E.1 RSA Challenge

Breaking it Down

This is an RSA encryption challenge. We're given the modulus `n`, public exponent `e`, and ciphertext `c`. The goal is to decrypt `c` to recover the original message (the flag) by finding the private key `d`.

RSA encryption works by: `c = m^e mod n`, where `m` is the plaintext message as an integer. Decryption requires: `m = c^d mod n`, where `d` is the private exponent.

To find `d`, we need to factor `n` into its prime factors `p` and `q`, then compute Euler's totient `phi_n = (p-1) * (q-1)`, and finally `d = e^{-1} mod phi_n`.

Since `n` is a large number (1024 bits), manual factoring is impossible, so we use factordb.com to get the factors.

My Thought Process

- RSA is asymmetric: public key (n, e) for encryption, private key (d) for decryption.
- Factoring `n` gives `p` and `q`.
- Use `phi_n` to compute `d` as the modular inverse of `e`.
- Decrypt `c` with `m = c^d mod n`.
- `m` is an integer representing the plaintext; convert to bytes (big-endian), then decode as UTF-8 string to get the flag.

Step-by-Step Solution

1. **Factor n using factordb.com**:
    - Query: `http://factordb.com/api?query={n}`
    - Parse JSON response for factors if status is 'FF' (fully factored).
    - Extract `p` and `q` as integers.
2. **Compute Euler's totient**:
    - Formula: `phi_n = (p - 1) * (q - 1)`
3. **Compute private exponent d**:
    - Formula: `d = pow(e, -1, phi_n)` (modular inverse of e modulo phi_n)
4. **Decrypt ciphertext**:
    - Formula: `m = pow(c, d, n)` (modular exponentiation)
5. **Convert m to bytes**:
    - Byte length: `(m.bit_length() + 7) // 8` (round up bits to bytes)
    - `m_byte = m.to_bytes(byte_length, 'big')` (big-endian byte order)

6. **Decode to flag**:
    - `flag = m_byte.decode('utf-8')`

Tools Used

- Python 3 with `requests` library for API calls
- factordb.com for factoring large numbers
- Modular arithmetic functions (`pow` for exponentiation and inverse)

Flag

CS2107{B1G_but_f4ct0rDB_c4n_h3lP_m3_S0lV3}
c:\Users\xiang\Desktop\y2\cs2107\assignment1\easy1\writeups.md

E.2: Office Document Password Cracking

Breaking it Down

This is a password cracking challenge for a Microsoft Office 2013 document (encrypted_flag.docx). The document is encrypted with AES-256 using PBKDF2-SHA1 with 100,000 iterations. To access the contents, which contain the flag, I needed to crack the password.

The process involves extracting the encryption hash from the document, cleaning it for proper encoding, then using a password cracker like Hashcat with GPU acceleration to dictionary attack the password.

My Thought Process

- Office documents store encryption metadata that can be extracted using tools like office2john.py.
- The hash needs to be cleaned to remove the filename prefix and ensure ASCII encoding, as it may be in UTF-16.
- Use Hashcat with mode 9600 for Office 2013, dictionary attack with rockyou.txt.
- For GPU acceleration on AMD cards, install AMD HIP SDK and set backend to HIP.
- Initial issues: No OpenCL/HIP platforms detected due to missing HIP SDK; hash encoding problems causing signature unmatched errors.
- Fallback: Used WSL with PoCL for CPU cracking, but slow; eventually installed HIP SDK for fast GPU cracking.

Step-by-Step Solution

1. **Extract hash**:
    - Run `python office2john.py encrypted_flag.docx > hash.txt`
    - Output:
      `encrypted_flag.docx:$office$*2013*100000*256*16*5d81505aba6694cacf9b8aaa49106cd7*eee24babcc141c8f3501f68408b9df93*e08dc732ca8a9862b4fd374118b651a4ba34cb6bdd32f340082f51c97d6451e9`

2. **Clean hash**:

- Remove filename prefix: `Get-Content hash.txt | ForEach-Object { $_.Split(':')[1] } | Set-Content -Encoding ASCII hash_clean.txt`
- Convert from UTF-16 to ASCII: `tr -d '\0' < hash_clean.txt > hash_clean_ascii.txt`

3. **Crack password**:
   - Install AMD HIP SDK 6.4.2 for GPU support.
   - Set backend: `$env:HASHCAT_BACKEND = "HIP"`
   - Run `hashcat -m 9600 -a 0 hash_clean_ascii.txt rockyou.txt`
   - Password found: `cs2107711923`
4. **Access document**:
   - Open encrypted_flag.docx with password `cs2107711923`.
   - Extract flag from contents.

