Merkle-Damgard Length Extension Lab, Part 1

0. Requirements and Setup

This lab was built and tested on Ubuntu 24.04 but should work on any Ubuntu/Debian-based Linux system. I recommend using a Linux VM (e.g., in VirtualBox). A SEED Ubuntu image will suffice for all exercises.

Requirements

- **Python 3.7+**: Verify your installation by running:

  ```
  ```
  python3 --version
  ```
  ```

  If you don't have Python 3.7 or newer, install it via your package manager.
- **GCC & OpenSSL headers**: These are often pre-installed on Linux. If they're missing, install them with:

  ```
  sudo apt install build-essential libssl-dev
  ```

  On macOS (using Homebrew), you can install OpenSSL headers with:

  ```
  brew install openssl
  ```

Setup

- **Directory structure**: Create a dedicated lab folder and starter script:

  ```
  mkdir -p ~/hashing_lab && cd ~/hashing_lab
  touch hashing_lab.py
  ```

All Python functions will be implemented in `hashing_lab.py`. A template with function declarations has been provided alongside this document which you can use to complete the tasks listed below.

1. Encoding Binary Data for URLs

Complete the following python function to convert binary data into url-encoded hex representation:

```
def url_format(data: bytes) -> str:
    """
    Convert a bytes sequence to its URL-encoded representation by percent-encoding each byte.

    Each byte in the input is formatted as a two-digit hexadecimal number,
    prefixed with '%', as commonly used in URL encoding.

    Args:
        data (bytes): The input data to encode.

    Returns:
        str: A string where each byte of `data` is represented as '%XX',
             with XX being the lowercase hexadecimal value of the byte.

    Example:
        >>> url_format(b'Hello!')
        '%48%65%6c%6c%6f%21'
    """
```

This will be needed for an in-class lab component later.

2. Using Hash Functions in Python

Learn how to use Python to compute hashes of data. In particular:
- Use the library `hashlib` to compute your hashes. Consult the documentation here: https://docs.python.org/3/library/hashlib.html
- Implement the following function using `hashlib`:

```
def compute_hash(algorithm = 'md5',
        message,
        output_format = 'bytes'):
    '''

    Parameters
    ----------
    algorithm : str
        Must be one of the following algorithms:
```

```
        1. 'md5'
        2. 'sha256'
        3. 'sha512'

        Otherwise throws an error.
    message : bytes (or bytes-like object)
        Encoded message to be hashed with the given algorithm.
    output_format : str
        Must be one of the following:
        i. 'bytes'
        ii. 'hex'
        iii. base64

        Otherwise throws an error

    Returns
    -------
    The hash digest of message, using the given algorithm, in the given
format. If 'bytes', will return a bytes object. If 'hex' or 'base64' will
return a string of the given encoding.
    '''
```

3. Implementing Correct Padding

Consult RFC6234, Section 4 , to understand the padding schemes for SHA-256 and SHA-512. Consult RFC1321, Section 3 to understand the padding scheme for MD5. Note that these documents describe data with bits as basic units, and think a bit about how these statements might translate into ones using bytes as units. Implement the following function:

```
def compute_padding( algorithm: str = 'md5',
         output_format: str = 'bytes',
         message: bytes = None,
        ):
    """
    Parameters
    ----------
    algorithm : str
        One of: 'md5', 'sha256', 'sha512'
    message : bytes
        Data to hash. Required.
    output_format : str
        One of: 'bytes', 'hex', 'base64'

    Returns
    -------
```

```
    bytes or str
        The padding that the given algorithm adds to the message before
processing. To be used in implementation of the length extension attack.
    """
```

