Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations 1. Thesis and Dissertation Collection, all items

2007-09

Vulnerability analysis of HD photo image viewer applications

Juan, Clifford C.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/3285

Downloaded from NPS Archive: Calhoun

DUDLEY
KNOX
LIBRARY

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

VULNERABILITY ANALYSIS OF HD PHOTO IMAGE VIEWER APPLICATIONS

by

Clifford C. Juan

September 2007

Thesis Advisor:	James Bret Michael
Second Reader:	Christopher S. Eagle

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Vulnerability Analysis of HD Photo Image Viewer Applications	**5. FUNDING NUMBERS** **DUE-0114018** **CNS-0430566**
6. AUTHOR(S) Clifford C. Juan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Science Foundation Arlington, VA	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The introduction of Microsoft's new graphics file format, Windows Media Photo, into the mainstream market in 2006 has been one of the most interesting developments in the digital world. The file format, which has since been renamed to HD Photo in November of 2006, is being touted as the successor to the ubiquitous JPEG image format, as well as the eventual de facto standard in the digital photography market. With massive efforts already underway to increase the software support of this file format, to make available support for digital camera makers to incorporate it into their products, and to propose the file format to the Joint Photography Experts Group in order to make HD Photo as a standard itself, HD Photo is poised to become as widespread as any of the common image file formats today. This provides the motivation into studying whether the HD Photo file format can be used as a vehicle to compromise a user's system.

This work addresses the security of handling the HD Photo file format as it pertains to image viewer applications. Whenever an application is updated to accommodate a new file format, it is possible that the application in question can be vulnerable to exploitation. This is a concern, especially if a malformed instance of that file format can make the application to deviate from its specified behavior and cause the execution of arbitrary code. This thesis investigates if some of the existing applications today that render image files are susceptible to compromise by opening a malformed HD Photo image file.

The goal of this thesis is to test the security of various image viewer applications compatible with the HD Photo file format. We modified MiniFuzz, an automated fuzzing tool, to conduct mutation-based smart fuzzing and generation-based fuzzing. The test instrumentation worked correctly, but the test cases did not reveal any security vulnerabilities.

14. SUBJECT TERMS Information Assurance, Vulnerability Analysis, File Format Fuzzing, HD Photo Image File Format, Binary Parsing	**15. NUMBER OF PAGES** 205
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

VULNERABILITY ANALYSIS OF HD PHOTO IMAGE VIEWER APPLICATIONS

Clifford C. Juan
DoD Civilian, Naval Postgraduate School
B.S., University of Missouri – Kansas City, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2007

Author: Clifford C. Juan

Approved by: James Bret Michael
Thesis Advisor

Christopher S. Eagle
Second Reader/Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The introduction of Microsoft's new graphics file format, Windows Media Photo, into the mainstream market in 2006 has been one of the most interesting developments in the digital world. The file format, which has since been renamed to HD Photo in November of 2006, is being touted as the successor to the ubiquitous JPEG image format, as well as the eventual de facto standard in the digital photography market. With intensive efforts already underway to increase the software support of this file format, to make available support for digital camera makers to incorporate it into their products, and to propose the file format to the Joint Photography Experts Group in order to make HD Photo a standard itself, HD Photo is poised to become as widespread as any of the common image file formats today; this provides motivation for studying whether the HD Photo file format can be used as a vehicle to compromise a user's system.

This thesis addresses the security of handling the HD Photo file format as it pertains to image viewer applications. Whenever an application is updated to accommodate a new file format, it is possible that the application in question can be vulnerable to exploitation. This is a concern, especially if a malformed instance of that file format can cause the application to deviate from its specified behavior and cause the execution of arbitrary code. This thesis investigates whether some existing applications that render image files are susceptible to compromise by opening a malformed HD Photo image file.

The goal of this thesis is to test the security of various image viewer applications compatible with the HD Photo file format. We modified MiniFuzz, an automated fuzzing tool, to conduct mutation-based smart fuzzing and generation-based fuzzing. The test instrumentation worked correctly, but the test cases did not reveal any security vulnerabilities.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Michael and Prof. Eagle for their contributions of time, patience, and expertise without which this thesis would be possible. I would also like to thank the National Science Foundation for sponsoring the Federal CyberCorps program which made it possible for me to attend the Naval Postgraduate School.

This material is based upon work supported by the National Science Foundation under Grant No. DUE-00114018 and Grant No. CNS-0430566. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. WHAT IS WINDOWS MEDIA PHOTO (ALIAS HD PHOTO)?

In May of 2006, the Microsoft Corporation revealed in detail a new image file format named Windows Media Photo [Evers 2006]. Introduced by the software maker at the Windows Hardware Engineering Conference, Windows Media Photo is defined as a still image file format and associated state-of-the-art compression algorithm "specifically designed [for] all types of continuous tone photographic" images [HDPhotoFeatureSpec 2006]. The file format is the heart of Microsoft's campaign to target the digital photography market as part of an ongoing effort by the Redmond-based company touting Windows Media Photo as a superior graphics file format that will eventually succeed the ubiquitous Joint Photographic Expert Group (JPEG) file format, which is the *de facto* standard for today's digital images (both for digital cameras and for the Internet).