Tools Used

- office2john.py (from John the Ripper)
- Hashcat v7.1.2
- AMD HIP SDK 6.4.2
- rockyou.txt wordlist
- WSL with PoCL (fallback for CPU)

Flag

CS2107{greycat>hashcat_provemewrong}

E.3: ZipCrypto

Breaking it Down

This is a ZipCrypto encryption challenge. The ZIP file sus_package.zip contains encrypted files using the legacy ZipCrypto algorithm, which is vulnerable to known-plaintext attacks. The ZIP includes a known file (Assignment 1.pdf) that I have in plaintext, allowing me to recover the encryption keys and decrypt the flag.txt file.

ZipCrypto uses a stream cipher with keys derived from a password, but the Linear Congruential Generator (LCG) used for the keystream is predictable and reversible given enough known plaintext. bkcrack exploits this by brute-forcing the internal keys (X, Y, Z) using the known data.

My Thought Process

- ZipCrypto is outdated and weak, AES-256 is preferred now.
- Known-plaintext attacks work because the cipher is a stream cipher, and with known input/output, the keystream can be recovered.
- The LCG in ZipCrypto has a small state (96 bits), making it feasible to brute-force with ~80KB of known data.

- If the file was compressed (deflate), the attack would be harder because compression randomizes the data, but here it's stored uncompressed.
- Use bkcrack for the attack: list files, attack with known plaintext, decrypt target file.

Step-by-Step Solution

1. **List ZIP contents**:
    - Run `.\bkcrack-1.7.0-win64\bkcrack.exe -L sus_package.zip`
    - Confirms files: "Assignment 1.pdf" and "flag.txt", both encrypted with ZipCrypto.
2. **Perform known-plaintext attack**:
    - Run `.\bkcrack-1.7.0-win64\bkcrack.exe -C sus_package.zip -c "Assignment 1.pdf" -p "../Assignment 1.pdf"`
    - This brute-forces the keys using the known plaintext.
    - Internally: bkcrack reads the encrypted "Assignment 1.pdf" from the ZIP and the plaintext "../Assignment 1.pdf". It XORs the encrypted data with the plaintext to recover the keystream. Then, it brute-forces the LCG state (keys X, Y, Z) by trying combinations until the generated keystream matches the recovered one. The LCG is reversible with enough data (~80KB here).
    - Output: Keys found: 7e827b05 98ea3b23 33a5bfc8
3. **Decrypt flag.txt**:
    - Run `.\bkcrack-1.7.0-win64\bkcrack.exe -C sus_package.zip -k 7e827b05 98ea3b23 33a5bfc8 -c "flag.txt" -d flag.txt`
    - This decrypts flag.txt to the current directory.
    - Internally: bkcrack initializes the LCG with the provided keys (X, Y, Z), generates the keystream for "flag.txt", and XORs it with the encrypted data from the ZIP to recover the plaintext, saving it as flag.txt.
4. **Read the flag**:
    - Open flag.txt, contains the flag.

Tools Used

- bkcrack v1.7.0 (for ZipCrypto known-plaintext attacks)

Flag

CS2107{but_what_if_zipcrypto_deflate_is_used_hmm...}

M.1: Rolling Thunder!

Breaking it Down

This is a symmetric XOR encryption challenge with a repeating key. The plaintext flag.png is encrypted using rolling XOR with key "elephant" (8 bytes), producing flag.png.enc. XOR is symmetric and associative: P XOR K = C; C XOR K = P. Since the key repeats, it's vulnerable when the plaintext header is known.

