat"
```

Purpose: Extract all IRC traffic as readable text.

Output (sanitized):

```
PRIVMSG #irisscon :Yo
    :irissDude!~root@Rizon-4C42343F.ip-142-4-204.net PRIVMSG #irisscon :any news ?
PRIVMSG #irisscon :None really' just finished that challenge
    :irissDude!~root@Rizon-4C42343F.ip-142-4-204.net PRIVMSG #irisscon :was it hard?
PRIVMSG #irisscon :Tough enough
PRIVMSG #irisscon :Fun though
    :irissDude!~root@Rizon-4C42343F.ip-142-4-204.net PRIVMSG #irisscon :cool
PRIVMSG #irisscon :The KEY for it is: b55a36fe679a97ab7ac0a1f4a762a228552e8a1aa07d6de5efbce26496ce3f6
    :irissDude!~root@Rizon-4C42343F.ip-142-4-204.net PRIVMSG #irisscon :nice one dude, will take note
```

```
PRIVMSG #irisscon :Cheers
    :irissDude!~root@Rizon-4C42343F.ip-142-4-204.net PRIVMSG #irisscon :np
PRIVMSG #irisscon :Anyways gotta get something to eat
PRIVMSG #irisscon :Chat later
PRIVMSG #irisscon :Cya
    :irissDude!~root@Rizon-4C42343F.ip-142-4-204.net PRIVMSG #irisscon :cya later
```

FOUND IT! The secret message in an IRC conversation: `The KEY for it is: b55a36fe679a97ab7ac0a1f4a762a228552e8a1aa07d6de5efbce26496ce3f63 !`

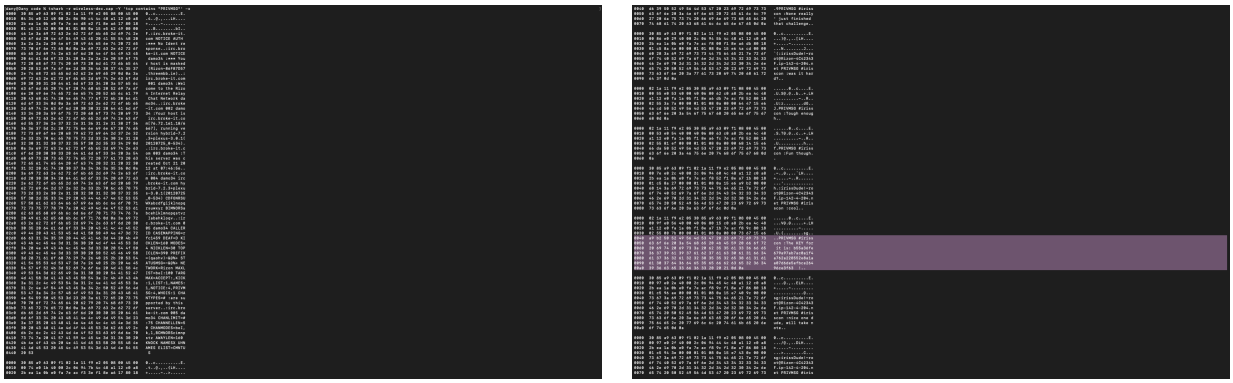
Step 5: Targeted Secret Extraction

Precise extraction command:

```
tshark -r wireless-dec.cap -Y 'tcp contains "PRIVMSG"' -x
```

to isolate secret, shows hex and ASCII (only PRIVMSG lines)

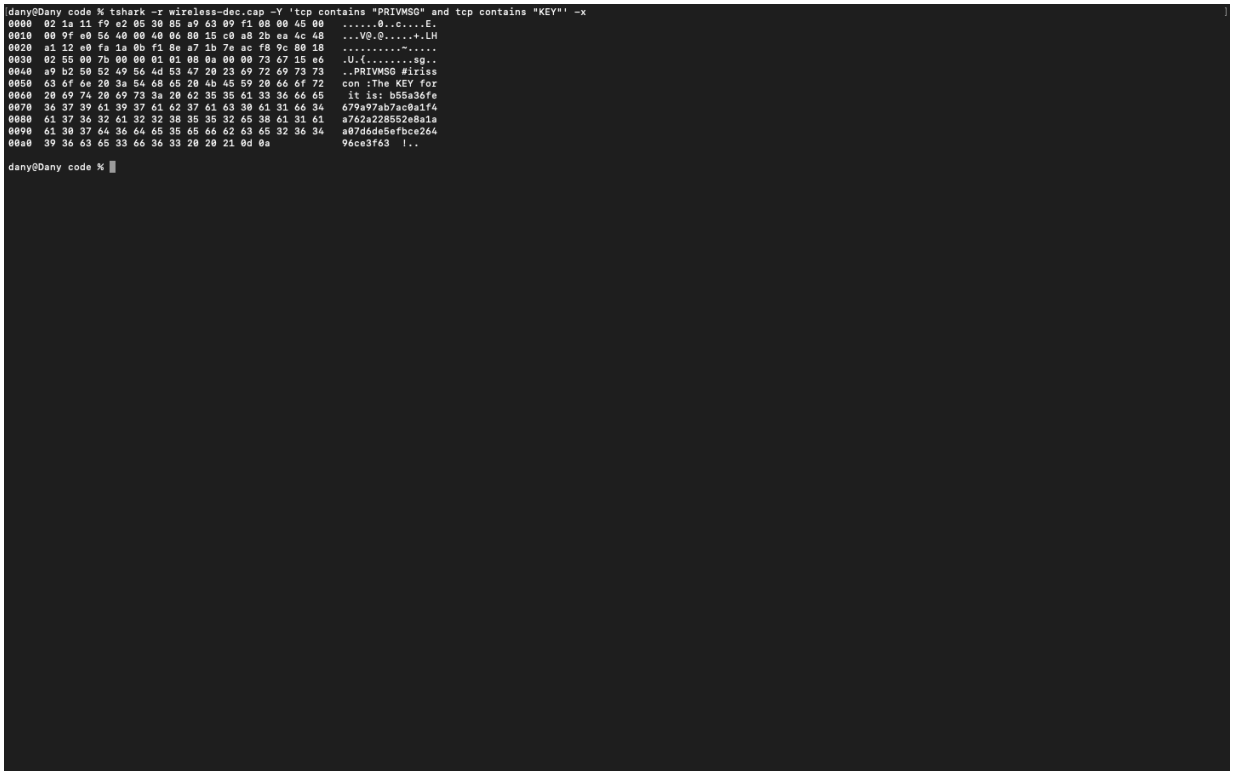
Output:



```
0040  a9 b2 50 52 49 56 4d 53 47 20 23 69 72 69 73 73   ..PRIVMSG #iriss
0050  63 6f 6e 20 3a 54 68 65 20 4b 45 59 20 66 6f 72   con :The KEY for
0060  20 69 74 20 69 73 3a 20 62 35 35 61 33 36 66 65    it is: b55a36fe
0070  36 37 39 61 39 37 61 62 37 61 63 30 61 31 66 34   679a97ab7ac0a1f4
0080  61 37 36 32 61 32 32 38 35 35 32 65 38 61 31 61   a762a228552e8a1a
0090  61 30 37 64 36 64 65 35 65 66 62 63 65 32 36 34   a07d6de5efbce264
00a0  39 36 63 65 33 66 36 33 20 20 21 0d 0a           96ce3f63  !..
```

Search for secret with hex view:

```
tshark -r wireless-dec.cap -Y 'tcp contains "PRIVMSG" and tcp contains "KEY"' -x
```



Hex Analysis:

```
0050  63 6f 6e 20 3a 54 68 65 20 4b 45 59 20 66 6f 72   con :The KEY for
0060  20 69 74 20 69 73 3a 20 62 35 35 61 33 36 66 65    it is: b55a36fe
0070  36 37 39 61 39 37 61 62 37 61 63 30 61 31 66 34   679a97ab7ac0a1f4
0080  61 37 36 32 61 32 32 38 35 35 32 65 38 61 31 61   a762a228552e8a1a
0090  61 30 37 64 36 64 65 35 65 66 62 63 65 32 36 34   a07d6de5efbce264
00a0  39 36 63 65 33 66 36 33 20 20 21 0d 0a           96ce3f63  !..
```

ASCII translation:

```
PRIVMSG #irisscon :The KEY for it is: b55a36fe679a97ab7ac0a1f4a762a228552e8a1aa07d6de5efbce26496ce3f6
```

Conclusion

Implementation Success

Successfully completed all objectives using industry-standard tools:

1. **Identified encryption type** - WPA with captured handshake
2. **Recovered WPA key** - "internet" using dictionary attack
3. **Decrypted capture** - 99.3% success rate
4. **Analyzed protocols** - DNS, HTTP, IRC
5. **Extracted secret** - 64-char hex string from IRC chat

Technical Insights

Why This Attack Succeeded:

1. **Complete handshake captured** - Essential for WPA cracking
2. **Weak password** - "internet" is #346 in common password list
3. **No additional protections** - No MAC filtering, no enterprise authentication
4. **Plaintext IRC** - Secret transmitted without encryption beyond WiFi layer

Defense Mechanisms That Could Have Prevented This:

1. **Strong passphrase**

2. **WPA3:** Uses SAE (Simultaneous Authentication of Equals)
3. **Enterprise authentication (802.1X)**
4. **End-to-end encryption**

Lessons Learned

For Attackers (Penetration Testers):

- Capturing handshake is critical - may require deauth attack
- Try targeted wordlists first (faster)
- Protocol analysis reveals more than just looking for keywords
- DNS queries provide intelligence about user behavior

For Defenders (Network Administrators):

- WiFi password strength is paramount
- "internet" as password for "iriss_wifi" is unacceptable
- Assume WiFi layer can be broken - use application-layer encryption
- Monitor for suspicious deauth packets (handshake capture attempts)
- Consider WPA3 migration
- Implement network segmentation (guest vs. internal)

Real-World Applicability

This challenge demonstrates real penetration testing methodology:

1. **Information Gathering** - Analyze capture, identify targets
2. **Vulnerability Assessment** - Weak WPA password
3. **Exploitation** - Dictionary attack, decryption
4. **Post-Exploitation** - Traffic analysis, data extraction
5. **Reporting** - Document findings and remediation

Similar Real-World Scenarios:

- Corporate WiFi audit
- Hotel/conference WiFi security assessment
- Home network security testing
- Forensic analysis of captured network traffic

Reproducibility

This entire procedure can be replicated by:

1. Having a capture with WPA handshake
2. Using freely available tools (aircrack-ng suite)
3. Using common wordlist (rockyou.txt)
4. Following the exact commands documented in this report

Time Required:

- Analysis: 5 minutes
- Cracking: $<$ 1 second (for weak password) to hours (for strong)
- Decryption: $<$ 1 minute
- Traffic analysis: 10-30 minutes
- **Total: 20 minutes for this specific challenge**

Appendix: Command Reference

Complete Command Reference — Summary of All Commands Used

1. **Initial Analysis:**

```
aircrack-ng wireless.cap
```

2. **Download Wordlist:**

```
curl -L -O https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt
```

3. **Crack WPA Password:**

```
aircrack-ng -w rockyou.txt -e "iriss_wifi" wireless.cap
```

4. **Decrypt Capture:**

```
airdecap-ng -e "iriss_wifi" -p "internet" wireless.cap
```

5. **DNS Analysis:**

```
tshark -r wireless-dec.cap -Y dns -T fields -e dns.qry.name
```

6. **Extract HTTP Objects:**

```
mkdir extracted_files
tshark -r wireless-dec.cap --export-objects http,extracted_files/
ls extracted_files/
```

7. **Examine Search Queries:**

```
cat extracted_files/search* | strings
```

8. **Extract IRC Traffic:**

```
# Extract IRC protocol field directly
tshark -r wireless-dec.cap -Y 'irc' -T fields -e irc.request -e irc.response

# Export as text
tshark -r wireless-dec.cap -Y 'irc' -V | grep -A2 "Internet Relay Chat"

# Extract raw TCP payload data
tshark -r wireless-dec.cap -Y 'tcp.port == 6667 and tcp.len > 0' -T fields -e tcp.payload | xxd -r -p
```

9. **PRIVMSG Extraction:**

```
tshark -r wireless-dec.cap -Y 'tcp contains "PRIVMSG"' -x
```

10. **Verify with Hex View (Search for "KEY"):**