At the time of this writing, HD Photo and Windows Media Photo both refer to the same file format, and are often used interchangeably. As of August 2007, this file format has been under formal review for standardization by the Joint Photographic Expert Group standards body, under the name JPEG XR [Jurrien 2007].

B. FEATURES OF WINDOWS MEDIA PHOTO

Windows Media Photo possesses an array of features that makes it superior to and more attractive than the existing graphics file formats out there, especially in comparison with the JPEG format. One of the key features that Windows Media Photo offers is that it has a compression technology that is claimed to be superior to that of the JPEG format. For lossy compression, Windows Media Photo represents a given image of better quality than the JPEG format in approximately half the file size. Windows Media Photo's high rate of compression saves storage space while retaining much of the image's original quality, making this file format attractive to the digital imagery market. In addition to displaying an image with higher quality at a reduced amount of memory, Microsoft claims that Windows Media Photo has other advantages over the JPEG format. By storing 16 or 32 bits of data for each color (compared to 8 data bits per color for JPEG), it is easier for someone to distinguish subtle variation in tone or shadow details for a given image via Windows Media Photo.

For the emergent technology called high dynamic-range photography (HDR), which can capture significantly more image information than JPEG can represent, there is indication that this technology can catch on due to digital cameras under development that can directly create an HDR image. Currently, to make an image of equal quality, a user would have to take "a series of shots with different exposures" and then concatenate them all together into a composite image [Gruener 2007]. Unlike the JPEG format, Windows Media Photo provides support for 32-bit HDRs [Gruener 2007].

Windows Media Photo offers both lossless compression and lossy compression, whereas JPEG only offers lossy compression. Like the PNG format, Windows Media Photo can store the full information of an image taken directly from a digital camera's sensor (i.e., the image's "full dynamic range and color gamut data") [MacNN 2007]. Unlike the PNG format, however, Microsoft's file structure can represent a high-grade image at or near its original quality without the need to take up the same amount of memory as the PNG format because Windows Media Photo allows for significant improvement in compressing the image's size while preserving the fidelity of the original image [MacNN 2007]. This characteristic appeals to both professional photographers as well as casual digital camera users. Professional photographers will appreciate the file format's ability to directly store the raw data from the digital camera's sensors [Shankland 2007]. Amateur photographers will find appeal in Windows Media Photo through convenience; the file format is capable of being taken directly from the digital camera and stored into a personal computer without the hassle of photo maintenance and massive memory requirements that go along with raw images. All of these previously described features make Windows Media Photo along with its other features (such as efficient decoding for multiple resolutions and minimal overhead for format conversion or transformations), an appealing alternative to the JPEG and other existing image file structures of today [Microsoft-HDPhoto 2007].

C. POTENTIAL FOR WIDESPREAD USE OF HD PHOTO

Microsoft has made it no secret that one of its priorities is to intensively promote the adoption of Windows Media Photo. The software maker renamed it to HD Photo to denote neutrality, contrary of its former name partisan to Microsoft, and appeal to the mainstream (hoping that the "HD" annotation will lead adopters to associate the file format with "high definition," such as that of HDTV [Shankland 2007]). To achieve this lofty goal of making the file format commonplace, Microsoft has included native support for HD Photo in Windows Vista, its latest operating system as of 2007. It has also made available support for HD Photo in Windows XP as well as in frequently used devices and platforms other than Windows that are web-accessible. This puts HD Photo in a good position to become one of the dominant image formats, because of the vast number of digital images stored throughout the Internet as well as in personal computers.

Microsoft is also leaning towards having digital camera makers adopt their file format into their cameras by designing HD Photo to work in conjunction with microchips used for image-processing in digital cameras [Shankland 2007]. In modern times, the trend has been such that more people prefer to store their photos in digital format rather than in traditional rolls of film, which is part of the reason (in addition to other factors such as people keeping up with the technological trend and the convenience of digital cameras) why the use of digital cameras has boomed. If Microsoft is successful at influencing digital camera vendors into having HD Photo built-in for use with their cameras, this will drastically increase the presence of HD Photo among the mainstream.

The fact that this file format is supported by widely used operating systems such as Windows (Windows XP and Windows Vista in particular) and the popular Mac OS X indicates that it has the possibility of being as pervasive as JPEG itself. Microsoft dominates the personal computer market, with the Microsoft Windows range of operating systems alone commanding ninety-seven percent (97%) of the total market share [MKH 2007]. The Microsoft Windows operating systems are also the most frequently used operating systems for computers that run web browsers [MKH 2007], with Microsoft Internet Explorer being the most popular browser used for accessing the Internet [SANS

2007]. With support on these platforms, HD Photo could have a formidable presence in the vast majority of the world's computer market.