PNG files have a fixed 8-byte header: 89 50 4E 47 0D 0A 1A 0A (hex), which allows key recovery from the encrypted header.

My Thought Process

- XOR properties: Commutative and associative, self-inverse (A XOR B XOR B = A).
- Repeating key: Key cycles every len(K) bytes, here 8.
- Known plaintext attack: With known P_header and C_header, K = P_header XOR C_header.
- Since K is repeating and short, first 8 bytes of C reveal K.
- Decryption: Apply same XOR with K to entire C.

Step-by-Step Solution

1. **Read encrypt.py**:
    - Rolling XOR: data[i] ^= KEY[i % len(KEY)]
    - KEY is string "elephant", len=8.
2. **Recover key**:
    - PNG header hex: 89504E470D0A1A0A
    - Convert to bytes: header = (0x89504E470D0A1A0A).to_bytes(8, 'big')
    - Encrypted header: enc[:8]
    - Key bytes: bytes(a ^ b for a, b in zip(enc[:8], header))
    - Key string: key_bytes.decode('utf-8') # "elephant"
3. **Decrypt file**:
    - For i in range(len(enc)): dec[i] = enc[i] ^ ord(key[i % len(key)])
    - Write dec to flag.png
4. **Verify**:
    - flag.png is valid PNG with flag.

Tools Used

- Python (bytes operations, XOR)

Flag

CS2107{r0LL1ng_x0R_k4y_98bfa}

M.2: Secret Message

Breaking it Down

This is a monoalphabetic substitution cipher. Each plaintext letter maps to a unique cipher letter. The ciphertext in secret_message.txt is decrypted using frequency analysis and pattern recognition.

Monoalphabetic ciphers preserve letter frequencies and patterns, making them breakable with statistical methods.

My Thought Process

- Substitution: 1-1 mapping, invertible.
- Frequency attack: Map high-frequency cipher letters to high-frequency English (E, T, A, O, I).
- Pattern attack: Identify common digrams (TH, HE, IN), trigrams (THE, AND), word shapes.
- Manual: Count frequencies, guess mappings, refine.
- Automated: Use hill-climbing solvers like quipqiup.com, which optimize mappings based on English n-gram probabilities.

Step-by-Step Solution

1. **Read ciphertext**:
    - secret_message.txt contains the encrypted text.
2. **Use quipqiup**:
    - Paste ciphertext into quipqiup.com.
    - It uses hill-climbing: Random initial mapping, scores with English quadgram/trigram probabilities, swaps letters to maximize score, iterates to local optimum.
    - Outputs ranked decryptions; pick the coherent one.
3. **Verify**:
    - Plaintext is readable, flag at end.

Tools Used

- quipqiup.com (hill-climbing monoalphabetic solver)

Flag

Flag

CS2107{AN4LYS1S_FREQU3NT1Y_AB56C}

M.4: Spill Some Tea

Breaking it Down

"M.4 Spill Some Tea" involves interacting with a server that encrypts messages using ChaCha20-Poly1305 AEAD (Authenticated Encryption with Associated Data) cipher. The server reuses the same nonce for all encryptions, which is a critical security flaw. The goal is to recover the encrypted flag by exploiting this reuse to decrypt it without the key.

The server provides two main options:

1. Encrypt user-provided "tea" (plaintext).
2. Retrieve the encrypted flag.

Given only the encrypted outputs, we must decrypt the flag using a known-plaintext attack.

ChaCha20-Poly1305 combines:

- **ChaCha20**: A stream cipher for confidentiality.
- **Poly1305**: A MAC for integrity.

ChaCha20 generates a pseudorandom keystream from key, nonce, and counter. The keystream is XORed with plaintext to produce ciphertext.

Poly1305 computes a 16-byte tag over the ciphertext for authentication.

Encrypted message = ciphertext || tag (16 bytes).

