Wagner
Spring 2014

CS 161
Computer Security

Project 1

Due: Monday, March 3, 2014, 11:59PM

Version 1, February 25, 2014

1 Overview

The FBI is after their man. They are convinced that Dr. Evil, a sinister hacker-wizard who has taken up residence in a converted missile silo, is up to no good. Recently, one FBI agent managed to infiltrate Dr. Evil's silo, bribe one of his minions, and make off with a dozen backup tapes of his data—but in the process the agent was caught, disappeared for two days, and came back with his Pinterest and Tumblr pages modified to redirect to a video of some Rick Astley song. That agent won't talk about what happened during those two days and flat-out refuses to ever set foot in Dr. Evil's silo again, so it looks like the FBI is going to have to make do with whatever data they've already got.

Unfortunately, a dozen backup tapes can hold an awful lot of data, as the FBI is discovering. They're frantically hiring developers to write forensic tools to analyze all the files on the backup tapes, in hopes that this might reveal useful intelligence about Dr. Evil's methods. They've hired you to write a forensic analysis tool that analyzes image files and extracts metadata in them. In particular, they want you to write a tool that will extract the textual metadata from (a) PNG images, and (b) JPEG images with Exif data, and print out this textual metadata. They plan to run your tool on all the image files from the backup tapes and have a team of FBI analysts look through your tool's output.

But be careful! This might all be a diabolical plan to set a trap for the FBI. For all we know, Dr. Evil might have hidden a few files on his backup tapes containing data maliciously crafted to attack your tool.

2 Your task

You must write a forensic metadata-extraction tool for PNG and JPEG-Exif images, while taking care to make sure your tool is secure against malicious inputs. The FBI purchasing officers have specified that your tool must be written in C and you should work in a team of two on this project. You may use the standard C library and a decompression library we specify, but you must not use any other software or libraries. (For instance, you may not copy, use, invoke, or link to other code found on the Internet.)

3 Requirements

Building. We provide a starter project with a Makefile, some skeleton code, and a few test cases. They can be downloaded from the course web page. You should modify `png.c` and `jpg.c` with your implementation. We recommend you start by implementing metadata extraction for PNG images, as the JPEG-Exif file format is more involved.

We will use the Makefile in the starter project to compile your program. Any Makefile you provide will be overwritten with our original copy, so don't modify the Makefile. You can test your code on `hive`N`.cs.berkeley.edu` (where N is between 1 and 28).

Your program should accept a list of file names on the command line, and analyze each one, one at a time, in the order provided. For each file it should write to standard output the filename and then one line per element of meta data, as described below.

Make sure your program compiles. If your program does not compile, its empty executable will not do well on the following tests.

Your program needs to satisfy two core requirements: functionality and security. These are described below.

Functionality. When your program is invoked with a valid PNG file, or with a valid JPEG file containing Exif data, it must produce correct output. By "valid", we mean a file that complies with all of the requirements in Section 6. Note that if the input file is not valid, your program is not required to produce correct output, but it still must be memory-safe (see security below).

For valid PNG files, your program must print one line per tEXt, zTXt, or tIME chunk. These contain textual metadata, compressed textual data, and image creation time information, respectively.

For valid JPEG files containing Exif data, your program must print one line per Exif tag, for the following Exif tag types: DocumentName, ImageDescription, Make, Model, Software, DateTime, Artist, HostComputer, Copyright, RelatedSoundFile, DateTimeOriginal, DateTimeDigitized, MakerNote, UserComment, and ImageUniqueID.

The first line of output for each file must be of the form

```
File: foo.xxx
```

where `foo.xxx` is replaced with the filename.

Next, you should have one line of output per chunk/tag. This line should be of the form

```
Header: Value
```

where `Header` is the type of the data (for PNG, the key field; for JPEG, the tag type, such as DocumentName), and `Value` is the value of the data.

To help you, we have provided a set of functionality test cases along with the starter code. If you run `make functionality-tests` from the directory with your code, the Makefile we provided will automatically run your program against each of the functionality tests. You can find the test input in the `tests/functionality` directory, and the desired output in the `tests/functionality/out` directory.

Security. Your program must be free of memory-safety bugs, and may not take longer than 10 seconds to parse any file of size < 1 MB. (In effect, this means that your code should never get into an infinite loop.) There must be no input that can trigger any memory-safety bug or a timeout, even if the input file is invalid or maliciously constructed. We will attempt to break your program by running it on malicious files, so be ready!

Memory-safety bugs include all of the following: array out-of-bounds writes, array out-of-bounds reads, out-of-bounds reads or writes to any buffer, use of uninitialized data, accessing memory after it has been deallocated or is no longer valid, freeing memory twice, and freeing memory that was not allocated with malloc(). Your code must be completely free of all such bugs.[1] Integer overflow/underflow, signed/unsigned, and integer casting can also lead to memory-safety bugs, so be careful with them as well.