What gives HD Photo even more of a possibility to be as widespread as the JPEG is Microsoft taking the action of going to an international standards organization to make HD Photo a neutral industry standard. Microsoft has taken the additional steps of lowering the barriers standing in the way of supporting HD Photo as a possible standard. Microsoft has made a device porting kit available to manufacturers so they can "add HD Photo support in devices and to other platforms" [QuickOnlineTips 2007]. Microsoft also loosened restrictions on its licensing policy (such as dropping fees) and added the file format's technology under the Open Specification Promise, which stipulates that Microsoft will not assert patent rights over HD Photo, all in hopes of making the file format more acceptable to viable competitors and open-source programmers [Shankland 2007].

Standardizing this image format will lead to motivating the printer, camera, and image application vendors to use HD Photo. It will strongly encourage the customers and the end users associated with these vendors (which make up the majority of the mainstream market) to adopt HD Photo for everyday use. By meeting these objectives, Microsoft hopes that this will result in HD Photo having the greatest potential to become as commonplace as JPEG.

D. NEED FOR VULNERABILITY ANALYSIS OF HD PHOTO FILE STRUCTURE AND COMPATIBLE IMAGE VIEWER APPLICATIONS

The possibility of HD Photo being the eventual successor of the JPEG format, combined with Windows XP already in use with many systems belonging to government agencies and Windows Vista being used for the next generation of systems within the Department of Defense, makes HD Photo of great interest to research if this file structure is vulnerable to carrying a malicious payload or carrying hidden information within its metadata. Thus, it is necessary to research whether the image viewer applications used today are vulnerable to exploits via HD Photo. If image viewer applications are vulnerable to such exploits, then the follow-on research question is what are the countermeasures that someone can take to lessen or eliminate the vulnerabilities.

Since HD Photo is relatively new (at the time of this writing) in comparison to JPEG and other commonly used file formats, it is not well known (outside of the Microsoft development team charged with developing and testing the HD Photo format) whether this file structure provides any vulnerabilities that, if exploited, can result in the compromise of a system. Unlike the JPEG (which has been studied extensively with respect to information hiding), it is not well known how resistant this file format is in regards to steganography. Thus, there are no software toolsets or methodologies available for conducting a vulnerability assessment study of HD Photo. Software toolsets and methods will need to be developed for this purpose. In addition, certain existing software toolsets and methods used for vulnerability discovery and exploitation may need to be used (and possibly modified) to discover vulnerabilities.

E. BENEFITS OF HD PHOTO VULNERABILITY RESEARCH

There are several benefits to be obtained from researching the HD Photo file format. First, a vulnerability assessment may uncover ways to exploit HD Photo, providing information which can be used to counter such exploits. Second, the research may uncover steganographic means for accomplishing information hiding in this file structure. Third, the results of this study will help determine what security measures, if necessary, may need to be taken by the Department of Defense in regards to this image file format. Fourth, the prototype software tools developed for examining these files will also be useful to the Department of Defense.

F. BRIEF DESCRIPTION OF THE REMAINING CHAPTERS OF THE THESIS

The first chapter of this thesis provides the introduction of HD Photo and the reasoning behind why it is of interest to look for security vulnerabilities in the file format. The second chapter provides background information about the file structure of the HD Photo file format. The third chapter covers a security software testing technique called “fuzzing,” and how it applies to file formats. The fourth chapter treats the development methodology in testing image viewer applications. The first part of this chapter presents a walkthrough, or handtrace, of a sample HD Photo image.   After completion of the handtrace of the HD Photo sample image, the results of the walkthrough are used to develop test cases with respect to file format fuzzing and HD Photo.

The fifth chapter covers the selection of existing image viewer applications used to open and display HD Photo files, and examining them in a vulnerability assessment study. Included in this chapter is the justification of the chosen toolsets used to conduct a vulnerability assessment on the image applications compatible with HD Photo. This chapter also describes the experimental setup to support the use of these toolsets and the image viewer applications being studied. The experimental procedures are detailed in this chapter, along with an explanation of how they relate to the testing methodologies covered in the previous chapter.

The sixth chapter provides the results and analysis of the experiments and tests conducted during the vulnerability assessment and discovery phase. The final chapter presents a summary of the results and findings, in addition to recommendations for future work.

II. BACKGROUND – THE FILE STRUCTURE OF THE HD PHOTO FILE FORMAT

N.B.: The information on the HD Photo file format presented in this chapter has been primarily taken from the official documents by Microsoft. These documents are named "HD Photo Feature Specification" [HDPhotoFeatureSpec 2006] and "HD Photo Bitstream Specification" [HDPhotoBitstreamSpec 2006], respectively. All other references will be noted as usual. Numerical values expressed in hexadecimal will be expressed using the prefix "**0x**" followed by the hexadecimal value itself. For example, the hexadecimal equivalent of the decimal number thirty-two (32) is expressed in hexadecimal as **0x20**.