My Thought Process

- ChaCha20 requires unique nonces per encryption with the same key; reuse makes keystream identical.
- This violates RFC 8439, enabling stream cipher attacks.
- Known-plaintext attack: Encrypt known string, recover keystream = known ⊕ ciphertext_known (trim tag).
- Decrypt flag: plaintext_flag = ciphertext_flag ⊕ keystream (trim tag).
- ChaCha20 uses Salsa20 quarter-rounds (addition, XOR, rotation) in 20 rounds.
- State: 16 words (constants + key + counter + nonce), mixed and added back.
- Poly1305: One-time key from keystream, accumulates with modular arithmetic.
- Attack ignores tags, focuses on ChaCha20 XOR.

Step-by-Step Solution

1. **Connect to server**:
    - Use socket or nc: `nc cs2107-ctfd-i.comp.nus.edu.sg 5003`
2. **Get encrypted flag**:
    - Send `2`, receive hex and length (76 bytes total, 60 + 16 tag).
3. **Encrypt known plaintext**:
    - Send `1`, input "A"*60, receive encrypted hex (76 bytes).
4. **Recover keystream**:
    - known = b"A"*60
    - enc_known = bytes.fromhex(hex)[:-16] # 60 bytes
    - keystream = bytes(a ^ b for a, b in zip(known, enc_known))
5. **Decrypt flag**:
    - enc_flag = bytes.fromhex(flag_hex)[:-16] # 60 bytes
    - flag_bytes = bytes(a ^ b for a, b in zip(enc_flag, keystream))

    - flag = flag_bytes.decode('utf-8')

Tools Used

- Python with socket for connection
- XOR operations for crypto math

Flag

CS2107{t3a_w4s_sP1lled_chacha20_n0nc3_r3us3_1s_d4ng3r0us!!!}

H.1: Lazy Programmer

Breaking it Down

This is a challenge exploiting the predictability of the C standard library's rand() function. The server provides a timestamp string, and the task is to predict the next random number generated after seeding rand() with time(NULL) at that timestamp, then compute the MD5 hash of that number as a hex string and send it back to get the flag.

The rand() function implements a Linear Congruential Generator (LCG) with parameters: X_{n+1} = (a * X_n + c) mod m, where a=1103515245, c=12345, m=2^31. It's deterministic once seeded, so if you know the seed (time(NULL)), you can reproduce the sequence.

The server seeds rand() with time(NULL) from the timestamp, generates one rand(), computes MD5 of that value, and expects you to match it.

My Thought Process

- The challenge name "Lazy Programmer" suggests something predictable or poorly implemented, like using rand() without proper seeding—pointing to the classic issue of rand() being seeded with time(NULL), making it predictable if you know the exact time.
- The server sends a time string (e.g., "Mon Oct 7 14:23:45 2025"), which looks like the output of ctime(time(NULL)). This infers that the server is using time(NULL) as the seed, and the string is in local time format.
- Since ctime() formats time in local timezone, and the server is likely in Singapore (UTC+8, common for NUS), I infer the timezone is Asia/Singapore. Parsing this string with std::get_time and setting TZ accordingly should give the correct Unix timestamp.
- Knowing the seed (time_t), I can replicate srand(seed); int r = rand(); exactly, as rand() is deterministic per seed.
- The server generates one rand() after seeding, so I need to do the same and MD5 the result as hex.
- To ensure identical behavior, the code must run on the same architecture (Intel x86_64 Linux), so I infer using NUS SSH VM for compilation and execution.
- Potential pitfalls: Timezone mismatches could shift the timestamp by hours, leading to wrong seeds; architecture differences might affect rand() implementation, though glibc is standard.

Step-by-Step Solution

1. **Understand the problem**:
    - Server sends time string, expects MD5 of next rand() after seeding with that time.
2. **Parse time string**:
    - Use std::get_time with format "%a %b %e %H:%M:%S %Y" to parse into tm.
    - Set timezone to Asia/Singapore, tzset().
    - Use mktime to get time_t seed.
3. **Seed and generate**:
    - srand(seed); int r = rand();
4. **Compute MD5**:
    - Convert r to string, MD5 it, output as hex.
5. **Write C++ code**:
    - Include necessary headers: , , , ,
    - Custom MD5 implementation (md5.c, md5.h).
    - Parse time, set TZ, seed, rand, MD5.
6. **Compile on NUS SSH**:
    - scp files to VM.
    - g++ solution.cpp md5.c -o solution
7. **Run solver**:
    - nc to server, get time, run ./solution "time string", get hash, send back.