Security test cases. You also must write at least *five* security test cases for the PNG task. We will write several buggy variants of a PNG analyzer, each with a different memory-safety bug. Your test cases should be designed to exploit memory-safety bugs that might plausibly occur when writing a tool to extract metadata from PNG files. We will check whether your test cases do indeed reveal the bug in our known-buggy programs. You can get full points by detecting a memory-safety error in at least two of our known-buggy programs. Put your security test cases in the `tests/security_my` directory. You may specify up to 20 test cases; each file may be up to 1 MB long.

You do not need to write any test cases for the JPEG-Exif task.

Language. Your code must be written in C. Do not use assembly language, C++, or any other programming language. Do not spawn any other process. Do not use any library other than the standard C library and zlib (which is linked in, in the sample Makefile). Do not look for code online. Do not use other code or libraries found online. You may consult references, specifications, and other documents found online if you like, though we don't expect this to be necessary.

[1] These kinds of bugs can typically be exploited to allow code injection. However, if we find a memory-safety bug in your code, we will not be testing whether there is a code injection exploit for that bug: if your code has a memory-safety bug, we will consider it vulnerable, period. Therefore, you must avoid all such bugs in your code, no matter what.

4 Grading Process

To submit. Submit your code and test cases by running `submit proj1` from the directory where your code is, while logged into your class account. Only one of you needs to submit; you will be asked for the name of your partner when you submit.

To encourage equal work between teammates, after the project we will ask for separate feedback on the contribution and effectiveness of each partner. Both of you must respond to the survey at (TBD). This is mandatory.

Grading criteria. We will grade you on three criteria: functionality (for both PNG and JPEG Exif files), security (for both PNG and JPEG Exif files), and test case coverage (for your PNG test cases). Security will receive the highest weight, so make sure you have no memory safety bugs! But, you must pass at least half the functional tests for a format to get any security points for it.

We'll test functionality by running your program on a few test cases (the functionality tests) and checking that your program produces the right output. We have provided the functionality test cases with the starter code, so you can test in advance whether you will receive full points for functionality.

We will also have additional functionality tests that we keep hidden from you. The purpose of these tests is to detect submissions which hard coded the results for other functionality tests. If you fail these additional tests, we will look at your code. If we see that you have hard coded in the expected results, you will lose points. Otherwise (even if you failed the test) you will not lose any points.

Every few days we might run the autograder with some of our private test cases to give you partial early feedback, so it pays to start early and submit as soon as you have working code. The autograder will send you feedback by email to the email you listed when you registered your class account. After the deadline, we will run the autograder with all tests, email you a full report to your registered email, and record your grade in `glookup`.

We will penalize late submissions, as described on the course web page: if you submit less than 24 hours late, you will lose 10%; if you submit less than 48 hours, you will lose 20%; less than 72 hours, lose 40%; and no submissions will be accepted 72 hours or more after the deadline.

You should work in a group of two. You may not share your code or solutions with anyone other than your project partner. The maximum group size is two; you may not work in a group of three or larger.

5 Helpful tools and tips

The Valgrind tool and the ZLib library will be useful in writing your forensic analysis program. They are both installed on the `hiveN.cs` instructional machines. Our autograder will compile and run your code on those machines, so they would be good places to test your code.

You may also find a binary hex editor (i.e., a tool for reading and editing binary files) helpful for writing test cases. `bless` or `ghex` are useful for this purpose, if you have a graphical login. On a text-only connection, you can try `hexedit` or Emacs. On Emacs, `M-x hexl-mode` takes you to the binary editor. Unfortunately, `hexedit` and Emacs don't seem to let you insert characters (only overwrite existing characters).

If you want to calculate a CRC-32 checksum by hand, you can do this quickly with an interactive Python shell, as sketched below:

```
$ python
>>> import zlib
>>> '%8.8X' % zlib.crc32('tEXtTitle\0PngSuite')
'4F55CF4C'
```

So if we have a PNG `tEXt` chunk whose data is `Title\0PngSuite`, its CRC-32 checksum will be `0x4F55CF4C`.

Using Valgrind. Valgrind is helpful for checking for memory safety bugs and diagnosing their full cause. Valgrind works by running your program, with extra checks added to each memory access to try to detect out-of-bounds reads/writes. You can run your program on an input `foo.png` using Valgrind, as follows:

```
valgrind ./analyze foo.pn
```

In other words, you just prepend `valgrind` to the command line.