A. OVERVIEW OF HD PHOTO

HD Photo is an image file format and a still image compression algorithm developed by Microsoft. For the scope of this document with respect to vulnerability assessment, only the file format itself will be covered. The reason for this is because it is the contents of the file itself that will be studied for the vulnerability assessment portion of this thesis, and not the compression algorithm. The compression algorithm of HD Photo will be referred to later in this report when comparing it against the compression algorithm of the JPEG with respect to anti-steganography. In this context, anti-steganography would refer to the effectiveness in destroying the data that represent hidden messages inside a medium itself (in this case, the JPEG or HD Photo image), to an extent such that the recipient of the messages cannot recover the data embedded inside because the decoded message(s) would be unintelligible.

In a manner that is very similar to the Tagged Image File Format (TIFF), the HD Photo file format is defined by a container, which holds one or more elementary bitstreams [HDPhotoBitstreamSpec 2006]. The HD Photo Container is a higher-level structure which holds descriptive metadata (i.e., *data about the HD Photo file, not the image data itself or the data that represents the actual image*). It is the metadata of an HD Photo file that enables the HD Photo image viewer application to understand how to handle and manage that particular HD Photo file itself (e.g., what type of image pixel format that the specific image in the HD Photo file utilizes, the location of the image data in the HD Photo file, the values for height and width in the image, how many images are contained in the HD Photo file if it is a multi-image file).

The HD Photo metadata can be purely descriptive as well. This includes optional information fields used by applications for document storage and retrieval, such as ***DocumentName*** (the name of the document from which the image was scanned), ***Make*** (the manufacturer of the scanner, video digitizer, or other type of equipment used to generate the image), ***Model*** (the model name or number of the scanner used to generate the image), ***PageName*** (the name of the page from which this image was scanned), and ***PageNumber*** (the page number of the page from which this image was scanned) [TIFF3 1992]. The metadata can also include other information about the image, such as the complete copyright notice of the person or organization making a copyright claim to the image (***Copyright***), and the data and time that the image was created (***DateTime***).

As mentioned previously, the HD Photo Container also contains at least one elementary bitstream. For each image found within an HD Photo file, there is an elementary bitstream in the HD Photo file that defines it [HDPhotoFeatureSpec 2006]. This bitstream represents the actual image data. This chapter will focus on the data contents of both the HD Photo Container and the HD Photo Bitstream, as well as how each one is physically structured.

B. HD PHOTO FILE HEADER

In the HD Photo file format, the HD Photo Container is composed of three key components. The first component is the HD Photo File Header. Just like the Image File Header (its counterpart in the TIFF format), the HD Photo File Header is an 8-byte structure and is the only part of the file known to always have a fixed location. This component can always be found within the first eight bytes of an HD Photo file, all HD Photo files must begin with the HD Photo File Header. The purpose of an HD Photo File header is to identify the file as an HD Photo image and to point to an Image File Directory (IFD), which contains further information about the image itself. Figure 1 summarizes the meaning of each of the eight bytes in the header.