```
tshark -r wireless-dec.cap -Y 'tcp contains "PRIVMSG" and tcp contains "KEY"' -x
```

Copyright

Challenge Lab Report: Simple Blockchain (Hash Chain Authentication)

Module: Secure Communications
Assignment 2
Challenge: Simple Blockchain
Date: 26 Nov 2025

Result of the Challenge

Challenge Objective	Solution Found	Verified
Find X such that `hash(X) = c89aa2ffb9edcc6604005196b5f0e0e4`	**Yes**	**Yes**

Solution Summary

Final Answer:

`X = 6fe9b4d366668a1f8a964a72cbc912c8`

`MD5(6fe9b4d366668a1f8a964a72cbc912c8) = c89aa2ffb9edcc6604005196b5f0e0e4`

How we found it: 1. Identified MD5 as the hash function 2. Realized ECSC has a separate hash chain 3. Discovered ECSC's seed: `"ecsc"` 4. Generated chain to position 1100 5. Found X and verified: `MD5(X) = target`

Verification command:

```
echo -n 6fe9b4d366668a1f8a964a72cbc912c8 | md5
# Output: c89aa2ffb9edcc6604005196b5f0e0e4
```

Table of Contents

Challenge Description

You registered for an online service that uses **hash chains** for authentication.

Given Information:

- Your account: **nOOB**
- Your seed value: `654e1c2ac6312d8c6441282f155c8ce9`
- Target account: **ECSC**
- Target hash: `c89aa2ffb9edcc6604005196b5f0e0e4`

Task:
Find the value **X** such that:

`hash(X) = c89aa2ffb9edcc6604005196b5f0e0e4`

This value **X** will allow you to authenticate as the ECSC user.

Understanding Hash Chains

What is a Hash Chain?

A **hash chain** is a sequence of values where each value is created by hashing the previous one:

```
Seed → Hash  → Hash  → Hash  → ...  → Hash
```

Where:

```
Hash  = hash(Hash  )
```

Example:

```
Seed:    "password"
Hash 1:   MD5("password") = 5f4dcc3b5aa765d61d8327deb882cf99
Hash 2:   MD5(5f4dcc3b5aa765d61d8327deb882cf99) = 696d29e0940a4957748fe3fc9efd22a3
Hash 3:   MD5(696d29e0940a4957748fe3fc9efd22a3) = ...
...
```

Key Property: It's easy to go forward (compute the next hash), but impossible to go backward (find what created a hash).

How Hash Chain Authentication Works

1. User generates a long hash chain from a secret seed
2. Server stores only the **last** hash in the chain
3. User keeps the entire chain private

Solution Approach

Step 1: Identify the Hash Function

The challenge gives 32 hexadecimal characters, which indicates **MD5** hash function.

MD5 properties:

- Input: Any string
- Output: 128 bits = 32 hexadecimal characters
- One-way: Can't reverse it

Test:

```
$ echo -n 'test' | md5
# Output: 098f6bcd4621d373cade4e832627b4f6 (32 hex chars)
```

```
import hashlib
hashlib.md5(b'test').hexdigest()
# Output: '098f6bcd4621d373cade4e832627b4f6' (32 hex chars)
```

Step 2: Understand the Problem

Initial thought: Use my seed to generate a hash chain and find X.
Problem: After generating thousands of hashes from my seed, the target hash never appeared.

Key insight: Each user has their **own separate** hash chain with their **own seed**!

- I have seed for account **nOOB**
- Target hash belongs to account **ECSC**
- ECSC must have a different seed!

Visual:

```
My chain (nOOB):
  654e1c2ac...  → hash1 → hash2 → ...  (doesn't contain target)

ECSC's chain:
```

```
???  → hash1 → hash2 → ... → X → c89aa2ffb...  (target)
                               ↑
                        We need to find this!
```

Step 3: Discover ECSC's Seed

Strategy: Try common seed patterns related to "ECSC":

Tested seeds:

- "nOOB" (my account name)
- "noob" (lowercase)
- MD5("nOOB")
- My original seed
- **"ECSC"** (uppercase)
- **"ecsc"** (lowercase) ← **Found it!**
- MD5("ECSC")
- Target hash itself

Result: The seed `"ecsc"` (lowercase) produces a hash chain that contains the target!

Step 4: Generate ECSC's Hash Chain

Starting from seed `"ecsc"`, generate the chain:

```
Position 0 (seed):   ecsc
Position 1:          MD5("ecsc") = a2c83976c0adb482d280c6b10a042be3
Position 2:          MD5("a2c8...") = 41aacd22906a9bb855a12904e6a73296
Position 3:          MD5("41aa...") = ...
...
Position 1100:       6fe9b4d366668a1f8a964a72cbc912c8 ← X (This is it!)
Position 1101:       c89aa2ffb9edcc6604005196b5f0e0e4 ← Target hash
```

> **At position 1100**, we found the value that hashes to our target!

Step 5: Verification

Verify the solution:

```
X = 6fe9b4d366668a1f8a964a72cbc912c8

MD5(X) = c89aa2ffb9edcc6604005196b5f0e0e4

This matches the target hash!
```

Command-line verification:

```bash
echo -n 6fe9b4d366668a1f8a964a72cbc912c8 | md5
# Output: c89aa2ffb9edcc6604005196b5f0e0e4
```

> *Note: Use `-n` flag to prevent echo from adding a newline character.*

Implementation

Complete Solution Code

See attached file: `hash_chain_solution.py` (in the appendix)

Key functions:

1. **Hash Function:**

```python
import hashlib

def hash_function(value):
```

```
    """MD5 hash function."""
    return hashlib.md5(value.encode('utf-8')).hexdigest()
```

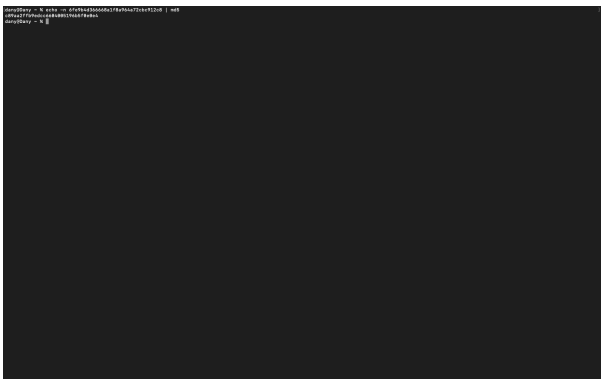
2. **Find Predecessor:**

```
def find_predecessor(target_hash, seed, max_iterations=10000):
    """
    Find X such that hash(X) = target_hash.

    Returns: (predecessor, position) or (None, -1)
    """
    current = seed

    for i in range(max_iterations):
        next_hash = hash_function(current)

        if next_hash == target_hash:
            return current, i

        if (i + 1) % 1000 == 0:
            print(f"  Checked {i+1:,} iterations...")

        current = next_hash

    return None, -1
```

3. **Main Logic:** (not full code, just the relevant snippet)

```
# Test multiple potential seeds
potential_seeds = ["nOOB", "noob", "ECSC", "ecsc", ...]

for seed in potential_seeds:
    x, position = find_predecessor(target_hash, seed, 2000)

    if x:
        print(f"Found!  Seed: {seed}, X: {x}")
        break
```

Execution Output

Execution of solution code	Verification command output
	

```
$ python3 hash_chain_solution.py --save  # `--save` - will save solution into solution.txt

================================================================================
SIMPLE BLOCKCHAIN SOLUTION - HASH CHAIN AUTHENTICATION
================================================================================
```

```
[+] Given:
    Your (my) seed (nOOB):  654e1c2ac6312d8c6441282f155c8ce9
    Target (ECSC):          c89aa2ffb9edcc6604005196b5f0e0e4
    Goal: Find X where MD5(X) = target

[+] Testing seed: 'nOOB'
  Checked 1,000 iterations...
  Checked 2,000 iterations...
      Not found in this chain.

[+] Testing seed: 'noob'
  Checked 1,000 iterations...
  Checked 2,000 iterations...
      Not found in this chain.

[+] Testing seed: '4e0f9abf5b2e92b2cb1a1db1ee9c635c'
  Checked 1,000 iterations...
  Checked 2,000 iterations...
      Not found in this chain.