If this test case triggers a buffer overrun, you might get an error message like this from Valgrind:

```
==24677== Invalid write of size 1
==24677==    at 0x400556: g (broken.c:11)
==24677==    by 0x400563: h (broken.c:15)
==24677==    by 0x40057E: main (broken.c:20)
==24677==  Address 0x4c320a4 is 20 bytes after a block of size 80 alloc'd
==24677==    at 0x4A06409: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux
==24677==    by 0x40053D: f (broken.c:7)
==24677==    by 0x400579: main (broken.c:19)
```

Here is how to interpret that example. Valgrind reports that this is a `Invalid write`, meaning that this was a write to an invalid memory address. Valgrind shows the call stack: `main()` called `h()`, which called `g()`, which caused the invalid memory access on line 11 of broken.c. Valgrind also tells you that this was an attempt to write 20 bytes past the end of a 80-byte buffer, and it gives you the call stack when that buffer was allocated (namely, f() called malloc()).

For this exercise, we have provided a .valgrindrc file which customizes any invocation of valgrind. When you run:

```
valgrind ./analyze foo.pn
```

This will automatically add the flags `--suppressions=./libz.supp --leak-check=no --track-origins` This will give extra information about a certain kind of error message (use of uninitialized data). Valgrind detects a memory problem in ZLib's uncompress function that is expected behavior. The suppressions flag tells Valgrind to ignore this problem.

So, Valgrind can be helpful for testing. You can run your program on many nasty test cases (deliberately chosen to try to exercise potential security holes), using Valgrind, and Valgrind will tell you if it has detected any memory safety flaw.

However, Valgrind is not perfect. It does not detect all memory-safety bugs. In particular, it is especially weak at detecting overruns of stack buffers. We have set up the Makefile to compile your program with stack canaries, to make it more likely that you detect these problems during testing. If a stack buffer is overrun and this is detected by the stack canary mechanism, you will see a message like:

```
*** stack smashing detected ***: ./analyze terminated
```

Nonetheless, even with this addition, it is important to understand the limitations of testing: it can only find bugs that your test cases can trigger. This is especially true for security bugs, which are often hard to trigger accidentally. Therefore, your first line of defense should be to write your source code carefully and review it line-by-line for potential memory safety errors.

On your own machine. If you have your own Linux machine and you have apt-get, you can install Valgrind, zlib, bless, ghex, and hexedit as follows:

```
sudo apt-get install valgrind zlib1g-dev bless ghex hexedit
```

6 File Formats

You must support the PNG files and JPEG/Exif image file formats. Everything you need to know about these file formats is documented below.

Of course, we may run your program on arbitrary malicious inputs. Your program must never exhibit any memory safety bug on those inputs (but it doesn't need to produce useful output, if the input is not a valid PNG or JPEG/Exif file).

6.1 PNG file format

A PNG file starts with the 8 bytes `0x89 0x50 0x4e 0x47 0x0d 0x0a 0x1a 0x0a`. The rest of the file is a sequence of chunks. The format of a chunk is:

field name	length	description
length	4 bytes	the length of the data field, in bytes (stored as big-endian integer)
chunktype	4 bytes	4 ASCII characters identifying what kind of data this is
data	length bytes	
checksum	4 bytes	a CRC-32 checksum of the chunktype and data

The checksum is a CRC-32 checksum that is computed over the chunk type and the data (but *not* the length). Helpful hint: the `crc32` function in `zlib` can be used to compute and check the validity of the CRC-32 checksum.

There are many possible chunktypes. You need to know about the following three: `tEXt` (`0x74 0x45 0x58 0x74`), `zTXt` (`0x7A 0x54 0x58 0x74`), and `tIME` (`0x74 0x49 0x4D 0x45`). These chunks may be found in any order in the file. There can be any number of `tEXt` and `zTXt` chunks, but there will never be more than one `tIME` chunk in a valid PNG file. There will also be many other chunks; you should skip over them.

The format of the data field in the `tEXt` chunk is:

field name	length	description
key	variable	a sequence of characters
nul	1 byte	always the byte `0x00` (“\0”)
value	variable	a sequence of characters

Thus, the `tEXt` chunk is a key-value pair. Neither the key nor the value are nul-terminated. There are no restrictions on the characters in the key and value, except that there should be no nul bytes.

The format of the `zTXt` chunk is similar:

field name	length	description
key	variable	a sequence of characters
nul	1 byte	always the byte `0x00` (“\0”)
compressiontype	1 byte	always `0x00`, in this assignment
compressedvalue	variable	a sequence of characters, compressed

You only need to consider the case where the compressiontype byte is zero, in this assignment. As in the `tEXt` chunk, the key is a non-nul-terminated sequence of characters (bytes), with no restriction on what characters may be used except that there should be no nul bytes. The compressedvalue is compressed using zlib; you can use the `uncompress()` function from zlib to retrieve the original value. Once decompressed, the value is a non-nul-terminated sequence of characters (bytes), with no restriction on what characters may be used except that there should be no nul bytes, just as in the `tEXt` chunk.