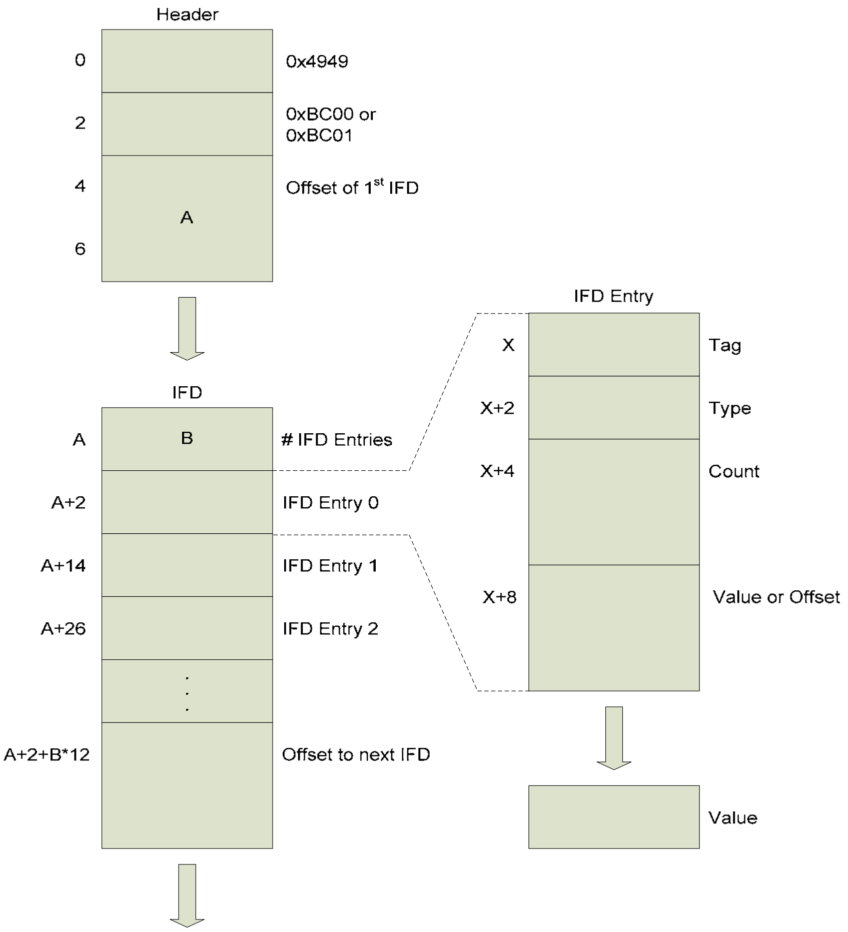


Figure 1. HD Photo File Header and IFD Table Structure, taken from [HDPhotoFeatureSpec 2006]

The first two bytes of the HD Photo File Header represents the HD Photo equivalent of the TIFF **identifier**, which indicates the byte-order (or "endianness") used in reading the file. The TIFF 6.0 format accepted legal values of **0x4949** (**"II"** in ASCII) or **0x4D4D** ("**MM**" in ASCII) for the **identifier**. The hexadecimal value **0x4949** indicates that the image is written in little-endian (known as the Intel format), in which the least significant byte is written at the lowest address [TIFF1 2007]. The hexadecimal value **0x4D4D** indicates that the image is written in big-endian encoding (also known as the Motorola format), in which the most significant byte is written at the highest address [TIFF1 2007]. Since the HD Photo file format supports only the Intel format (or little-endian encoding), the **identifier** value will always be **0x4949** or "**II**". Unlike the TIFF identifier, the HD Photo identifier will never have a value of **0x4D4D** or "**MM**".

The third byte of the header represents “a unique byte value” that distinguishes this image file as an HD Photo file as opposed to a TIFF file [HDPhotoFeatureSpec 2006]. This byte will always have a value of **0xBC** (188 in decimal). Microsoft notes that although it would have been idealistic to make this identification field longer, the constraints of having the HD Photo File Header retain compatibility with the Image File Header of the TIFF limits this field to one byte [HDPhotoFeatureSpec 2006].

The **version number** is found in the fourth byte of the HD Photo File Header. Presently, only two values are legally accepted: 0x00 (or 0 in decimal) and 0x01 (or 1 in decimal). Since a version number of 0 indicates which files have been made during the pre-release development phase, an HD Photo file of version 0.0 may have a bitstream containing data that is not compatible with the final version (1.0) of the HD Photo format [HDPhotoFeatureSpec 2006]. An HD Photo image of version 1.0 must meet the requirements dictated by the HD Photo Specification documents released for version 1.0 [HDPhotoFeatureSpec 2006]. Numerical values greater than 1 are being reserved for future versions of HD Photo, which may interpret information of the HD Photo image differently than the current version. Thus, all applications written for decoding HD Photo images “must reject any files with a version number greater than 1” [HDPhotoFeatureSpec 2006].

The last four bytes of the HD Photo File Header represent the offset to the first **Image File Directory (IFD) -** the second key component of the HD Photo file structure. In the context of the HD Photo File Header, the term offset refers to a location in the file with respect to the start of the HD Photo file, so the beginning of the HD Photo file is indicated with an offset of zero [HDPhotoFeatureSpec 2006]. In this case, the offset to the first IFD of the HD Photo file can be located anywhere in the image, as long as it meets both of the following conditions:

- The first IFD is located after the HD Photo File Header.
- The first IFD begins on a word boundary, which is an offset that divides two 16-bit quantities. This means that the IFD starts at the beginning of an even-numbered byte. For example, the file offset 0x08 beings on a word boundary.

The first condition means that the first IFD must not start before file offset **0x08**. The second condition indicates that the offset of the first IFD (or the offset of any other IFD inside the image) must be divisible by two (2). For example, the file offset of the first IFD can be **0x08**, but not **0x09**. The reason for word-alignment is to provide ease for an application to read the format.

C. THE IMAGE FILE DIRECTORY

The Image File Directory (**IFD**) is the second key component which makes up the very core of the HD Photo file structure. The IFD can be thought of as a container that holds information about the image, as well as pointers to the image data itself [TIFF3 1992]. Similar to a multi-image TIFF file, if there is more than one image contained in an HD Photo file, there is an IFD defined for each image in the file. Thus, more than one IFD can be found inside an HD Photo file.