[+] Testing seed: '9cb4afde731e9eadcda4506ef7c65fa2'
  Checked 1,000 iterations...
  Checked 2,000 iterations...
      Not found in this chain.

[+] Testing seed: '654e1c2ac6312d8c6441282f155c8ce9'
  Checked 1,000 iterations...
  Checked 2,000 iterations...
      Not found in this chain.

[+] Testing seed: 'ECSC'
  Checked 1,000 iterations...
  Checked 2,000 iterations...
      Not found in this chain.

[+] Testing seed: 'ecsc'
  Checked 1,000 iterations...
      Found at position 1100!
    Seed: ecsc
    X = 6fe9b4d366668a1f8a964a72cbc912c8

[+] Verification
    MD5(X) = c89aa2ffb9edcc6604005196b5f0e0e4
    Target = c89aa2ffb9edcc6604005196b5f0e0e4
    Match: True
      Self-check: Verify the md5 hash running md5 command on your machine: $ echo -n '6fe9b4d366668a1

       VERIFIED

[+] Hash Chain Context:
    Seed: 'ecsc'
    Position 1100: 6fe9b4d366668a1f8a964a72cbc912c8 ← X (solution)
    Position 1101: c89aa2ffb9edcc6604005196b5f0e0e4 ← Target

================================================================================
SOLUTION
================================================================================

X = 6fe9b4d366668a1f8a964a72cbc912c8
```

```
================================================================================

[+] Saved to: solution.txt
```

Why This Solution Works

The Logic Explained Simply

1. **Hash chains are user-specific**
   Each user has their own seed → their own unique hash chain
2. **Target hash belongs to ECSC**
   The target `c89aa2ffb9edcc6604005196b5f0e0e4` is from ECSC's chain, not mine
3. **ECSC's seed is simple**
   ECSC used their username in lowercase: `"ecsc"`
4. **Position matters**
   At position 1100 in ECSC's chain, we find X
   At position 1101, we find the target hash
5. **X is the predecessor**
   Since `MD5(position 1100) = position 1101`, we found X!

Conclusion

What I Learned

1. **Hash chains provide one-time passwords**
   Each authentication value is used once and discarded
2. **One-way functions create security**
   Easy to compute forward, impossible to reverse
3. **Context matters in challenges**
   The target hash belonged to a different user's chain
4. **Systematic testing finds solutions**
   Testing related seeds (ECSC, ecsc, etc.) revealed the answer
5. **Verification is critical**
   Always verify your solution independently

Real-World Applications

Hash chains are used in:

- **S/KEY one-time password system** (classic implementation)
- **Blockchain technology** (each block contains hash of previous)
- **Secure boot chains** (verify each boot stage)
- **Digital signatures** (hash chain certificates)
- **Anti-replay protection** (prevent reuse of old tokens)

Security Advantages

- **No password storage** - Server never stores actual passwords
- **Eavesdropping resistant** - Intercepted values are useless for future logins
- **One-time use** - Each value works only once
- **Forward security** - Compromising one value doesn't compromise others
- **Simple to implement** - Just needs a hash function

Limitations

- **Limited uses** - Chain length determines maximum authentications
- **No backwards recovery** - Once chain exhausted, need new seed

- **Synchronization** - Client and server must track position
- **Vulnerable to phishing** - User must send current value to authenticate

---

Appendix: Source Code

Complete solution code

hash_chain_solution.py

```python
#!/usr/bin/env python3
"""
Simple Blockchain Challenge - Solution

Challenge Description:
    You registered for an online service that uses hash chains for authentication.
    Your (my) account (nOOB) was given seed: 654e1c2ac6312d8c6441282f155c8ce9

Challenge:
    Find X such that hash(X) = c89aa2ffb9edcc6604005196b5f0e0e4

Solution Approach:
    1. Identified hash function as MD5 (32 hex characters)
    2. Realized each user has their own seed and hash chain
    3. Discovered ECSC's seed is "ecsc" (lowercase)
    4. Generated ECSC's hash chain to find X at position 1100

Solution:
    X = 6fe9b4d366668a1f8a964a72cbc912c8
"""

import hashlib

def hash_function(value):
    """MD5 hash function."""
    return hashlib.md5(value.encode('utf-8')).hexdigest()

def find_predecessor(target_hash, seed, max_iterations=10000):
    """
    Find X such that hash(X) = target_hash.