The format of the data field in the `tIME` chunk is:

field name	length	description
year	2 bytes	the year, as a 16-bit big-endian integer
month	1 byte	the month, as a 8-bit integer
day	1 byte	...
hour	1 byte	
minute	1 byte	
second	1 byte	

When you see this chunk, you should produce output like

```
Timestamp: 12/25/2004 2:39:2
```

except using the correct year, month, day, etc. (Don't add a leading zero to the seconds, minutes, or other fields. You can use a printf format string like `%d/%d/%d %d:%d:%d`.)

For a detailed example of how a small PNG file is decoded, see the figure on the next page.

6.2 JPEG file format

A JPEG file consists of a sequence of variable-length chunks.

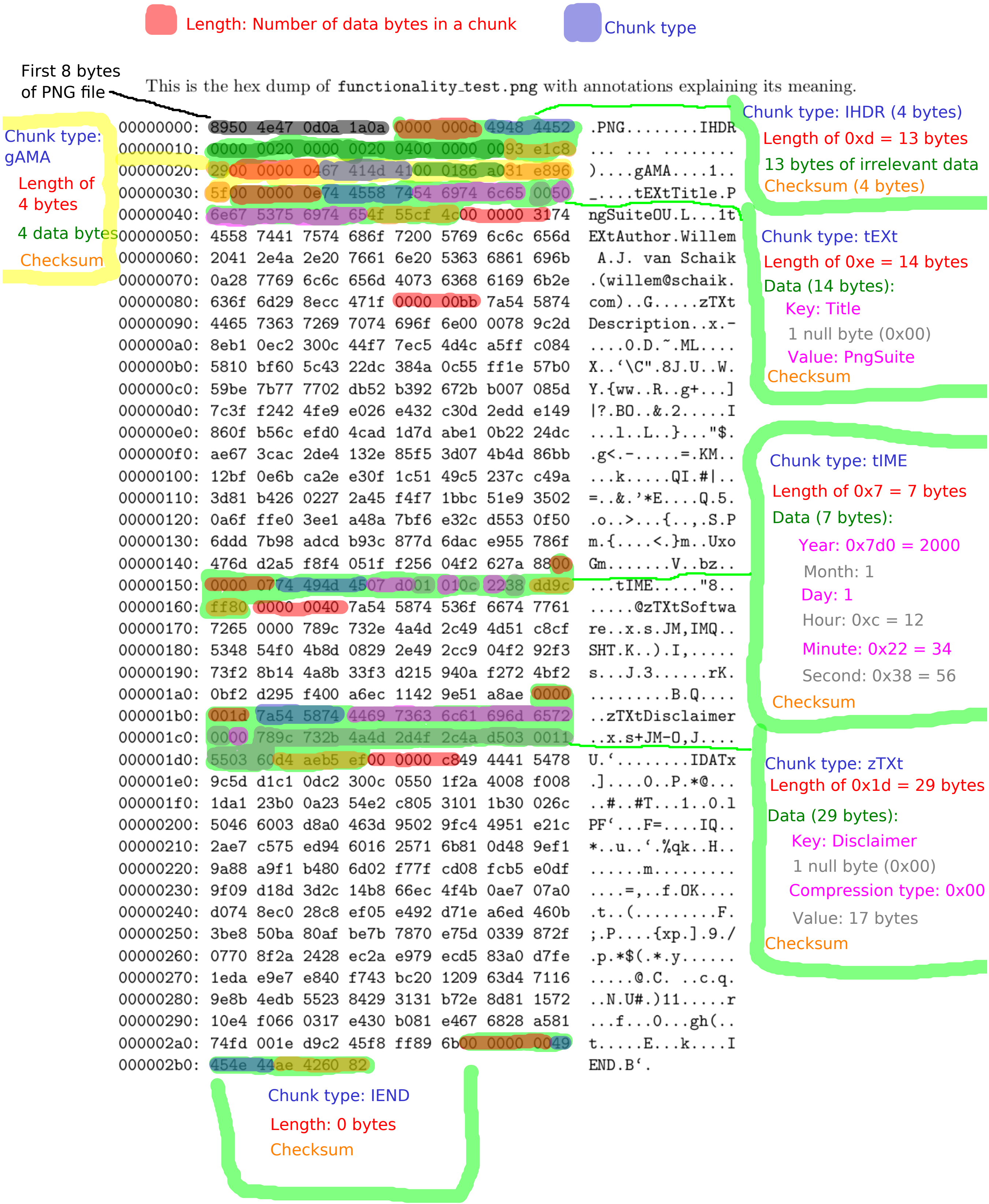
A chunk starts with a 2-byte marker, which indicates the type of the chunk. The marker is a 2-byte value in the range `0xff01` to `0xfffe`; the first byte of the marker is always `0xff`.

