Question 1: Hash Collision (8 points)

Note that this question will be submitted as "task 1" on protest.

The MD5 hash function is known to have collisions. Length extension attacks therefore create infinately many collisions. Create a collision that includes your UCID as part of the string being hashed. You'll submit your answer by providing the data represented in hexadecimal notation. You can use the tool `xxd` to convert binary data:

```
$ echo -n "hello"  |xxd -p
68656c6c6f
```

A collision in md5 is the following:

```
d131dd02c5e6eec4693d9a0698aff95c2fcab58712467eab4004583eb8fb
7f8955ad340609f4b30283e488832571415a085125e8f7cdc99fd91dbdf2
80373c5bd8823e3156348f5bae6dacd436c919c6dd53e2b487da03fd0239
6306d248cda0e99f33420f577ee8ce54b67080a80d1ec69821bcb6a88393
96f9652b6ff72a70
```

and

```
d131dd02c5e6eec4693d9a0698aff95c2fcab50712467eab4004583eb8fb
7f8955ad340609f4b30283e4888325f1415a085125e8f7cdc99fd91dbd72
80373c5bd8823e3156348f5bae6dacd436c919c6dd53e23487da03fd0239
6306d248cda0e99f33420f577ee8ce54b67080280d1ec69821bcb6a88393
96f965ab6ff72a70
```

Note that this is given *in hexadecimal*. This means that you need to create a file such that when you type: `cat file | xxd -p` you will see this data, while if you simply type `cat file` you'll see binary noise. You'll know you are storing your collision data correctly when you type `md5sum file1` and `md5sum file2` and you see the same MD5 value, but if you type `diff -q file1 file2` it will say `Files file1 and file2 differ`.

Create two files with your UCID in it and whatever else you feel like telling me. Make sure they have the same MD5 hash. Submit them using task1 on protest. Be sure to submit it as hexadecimal (i.e., output from `xxd -p`). I'll have a regexp check that before you submit. You can submit you answer as many times as you want until the deadline; the most recent valid one will be used.

Question 2: More Fun with Collisions (20 points)

We saw that collisions can be more powerful dependings on the file format. Some formats by virtue of their design can have more devastating effects for an arbitrary collision. Imagine the following bizzarely designed file format that renders it quite vulnerable to collision attacks.[1]

[1] One may ask "why would a file format exist with such a bizzare semantics and which renders it so particularly vulnerable to hash collisions" and indeed it is rather contreived, nevertheless it should not be the concern for file format designers to consider susceptibility for hash collisions in the designs but rather the concern of hash function designers to avoid having collisions.

```
byte  |  meaning
------+---------------------------------
 00   | magic number fixed 0xd1
 01   | magic number fixed 0x31
 02   | magic number fixed 0xdd
 03   | short "length" high value
 04   | short "length" low value
05-18 | reserved
 19   | short "jump" high value
 20   | short "jump" low value
 ..   |
 J+0  | short "datalength" high value
 J+1  | short "datalength" low value
 J+2  | byte 0 for stored data
 J+3  | byte 1 for stored data
 ..   |  ..
J+1+dl| byte dl-1 for stored data (last)
------+---------------------------------

J = jump MOD length
dl = datalength
```

The first three bytes are a "magic number", which typically is used to indicate the file format; in this case the byte sequence `0xd131dd` is used. (The tool `file` can tell you what type of file some blob is in part by using this magic numbers, though it will of course not recognize this ad hoc format.) The next 2 bytes are a big-endian short (16-bit) value that indicates the length of the file. The next bunch of bytes are reserved for future use (another typical feature of file formats or network protocols). Then bytes 19 and 20 are another big-endian short that indicates the "jump" or offset in the file where the data is actually stored. If this jump value is larger then the file size (bytes 3 and 4) then the jump is reduced modulo the file size.

The data is stored as a length-prefixed string starting at the offset jump (modulo file size). The first two bytes of jump indicate the length of the data, and the bytes after that store the actual data.

A C++ program that "prints" the stored data for this file format is available on the course website:

`http://pages.cpsc.ucalgary.ca/~joel.reardon/526/a2q2.cc`

You can compile this and run it, passing a filename as the first argument. If the file is not correctly formatted, then it will hopefully print a useful error message or otherwise just fail.

Your task is to submit two different files. Submit them as hex string (i.e., similiar to question 1) under "task 2" in protest. These two files must have the same md5 hash, and you will find the md5 collision from question 1 useful (indeed, what an unfortunate choice

the file format designed had for the magic number). The "stored data" for one file shall be "<your ucid> will receive an A+ in CPSC 526" and the other shall be "<your ucid> will receive a Z- in CPSC 526". You can use the C++ program to check the output is correct and `md5sum` to ensure that they have the same hash.

Question 3: Kerberos Forward Secrecy (20 points)

Kerberos does not have forward secrecy as provided for communications between Alice (the user) and Bob (e.g., the printer). But forward secrecy can still be added after Alice and Bob have mutually authenticated.

1. What are the long-term secrets in Kerberos?
2. What are the short-term secrets in Kerberos?
3. For each secret (long-term and short-term), assume that the secret is leaked to a passive adversary who has previously recorded all network traffic. What *new* information does this adversary learn that was not already known?
4. Explain why Kerberos does not have forward secrecy. Be specific about what data needs to be compromised and what the consequences of it are.
5. Augment Kerberos to have forward secrecy for the actual communication between Alice and Bob. You only need to specify the message sequences for the parts of the Kerberos protocol that you change (i.e., if Alice's communication with the TGS is unchanged then you do not need to specify it).