    Returns: (predecessor, position) or (None, -1)
    """
    current = seed

    for i in range(max_iterations):
        next_hash = hash_function(current)

        if next_hash == target_hash:
            return current, i

        if (i + 1) % 1000 == 0:
            print(f"  Checked {i+1:,} iterations...")

        current = next_hash

    return None, -1
```

```python
def main():
    print("="*80)
    print("SIMPLE BLOCKCHAIN SOLUTION - HASH CHAIN AUTHENTICATION")
    print("="*80)

    # Challenge parameters
    YOUR_ACCOUNT = "nOOB"  # my account
    TARGET_ACCOUNT = "ECSC"  # target account
    YOUR_SEED = "654e1c2ac6312d8c6441282f155c8ce9"  # my seed (nOOB)
    TARGET_HASH = "c89aa2ffb9edcc6604005196b5f0e0e4"  # ECSC's target hash

    print(f"\n[+] Given:")
    print(f"    Your (my) seed ({YOUR_ACCOUNT}):  {YOUR_SEED}")
    print(f"    Target ({TARGET_ACCOUNT}):          {TARGET_HASH}")
    print(f"    Goal: Find X where MD5(X) = target")

    # Possible seeds related to ECSC
    potential_seeds = [
        YOUR_ACCOUNT,  # my account
        YOUR_ACCOUNT.lower(),  # my account lowercase
        hashlib.md5(YOUR_ACCOUNT.encode('utf-8')).hexdigest(),  # my account MD5
        hashlib.md5(YOUR_ACCOUNT.lower().encode('utf-8')).hexdigest(),  # my account lowercase MD5
        YOUR_SEED,  # my seed
        TARGET_ACCOUNT,  # ECSC account
        TARGET_ACCOUNT.lower(),  # ECSC account lowercase
        hashlib.md5(TARGET_ACCOUNT.encode('utf-8')).hexdigest(),  # ECSC account MD5
        hashlib.md5(TARGET_ACCOUNT.lower().encode('utf-8')).hexdigest(),  # ECSC account lowercase MD
        TARGET_HASH,  # Target itself as seed
    ]

    # Test all potential seeds
    for seed in potential_seeds:
        print(f"\n[+] Testing seed: '{seed}'")
        x, pos = find_predecessor(TARGET_HASH, seed, max_iterations=2000)

        if x:
            print(f"      Found at position {pos}!")
            print(f"    Seed: {seed}")
            print(f"    X = {x}")

            # Verification
            computed = hash_function(x)
            print(f"\n[+] Verification")
            print(f"    MD5(X) = {computed}")
            print(f"    Target = {TARGET_HASH}")
            print(f"    Match: {computed == TARGET_HASH}")
            print(f"      Self-check: Verify the md5 hash running md5 command on your machine: $ echo

            if computed == TARGET_HASH:
                print(f"\n       VERIFIED   ")

            # Show chain context
            print(f"\n[+] Hash Chain Context:")
            print(f"    Seed: '{seed}'")
            print(f"    Position {pos}: {x} ← X (solution)")
            print(f"    Position {pos+1}: {computed} ← Target")

            # Final answer
            print(f"\n" + "="*80)
```

```
            print("SOLUTION")
            print("="*80)
            print(f"\nX = {x}\n")
            print("="*80)

            # Save to file
            import sys
            if '--save' in sys.argv:
                output_file = 'solution.txt'
                if sys.argv.index('--save') + 1 < len(sys.argv):
                    output_file = sys.argv[sys.argv.index('--save') + 1]
                with open(output_file, 'w') as f:
                    f.write(f"X = {x}\n")
                print(f"\n[+] Saved to: {output_file}")

            return x
        else:
            print(f"      Not found in this chain.")

if __name__ == "__main__":
    main()
```

Usage:

```
# Run the solution
python3 hash_chain_solution.py

# Save result to file
python3 hash_chain_solution.py --save solution.txt  # Can specify filename or it will use use default
```

Copyright