There are two formats for chunks: standard chunks and super chunks. Standard chunks have the following form:

field name	length	description
marker	2 bytes	indicates the type of chunk
length	2 bytes	the length of the chunk, not including the marker
data	(length − 2) bytes	

Chunks with markers between `0xffd0` and `0xffd9` (inclusive) are super chunks. Super chunks are like standard chunks, except they do not have a `length` field. Instead, in a super chunk, the marker is followed by a variable-length stream of encoded data. The encoded data has been encoded so that every `0xff` byte will be followed by a `0x00` byte. The length of the encoded data can be determined by looking for the next `0xff` byte that's followed by something other than a `0x00` byte; when you find one, that must be a marker that starts a new chunk, which in turn identifies the end of the encoded data stream. The SOS (Start of Scan) chunk starts with marker `0xffda` and is a super chunk. Any chunk that is not a super chunk (i.e., any chunk whose is below `0xffd0` or above `0xffda`) is a standard chunk.

This is the hex dump of `functionality_test.png` with annotations explaining its meaning.



🟥 Length: Number of data bytes in a chunk

🟦 Chunk type

First 8 bytes of PNG file

```
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
00000010: 0000 0020 0000 0020 0400 0000 0093 e1c8  ... ... ........
00000020: 2900 0000 0467 414d 4100 0186 a031 e896  )....gAMA....1..
00000030: 5f00 0000 0e74 4558 7454 6974 6c65 0050  _....tEXtTitle.P
00000040: 6e67 5375 6974 654f 55cf 4c00 0000 3174  ngSuiteOU.L...1t
00000050: 4558 7441 7574 686f 7200 5769 6c6c 656d  EXtAuthor.Willem
00000060: 2041 2e4a 2e20 7661 6e20 5363 6861 696b   A.J. van Schaik
00000070: 0a28 7769 6c6c 656d 4073 6368 6169 6b2e  .(willem@schaik.
00000080: 636f 6d29 8ecc 471f 0000 00bb 7a54 5874  com)..G.....zTXt
00000090: 4465 7363 7269 7074 696f 6e00 0078 9c2d  Description..x.-
000000a0: 8eb1 0ec2 300c 44f7 7ec5 4d4c a5ff c084  ....0.D.~.ML....
000000b0: 5810 bf60 5c43 22dc 384a 0c55 ff1e 57b0  X..‘\C".8J.U..W.
000000c0: 59be 7b77 7702 db52 b392 672b b007 085d  Y.{ww..R..g+...]
000000d0: 7c3f f242 4fe9 e026 e432 c30d 2edd e149  |?.BO..&.2.....I
000000e0: 860f b56c efd0 4cad 1d7d abe1 0b22 24dc  ...l..L..}..."$.
000000f0: ae67 3cac 2de4 132e 85f5 3d07 4b4d 86bb  .g<.-.....=.KM..
00000100: 12bf 0e6b ca2e e30f 1c51 49c5 237c c49a  ...k.....QI.#|..
00000110: 3d81 b426 0227 2a45 f4f7 1bbc 51e9 3502  =..&.'*E....Q.5.
00000120: 0a6f ffe0 3ee1 a48a 7bf6 e32c d553 0f50  .o..>...{..,.S.P
00000130: 6ddd 7b98 adcd b93c 877d 6dac e955 786f  m.{....<.}m..Uxo
00000140: 476d d2a5 f8f4 051f f256 04f2 627a 8800  Gm.......V..bz..
00000150: 0000 0774 494d 4507 d001 010c 2238 dd9c  ...tIME....."8..
00000160: ff80 0000 0040 7a54 5874 536f 6674 7761  .....@zTXtSoftwa
00000170: 7265 0000 789c 732e 4a4d 2c49 4d51 c8cf  re..x.s.JM,IMQ..
00000180: 5348 54f0 4b8d 0829 2e49 2cc9 04f2 92f3  SHT.K..).I,.....
00000190: 73f2 8b14 4a8b 33f3 d215 940a f272 4bf2  s...J.3......rK.
000001a0: 0bf2 d295 f400 a6ec 1142 9e51 a8ae 0000  .........B.Q....
000001b0: 001d 7a54 5874 4469 7363 6c61 696d 6572  ..zTXtDisclaimer
000001c0: 0000 789c 732b 4a4d 2d4f 2c4a d503 0011  ..x.s+JM-O,J....
000001d0: 5503 60d4 aeb5 ef00 0000 c849 4441 5478  U.‘.......IDATx
000001e0: 9c5d d1c1 0dc2 300c 0550 1f2a 4008 f008  .]....0..P.*@...
000001f0: 1da1 23b0 0a23 54e2 c805 3101 1b30 026c  ..#..#T...1..0.l
00000200: 5046 6003 d8a0 463d 9502 9fc4 4951 e21c  PF‘...F=....IQ..
00000210: 2ae7 c575 ed94 6016 2571 6b81 0d48 9ef1  *..u..‘.%qk..H..
00000220: 9a88 a9f1 b480 6d02 f77f cd08 fcb5 e0df  ......m.........
00000230: 9f09 d18d 3d2c 14b8 66ec 4f4b 0ae7 07a0  ....=,..f.OK....
00000240: d074 8ec0 28c8 ef05 e492 d71e a6ed 460b  .t..(.........F.
00000250: 3be8 50ba 80af be7b 7870 e75d 0339 872f  ;.P....{xp.].9./
00000260: 0770 8f2a 2428 ec2a e979 ecd5 83a0 d7fe  .p.*$(.*.y......
00000270: 1eda e9e7 e840 f743 bc20 1209 63d4 7116  .....@.C. ..c.q.
00000280: 9e8b 4edb 5523 8429 3131 b72e 8d81 1572  ..N.U#.)11.....r
00000290: 10e4 f066 0317 e430 b081 e467 6828 a581  ...f...0...gh(..
000002a0: 74fd 001e d9c2 45f8 ff89 6b00 0000 0049  t.....E...k....I
000002b0: 454e 44ae 4260 82                        END.B‘.
```