Structurally, an IFD is organized as a table of one or more **directory entries** or **fields**. (N.B.: In this document, the terms "field" and "directory entry" are used interchangeably with respect to the HD Photo Image File Directory. They both refer to the same object.) A mirror image of the TIFF IFD **field** found in the TIFF 6.0 format, the HD Photo IFD **field** is a 12-byte record that describes a specific aspect about the image data itself. An IFD field may contain any type of data, and the HD Photo Feature Specification document lists both mandatory and optional fields that may be found in an HD Photo image. Thus, an IFD can be likened to a road map; it indicates where all of the data associated with an image can be found inside the HD Photo file [TIFF3 1992].

The IFD begins with a 2-byte count that indicates the number of directory entries that can be found inside that particular IFD. In the example in Figure 1, the number of IFD entries can be found at offset **A**, which stands for the number of bytes away from the beginning of the HD Photo file. The number of IFD directory entries is followed by a sequence of "**B"** IFD entries themselves, where **B** is the number of IFD directory entries as denoted in Figure 1. After the last IFD entry is a 4-byte offset to the next IFD. In Figure 1, the offset to the next IFD is found at offset **A+2+B*12**. Thus, the size of the IFD in bytes is determined by the total count of the directory entries inside the IFD (two bytes), the sequence of the 12-byte directory entries that follow (the number of directory entries multiplied by twelve bytes), and the offset to the next IFD (four bytes). If the IFD is the last one in the HD Photo file, then the last IFD entry is followed by four null bytes instead (a value of zero).

D. THE IFD ENTRY

The third key component of the HD Photo Container is the fundamental building block of the IFD: the **IFD entry**. Also known as the directory entry, this data structure is the reason behind the flexibility of the HD Photo format. As mentioned previously, the IFD entry is 12 bytes in length. The purpose of each entry in the IFD is to define a specific piece of information about the image, such as its width and height.

Each IFD entry is defined in the format as shown in Figure 1. The IFD entry is divided into four segments: the field **tag**, the field **type**, the field **count**, and the field **value/offset**. The first two bytes of an IFD entry (bytes 0 and 1) represent the field's **tag**. The **tag** value serves as the primary means of identification for the field. For example, in HD Photo the **tag** value of **0xBC01** (48129 in decimal) indicates that the IFD entry identifies the pixel format of the image [HDPhotoFeatureSpec 2006]. The tags specific to HD Photo are defined in Chapter 3 of the HD Photo Feature Specification document. To aid further understanding of how HD Photo tags work, some of these tags will be explained in detail later on in this thesis, using a handtrace demonstration of a sample HD Photo image. The **tag** value also determines the location of the IFD entries contained within the IFD itself. It is a requirement in the HD Photo specification that all entries inside the IFD must be sorted in ascending order with respect to their **tag** value [HDPhotoFeatureSpec 2006].

The next two bytes in the IFD entry (bytes 2 and 3) indicate the field **type**. The value of the field **type** indicates the type of data contained in the field **value/offset**. The meaning of the field **types** and their **sizes** are listed below [HDPhotoFeatureSpec 2006]:

Field type value	Description of data type
1 = BYTE	8-bit unsigned integer
2 = ASCII	8-bit byte that contains a 7-bit ASCII code; the last byte must be NUL (binary zero)
3 = SHORT	16-bit (2-byte) unsigned integer in little-endian (LSB first) byte order
4 = LONG	32-bit (4-byte) unsigned integer in little-endian (LSB first) byte order
5 = RATIONAL	Two LONGs: the first represents the numerator of a fraction; the second, the denominator, both in little-endian (LSB first) byte order.
6 = SBYTE	An 8-bit signed (twos-complement) integer.
7 = UNDEFINED	An 8-bit byte that may contain anything, depending on the definition of the field.
8 = SSHORT	A 16-bit (2-byte) signed (twos-complement) integer in little-endian (LSB first) byte order.
9 = SLONG	A 32-bit (4-byte) signed (twos-complement) integer in little-endian (LSB first) byte order.
10 = SRATIONAL	Two SLONG's: the first represents the numerator of a fraction, the second the denominator, both in little-endian (LSB first) byte order.
11 = FLOAT	Single precision (4-byte) IEEE format in little-endian (LSB first) byte order.
12 = DOUBLE	Double precision (4-byte) IEEE format in little-endian (LSB first) byte order.

As seen above, the numeric value assigned to the field **type** determines what kind of data that the field **value/offset** contains. For example, assigning a decimal value of 4 to the field **type** indicates that the field's **value** is a 32-bit (4-byte) unsigned integer in little-endian (LSB first) byte order.

The next four bytes in an IFD entry (bytes 4 through 7) represent the field **count**. The **count** indicates the total number of values in the field. For example, if the field **type** is a single 32-bit word (LONG), then a **count** of four (4) means that the field's **value** consists of four 32-bit words (or four LONGs). As noted in [HDPhotoFeatureSpec 2006] and [TIFF3 1992], there are some important points of consideration with respect to the field **type** and the field **count**. These notes are listed below.