4. Compiling a C Binary for the Attack

The following C program continues the `sha256` algorithm from a given input state

```
/*
 *   gcc length_ext.c -o length_ext -lcrypto
 *
 *  Usage
 *  -----
 *      ./length_ext <sha256(M)_hex> <len_padded_bytes> <extension_hex>
 *
 *  where
 *      sha256(M)_hex      – 64 hex chars (digest of   M‖pad(M)  we stole)
 *      len_padded_bytes   – multiple of 64 (length of padded bytes used to
create the hash of M)
 *      extension_hex      – *even-length* hex string for the bytes you want
 *                           to append (e.g. "4578747261206d7367" == "Extra
msg")
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <arpa/inet.h>          /* htole32 */
#include <openssl/sha.h>

/* ---------- portable htole32 ------------------------------------------
*/
#ifndef htole32
# if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
#   define htole32(x)  (x)
# else
#   define htole32(x)  __builtin_bswap32((x))
# endif
#endif

/* ---------- helpers ---------------------------------------------------
*/
static int hex_to_state(const char *hex, uint32_t st[8])
```

```
{
    if (strlen(hex) != 64) return -1;
    for (int i = 0; i < 8; i++) {
        unsigned int w;
        if (sscanf(hex + 8*i, "%8x", &w) != 1) return -1;
        st[i] = htole32(w);                 /* OpenSSL keeps words LE-in-mem
*/
    }
    return 0;
}

static int hex_to_bytes(const char *hex, unsigned char **out, size_t
*outlen)
{
    size_t n = strlen(hex);
    if (n & 1) return -1;                   /* must be even number of chars
*/
    *outlen = n / 2;
    *out = malloc(*outlen);
    if (!*out) return -1;

    for (size_t i = 0; i < *outlen; i++) {
        unsigned int byte;
        if (sscanf(hex + 2*i, "%2x", &byte) != 1) {
            free(*out);
            return -1;
        }
        (*out)[i] = (unsigned char)byte;
    }
    return 0;
}

/* ---------- main ------------------------------------------------------
*/
int main(int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr,
            "Usage: %s <sha256(M)_hex> <len_padded_bytes>
<extension_hex>\n",
            argv[0]);
        return 1;
    }

    /* -- 1. parse the stolen digest ---------------------------------- */
    uint32_t state[8];
```

```c
    if (hex_to_state(argv[1], state) != 0) {
        fprintf(stderr, "sha256(M) must be exactly 64 hexadecimal
characters\n");
        return 1;
    }

    /* -- 2. parse length of M||pad(M) in bytes -------------------------- */
    size_t len_padded = strtoul(argv[2], NULL, 10);
    if (len_padded == 0 || (len_padded % 64)) {
        fprintf(stderr,
            "len_padded_bytes must be a positive multiple of 64 (got
%zu)\n",
            len_padded);
        return 1;
    }

    /* -- 3. decode attacker-controlled extension ------------------------ */
    unsigned char *ext = NULL;
    size_t ext_len = 0;
    if (hex_to_bytes(argv[3], &ext, &ext_len) != 0) {
        fprintf(stderr, "extension_hex must be valid even-length hex\n");
        return 1;
    }

    /* -- 4. normal SHA-256 init ----------------------------------------- */
    SHA256_CTX ctx;
    SHA256_Init(&ctx);

    /* -- 5. advance bit-counter by hashing dummy bytes ---------------- */
    for (size_t i = 0; i < len_padded; i++)
        SHA256_Update(&ctx, "*", 1);        /* content irrelevant */

    /* -- 6. overwrite chaining variables ------------------------------ */
    for (int i = 0; i < 8; i++)
        ctx.h[i] = state[i];

    /* -- 7. append extension -------------------------------------------- */
    SHA256_Update(&ctx, ext, ext_len);

    /* -- 8. final digest ------------------------------------------------ */
    unsigned char out[SHA256_DIGEST_LENGTH];
    SHA256_Final(out, &ctx);

    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++)
        printf("%02x", out[i]);
    putchar('\n');
```

```
    free(ext);
    return 0;
}
```

Even if you're unfamiliar with the precise workings of C, see if you can understand what this code is doing, based on what we have learned about how a Merkle-Damgard hash function works. In the same directory as your code, save this in a file named `length_ext.c` and run the following command (you will need `gcc` installed):

```
gcc length_ext.c -o length_ext -lcrypto
```

This will compile the C program into an executable program named `length_ext`. You might get some warnings, but this should compile. You can now run the program as follows from the command line:

```
  ./length_ext <sha256(M)_hex> <len_padded_bytes> <extension_hex>
```

Where

- sha256(M)_hex is the hex encoding of the sha256 hash of a message to be extended
- len_padded_bytes is the length of the message M, in bytes and after padding
- extension_hex is the hex encoding of the data you would like to extend by

5. Integrating Our Binary into Python

Looking up what you need to about calling external binaries from Python, implement the following function in Python:

```
def length_extend_sha256(digest_hex: str,
                         len_padded: int,
                         extension_hex: str,
                         binary: str | Path = "./length_ext") -> str:
    """
    Run the `length_ext` C program and return the forged digest.

    Parameters
    ----------
    digest_hex : str
        64-character hex SHA-256 of `M || pad(M)`.
```

```
    len_padded : int
        Length in **bytes** of `M || pad(M)` (must be a multiple of 64).
    extension_hex : str
        Even-length hex string for the data to append.
    binary : str or Path, optional
        Path to the compiled `length_ext` executable (default:
./length_ext).

    Returns
    -------
    str
        64-character hex digest of `M || pad(M) || extension`.
    """
```

Your function should successfully perform a `sha256` length extension attack on the given digest and length. It will likely be helpful to look into the `subprocess` library for Python. In particular, you can consult the method `subprocess.run()`, which allows you to execute shell commands and retrieve the output.

6. Running a Length Extension Attack

Test your length extension attack with the following function:

```
def test_attack(message: bytes, extension: bytes) -> None:
    """
    Test a SHA-256 length-extension attack by comparing:

      1. The hash of (message || padding(message) || extension)
      2. The hash obtained via a length-extension routine.

    Args:
        message (bytes):      The original message.
        extension (bytes):    The data to append via the length-extension
attack.

    Raises:
        AssertionError: If the two hashes don't match.
    """
    # 1) Compute the hash of (message + padding + extension)
    padding = compute_padding('sha256', 'bytes', message)
    extension_hash = compute_hash('sha256', 'hex', message + padding +
extension)
    print(f"Conventioally computed extension hash: {extension_hash}")

    # 2) Run the length-extension attack
```

```
    orig_hash = compute_hash('sha256', 'hex', message)
    attack_hash = length_extend_sha256(
        orig_hash,
        len(message + padding),               # original message length in
bytes
        extension.hex(),            # hex-encoded extension
        './length_ext'              # path to your extension binary/script
    )
    print(f"Hash computed with Length Extension: {attack_hash}")

    # 3) Verify they agree
    assert extension_hash == attack_hash, "Length-extension attack failed:
hashes differ"
```

```
```