Chunk type: gAMA
Length of 4 bytes
4 data bytes
Checksum

Chunk type: IHDR (4 bytes)
Length of 0xd = 13 bytes
13 bytes of irrelevant data
Checksum (4 bytes)

Chunk type: tEXt
Length of 0xe = 14 bytes
Data (14 bytes):
Key: Title
1 null byte (0x00)
Value: PngSuite
Checksum

Chunk type: tIME
Length of 0x7 = 7 bytes
Data (7 bytes):
Year: 0x7d0 = 2000
Month: 1
Day: 1
Hour: 0xc = 12
Minute: 0x22 = 34
Second: 0x38 = 56
Checksum

Chunk type: zTXt
Length of 0x1d = 29 bytes
Data (29 bytes):
Key: Disclaimer
1 null byte (0x00)
Compression type: 0x00
Value: 17 bytes
Checksum

Chunk type: IEND
Length: 0 bytes
Checksum

The first chunk in a JPEG file must be a SOI (Start of Image) chunk, with marker `0xffd8`. The last chunk will be EOI (End of Image), with marker `0xffd9`. Both of these are super chunks: they do not have a length field. In addition, they do not have a data field either - the next marker is found immediately after it.

One of the chunks in the file will be a APP1 chunk, with marker `0xffe1`. This is a standard chunk. The data portion of this chunk contains a TIFF file, which in turn contains the Exif data. This is the only chunk whose contents you need to analyze. Your program should parse the JPEG file into chunks, skip over other than the APP1 chunk, find the APP1 chunk, and then analyze its contents as described below.

When you find the APP1 chunk, its data field will contain the byte sequence `0x45 0x78 0x69 0x66 0x00 0x00` followed by a TIFF file, whose format is described next.

6.3 TIFF file format

A TIFF file starts with a TIFF header. Somewhere after that is a 0th IFD as well as an Exif IFD, plus lots of other stuff you can ignore. Intuitively, each IFD is itself a sort of "TIFF mega-chunk."

The TIFF header is 8 bytes long and has the following form:

field name	length	description
endianness	2 bytes	`0x49 0x49`, in this project
magic string	2 bytes	always `0x2a 0x00`, in this project
offset	4 bytes	the start position of the 0th IFD, as an offset into the TIFF file

The first 2 bytes of the TIFF header are either `0x4949` (indicating a little-endian TIFF file) or `0x4d4d` (indicating a big-endian TIFF file). For this project, you only need to implement support for little-endian TIFF files. In other words, the first two bytes of the TIFF header will be `0x49 0x49`. For a little-endian TIFF image, all 2-byte and 4-byte integers will be stored in little-endian form. The next 2 bytes of the TIFF header are always `0x2a` followed by `0x00`. The final 4 bytes of the TIFF header indicate where the 0th IFD begins. In particular, they encode an integer that is the offset (in bytes, from the start of the TIFF file) where the 0th IFD begins. For instance, if this field holds the value `0x08 0x00 0x00 0x00`, then this corresponds to the integer `0x00000008` after taking into account the little-endian encoding, so the 0th IFD begins at the 8th byte of the TIFF file.

An IFD is a variable-length record, which contains a variable number of tag structures. A tag structure is used to store some kind of parameter; think of it as a name-value pair. The IFD starts with a 2-byte unsigned integer that records the number of tag structures in the IFD (in little-endian form, as always). This is followed by the sequence of tag structures.

A tag structure is a container for a list of items. All of the items in the list have the same type. The tag structure has the following format:

field name	length	description
tagid	2 bytes	what kind of tag structure this is
datatype	2 bytes	indicates the type of each item (e.g., int, float, etc.)
count	4 bytes	the number of items in the tag structure
offset_or_value	4 bytes	normally, start of item list, as an offset into the TIFF file

The tagid indicates what kind of tag structure this is and how to interpret the values in the list of items.