- "The value of the Count part of an ASCII field entry includes the NUL. If padding is necessary, the Count does not include the pad byte. Note that there is no initial 'count byte' as in Pascal-style strings" [HDPhotoFeatureSpec 2006, TIFF3 1992].

- "Any ASCII field can contain multiple strings, each terminated with a NUL. A single string is preferred whenever possible. The Count for multi-string fields is the number of bytes in all the strings in that field plus their terminating NUL bytes. Only one NUL is allowed between strings, so that the strings following the first string will often begin on an odd byte" [HDPhotoFeatureSpec 2006, TIFF3 1992].

- "The reader must check the type to verify that it contains an expected value. HD Photo currently allows more than 1 valid type for some fields. For example, ***ImageWidth*** and ***ImageLength*** are usually specified as having type SHORT. But photos with more than 64K rows or columns must use the LONG field type" [HDPhotoFeatureSpec 2006, TIFF3 1992].

- "HD Photo readers must accept BYTE, SHORT, or LONG values for any unsigned integer field. This allows a single procedure to retrieve any integer value, makes reading more robust, and saves disk space in some situations." [HDPhotoFeatureSpec 2006, TIFF3 1992].

- "It is possible that other HD Photo field types will be added in the future. Readers must not try to interpret fields containing an unexpected field type" [HDPhotoFeatureSpec 2006].

There is a discrepancy in the third note referenced from [HDPhotoFeatureSpec 2006] with respect to the ***ImageWidth*** and ***ImageLength*** tags. The HD Photo file format does not use the original TIFF metadata tags ***ImageWidth*** and ***ImageLength*** to indicate the number of rows (i.e., scanlines) in the image and the number of columns in the image (i.e., the number of pixels per scanline), respectively. Instead of using the TIFF ***ImageWidth*** and ***ImageLength*** tags (which accept either SHORT or LONG in the field type), the HD Photo format uses its own set of metadata tags to indicate the number of rows and columns in the image. These tags, called ***ImageWidth*** and ***ImageHeight*** respectively, only accept a field type of LONG and have different tag values from its TIFF counterparts. Whereas the TIFF tags ***ImageWidth*** and ***ImageLength*** have tag values of **0x100** (256 in decimal) and **0x101** (257 in decimal) respectively, the HD Photo tags ***ImageWidth*** and ***ImageHeight*** have tag values of **0xBC80** (48256 in decimal) and **0xBC81** (48257 in decimal), respectively.

The last four bytes in the IFD entry (bytes 8 through 11) correspond to either the field's actual **value** or the **offset** to its actual value. This segment is called the **value/offset** for that reason. If the **value** of an IFD entry field can fit within 4 bytes, then the field's **value** immediately follows the field **count**. Otherwise, the value stored in **value/offset** contains the file offset where the reader can find the value of that field inside the HD Photo file. As noted in [HDPhotoFeatureSpec 2006] and [TIFF3 1992], the field's **type** and **count** will determine whether or not the field's value will fit within the 4 bytes allocated at the end of an IFD entry.

Although most fields only contain a single value (i.e., an IFD directory entry's count is frequently set to 1 in HD Photo), all fields can be treated as one-dimensional arrays, since each directory entry has an associated **count**. In the case where a complex data structure is used as the value of the field itself, the directory entry would have to set the field **type** to UNDEFINED and set the field **count** to the number of bytes needed to hold that data structure [HDPhotoFeatureSpec 2006, TIFF3 1992]. This makes it possible to add other HD Photo field types to existing tags, as well as define field types for HD Photo tags that may be developed for future versions of the file format.

E. THE HD PHOTO BITSTREAM

The core component that complements the HD Photo Container in Microsoft's HD Photo image file format is the **HD Photo Bitstream**. This contiguous data structure is what defines an image stored in an HD Photo file. The abstract structure of the HD Photo Bitstream can be classified into three parts: the **Image Header** (IMG_HDR), the **Index Table** (INDEXTBL), and a sequence of one or more tiles that represent the HD Photo image. These parts of the HD Photo Bitstream are illustrated in Figure 2.

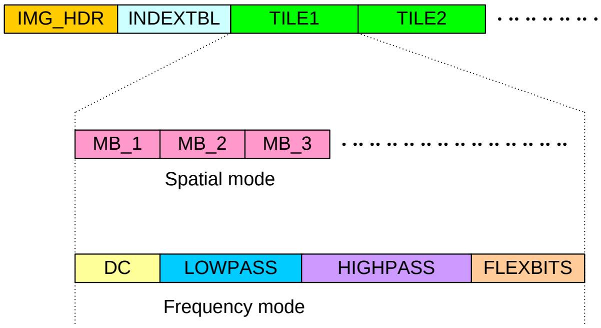


Figure 2. Layout of HD Photo bitstream. The Image Header (IMG_HDR) is followed by an index table (INDEXTBL), followed by a sequence of tiles that are in Spatial mode or Frequency mode. The mode determines the structural layout of the tile, as evidently seen above. Taken from [HDPhotoBitstreamSpec 2006]

Just like the HD Photo File Header will always be at the beginning the HD Photo file (and also the beginning of the HD Photo Container), the **Image Header** will always mark the start of the HD Photo Bitstream. The **Image Header** contains pertinent information about the image itself. Among the type of data it holds is how wide the image is in pixels, how high (or how long) the image is in pixels, whether or not the image uses tiling, what mode the bitstream layout is using, whether overlapping is present, whether or not an interleaved alpha channel is present in the bitstream, and so forth. If the image uses tiling, the **Image Header** contains information about each of the tiles themselves (i.e., the height and width of each tile).