Tools Used

- C++ (std::get_time, mktime, srand/rand)
- Custom MD5 implementation
- SSH/scp for NUS VM
- Netcat for server interaction

Flag

CS2107{p53ud0_r4nd0m_15_n0t_7ru3_r4nd0m!!}

H.2: MD5 Collision Attack

Breaking it Down

This is a cryptographic attack challenge exploiting the broken MD5 hash function. The task is to create two different PDF files that have the same MD5 hash but different contents, while appearing visually identical to a

given "Assignment 1.pdf". The server validates that the files are valid PDFs, have identical MD5 hashes, and have different binary contents.

MD5 collisions occur when two different inputs produce the same 128-bit hash. This is possible because MD5's 128-bit output has only 2^128 possible values, but the input space is infinite. Researchers have found practical collision attacks, including chosen-prefix collisions where you can find suffixes for a given prefix that result in the same hash.

For PDFs, collisions can be embedded in unused binary data (e.g., comments or streams) without affecting the visual appearance, since PDF viewers ignore unknown data.

My Thought Process

- The challenge requires generating two visually identical PDFs with the same MD5 hash but different binary contents, which points to exploiting MD5's known weaknesses—recalling that MD5 was broken in 2004 by Wang et al., making collisions computationally feasible even on standard hardware.
- Since the PDFs must be based on a given prefix ("Assignment 1.pdf"), this suggests a chosen-prefix collision attack, where I find different suffixes S1 and S2 such that MD5(prefix + S1) = MD5(prefix + S2), allowing the colliding files to be prefix + S1 and prefix + S2.
- To generate such collisions efficiently, I infer needing a specialized tool; hashclash by Marc Stevens comes to mind, as it implements differential path attacks on MD5 specifically for finding collisions quickly.
- Encountering build issues with GCC segfaulting on Boost suggests compiler incompatibility, so I infer switching to clang++ and using system Boost libraries to resolve it, ensuring the tool compiles successfully.
- Once built, the tool outputs two binary files with identical MD5 (the full colliding messages including the prefix), so I infer renaming them to .pdf extensions for submission, as they should remain valid PDFs visually.
- Potential pitfalls: If the PDFs aren't valid after collision embedding, it might indicate the collision affected visible parts—need to verify with a PDF viewer; also, ensuring binary differences while keeping visual sameness requires the collision to be in non-rendering data.

Step-by-Step Solution

1. **Install dependencies**:
    - In WSL: `sudo apt install git build-essential autoconf automake libtool zlib1g-dev libbz2-dev clang g++-11`
    - Install Boost: `sudo apt install libboost-all-dev`
2. **Clone and build hashclash**:
    - `git clone https://github.com/cr-marcstevens/hashclash.git`
    - Faced specifications issues hence modified build.sh to use clang++ and system Boost (comment out local boost build).
    - `./build.sh` (uses clang++ to avoid GCC issues).
3. **Generate collisions**:
    - `./bin/md5_fastcoll -p "../Assignment 1.pdf" -o colliding1.bin colliding2.bin`

    - This finds collision suffixes for the PDF prefix, generating two full PDFs with same MD5.

4. **Prepare submission files**:
    - `mv colliding1.bin colliding1.pdf`
    - `mv colliding2.bin colliding2.pdf`
    - Verify: `md5sum colliding1.pdf colliding2.pdf` (should be identical).
    - Check contents differ: `diff colliding1.pdf colliding2.pdf` (Binary files diff).
5. **Submit to server**:
    - Upload colliding1.pdf and colliding2.pdf to http://cs2107-ctfd-i.comp.nus.edu.sg:5002/

Checks: valid PDFs, same MD5, different contents.

Tools Used

- hashclash (MD5 collision tool by Marc Stevens)
- clang++ (C++ compiler, avoids GCC Boost issues)
- Boost libraries (system installation)
- WSL Ubuntu for building
- md5sum for verification

Flag

CS2107{no_md5_for_you!!!}