The datatype of a tag structure can be one of of the following values:

value	name	description
`0x0001`	byte	a byte
`0x0002`	ASCII string	one character (i.e., byte) of a nul-terminated ASCII string
`0x0003`	unsigned short	a two-byte unsigned integer
`0x0004`	unsigned long	a four-byte unsigned integer
`0x0005`	unsigned rational	two unsigned longs (numerator, denominator)
`0x0007`	undefined	one character (i.e., a byte)
`0x0008`	signed short	a two-byte signed integer
`0x0009`	signed long	a four-byte signed integer
`0x000a`	signed rational	two signed longs (numerator, denominator)
`0x000b`	float	4-byte floating point number
`0x000c`	double	8-byte floating point number

As always, the 2-byte integers shown above are stored in little-endian form; for instance, a 'short' is indicated by the two-byte datatype `0x03 0x00`. The length of an ASCII string is given by the count of its containing tag structure. The ASCII string must be nul-terminated, i.e., its last character is "\0" (`0x00`). Data of type 'undefined' can have any kind of data, but we will interpret it as an ASCII string. Since it can have any kind of data, it may or may not be nul-terminated.

The offset_or_value field of a tag structure is a bit tricky. Normally, it is an offset into the TIFF file, identifying the position where the list of items are found. For instance, if its value is `0x00000040` (bytes `0x40 0x00 0x00 0x00`), then the list of items starts at the 64th byte of the TIFF file. (The datatype field of the tag structure reveals how long each item is, and the count field indicates how many items are in the list, so once you know the starting position of the list of items, you can read all the items.) However, as a special case, if the entire list of items fits in 4 bytes or less, the list is stored directly in the offset_or_value field (with zero bytes appended if needed to make it exactly 4 bytes long). For instance, if the datatype is 'byte' (`0x0001`) and the count is 3 and the three data bytes are `0x42 0x43 0x44`, then the offset_or_value field will contain the bytes `0x42 0x43 0x44 0x00`.

Here are the kinds of tag structures that you'll need to output, and their corresponding tagid:

name	tagid	description
DocumentName	`0x010D`	name of the document from which the image was obtained
ImageDescription	`0x010E`	a description of the image
Make	`0x010F`	make of the camera
Model	`0x0110`	model of the camera
Software	`0x0131`	software used to create the image
DateTime	`0x0132`	date and time of image creation
Artist	`0x013B`	creator of the image
HostComputer	`0x013C`	computer used to create the image
Copyright	`0x8298`	copyright notice
RelatedSoundFile	`0xA004`	name of a related audio file
DateTimeOriginal	`0x9003`	date and time of original image creation
DateTimeDigitized	`0x9004`	date and time when image was stored as digital data
MakerNote	`0x927C`	manufacturer-defined information
UserComment	`0x9286`	comments on the image from the creator
ImageUniqueID	`0xA420`	a unique identifier assigned to the image

(As a reminder, the tagid is a 2-byte integer that's stored in little-endian format in the file; for instance, the DocumentName tagid is stored as the two bytes `0x0D 01`.) Each of these tag structures stores an ASCII string, except for MakerNote and UserComment, which are of datatype 'undefined': i.e., a sequence of characters whose length is indicated by the count field of the tag structure (just like an ASCII string, except it's not nul-terminated). The UserComment tag structure is special: its data part starts with 8 bytes that indicate the character set (`0x41 0x53 0x43 0x49 0x49 0x00 0x00 0x00` indicates ASCII, and you can ignore all other values), followed by the actual user comment. Your program should output one line for each of the tag structures listed in the table above, in the format `Header: Value`, e.g., `Artist:  Dorothea Lange`.

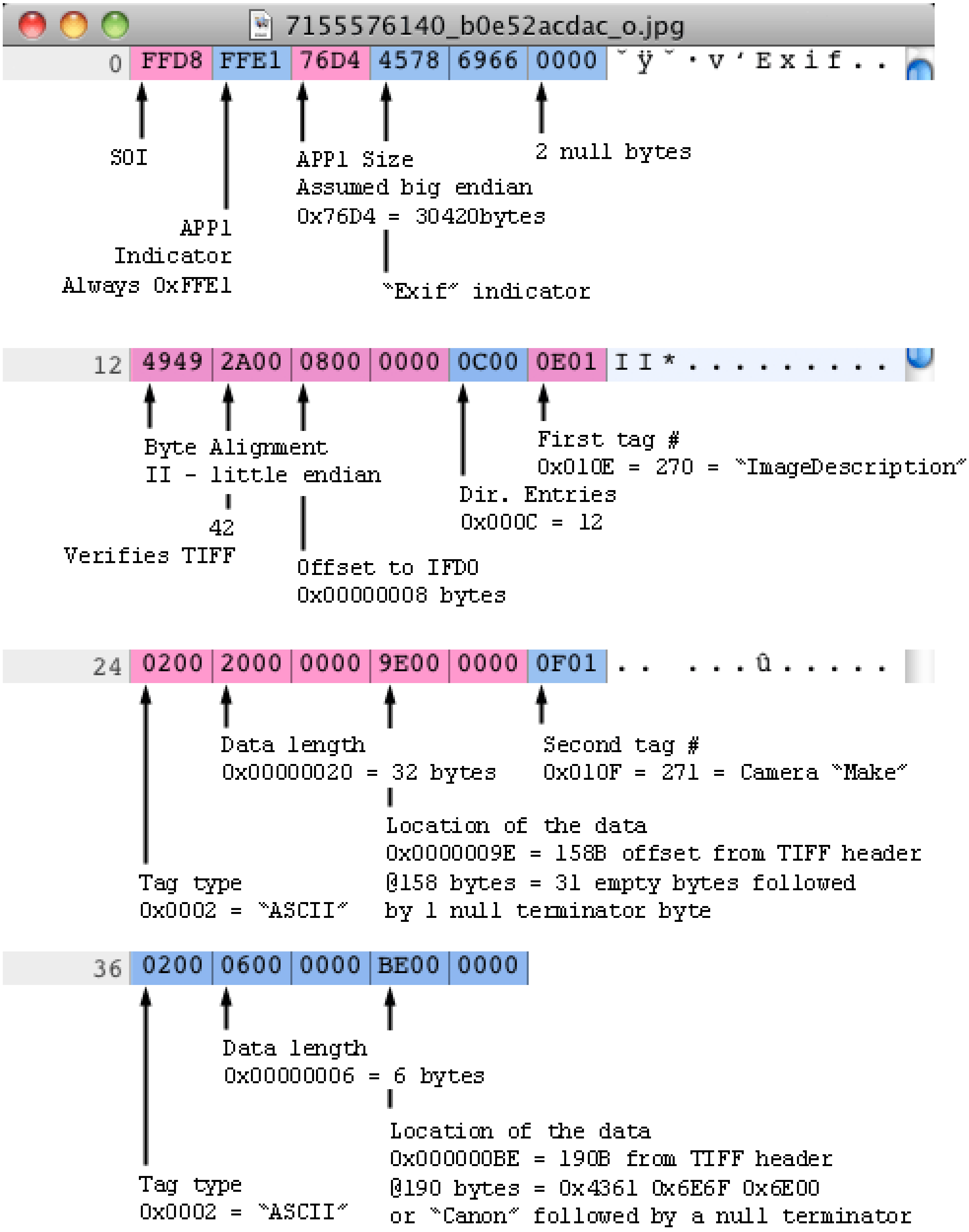
In addition, you need to know about the following tag structure, which should be present in the 0th IFD:

name	tagid	description
Exif IFD ptr	`0x8769`	start position of the Exif IFD, as an offset into the TIFF file

This tag structure contains a single long (a 4-byte integer) identifying the file offset where the Exif IFD starts. Since the offset fits into 4 bytes, it is stored directly in the offset_or_value field of the tag structure.

Your program should parse the TIFF header, find the location of the 0th IFD, parse the 0th IFD as a sequence of tag structures, print the value of all string-valued tag structures listed in the table above, look for the Exif IFD ptr tag structure in the 0th IFD, use that to find the location of the Exif IFD, parse the Exif IFD as a sequence of tag structures, and print the value of all string-valued tag structured listed in the table above.

For an example of a small JPEG file and the decoding of its first 46 bytes, see the figure on the next page. (Credit: Flickr.)



```
 0  FFD8 FFE1 76D4 4578 6966 0000  ˇÿˇ·v‘Exif..
12  4949 2A00 0800 0000 0C00 0E01  II*.........
24  0200 2000 0000 9E00 0000 0F01  .. ...û.....
36  0200 0600 0000 BE00 0000
```

SOI

APP1
Indicator
Always 0xFFE1

APP1 Size
Assumed big endian
0x76D4 = 30420bytes

"Exif" indicator

2 null bytes

Byte Alignment
II - little endian

42
Verifies TIFF

Offset to IFD0
0x00000008 bytes

Dir. Entries
0x000C = 12

First tag #
0x010E = 270 = "ImageDescription"

Data length
0x00000020 = 32 bytes

Second tag #
0x010F = 271 = Camera "Make"

Location of the data
0x0000009E = 158B offset from TIFF header
@158 bytes = 31 empty bytes followed
by 1 null terminator byte

Tag type
0x0002 = "ASCII"

Data length
0x00000006 = 6 bytes

Location of the data
0x000000BE = 190B from TIFF header
@190 bytes = 0x4361 0x6E6F 0x6E00
or "Canon" followed by a null terminator

Tag type
0x0002 = "ASCII"