The next entity that follows the Image Header in the HD Photo Bitstream is the **Index Table**. The **Index Table** contains information about each location of the tiles used in the image. Thus, for each tile in the image, there is an entry in the Index Table that corresponds to the offset of that tile from the beginning of the encoded image data. [HDPhotoBitstreamSpec 2006]. The Index Table will only exist in the HD Photo Bitstream if the image uses tiling. If the image doesn't use tiling, then it is said to be "untiled" and the image will be represented by a single tile. In this case, the only tile-frequency packet will have an offset of zero.

The physical layout of the **tile** in HD Photo is determined by the mode, as shown in Figure 2. In the Frequency Mode, the bitstream layout of a tile is laid out as a hierarchy of bands [HDPhotoBitstreamSpec 2006]. The first band, called the **DC** band, carries information about the DC coefficient of each macroblock, in raster scan order. The second band, called the **Lowpass** Band, carries information of the lowpass coefficients of each macroblock (for each color plane, with some exceptions). The third band, called the **AC** band, carries information about the remaining 240 coefficients of each macroblock color plane (with some exceptions). The fourth and final band is called **Flexbits**, which is an optional layer that carries information regarding the low order bits of the AC coefficients, particularly for lossless and low loss cases [HDPhotoBitstreamSpec 2006]. The bitstream layout of each tile differs in the Spatial Mode; in this mode the bitstream layout of a tile is laid out in macroblock order, with the compressed bits corresponding to its respective macroblock being located together in the bitstream. In Figure 2, the first macroblock of the image is labeled as **MB_1**, the second macroblock labeled as **MB_2**, the third macroblock labeled as **MB_3**, and so on.

To understand the visual physical layout of a tile in HD Photo and how tiles work in this file format, a look at the visual image structure of HD Photo is warranted. This understanding is important when the data contents of the HD Photo bitstream will be covered. A typical HD Photo image can be composed of multiple color planes or channels. If the HD Photo image is a grayscale image, it would consist of just a single plane, called the **luminance channel**. The **luminance channel** corresponds to the image's panchromatic or grayscale rendering, and can be internally generated by the color conversion process from the RGB (red / green blue) input of the original source [HDPhotoBitstreamSpec 2005]. If the original source is a YUV image, then the luminance channel would represent the Y color plane in the HD Photo image. If the original source is a grayscale image, then the luminance channel simply represents the image itself.

The **chrominance channels** correspond to the “difference planes that contain the color information of the image” [HDPhotoBitstreamSpec 2005]. An image typically has two chrominance channels, with some specific exceptions such as the grayscale image format (it has no chrominance channels) and the n-channel image formats (it has n-1 chrominance channels). Chrominance channels can also differ in size for specific image types.

Each HD Photo image is arranged as a series of four-by-four (4x4) group of blocks, with each **block** being a four-by-four group of pixels. In HD Photo, blocks are horizontally aligned by 4-pixel offsets from the left of the image and vertically aligned by 4-pixel offsets from the top of the image [HDPhotoBitstreamSpec 2006]. Unless otherwise indicated, in this thesis a block refers to a four-by-four pixel group. In HD Photo, a group of four-by-four blocks is called a **macroblock**. The core structural components of HD Photo’s visual image structure, macroblocks are chunks of sixteen-by-sixteen pixels across all color channels used by the image. All HD Photo images are aligned by integer macroblock multiples; this means that for any image whose width or height in pixels is not a multiple of sixteen, HD Photo extends the image width or height boundary so that the width or height in pixels is extended to the next highest multiple of sixteen. Unless otherwise indicated, in this thesis a macroblock refers to a sixteen-by-sixteen pixel group, or the equivalent of a four-by-four block group. For certain color formats such as YUV420 and YUV422, a macroblock will have a different size with respect to the chrominance channels. A macroblock will be a 8-by-8 pixel chunk in YUV420, and a 16-by-8 pixel chunk in YUV422. (N.B.: This thesis primary treats the handling and the processing of an HD Photo file itself by an image viewer application, rather than the visual rendering of the HD Photo image.)

In an HD Photo image, blocks do not overlap, and they cover the entire image, even if doing so results in spilling over the boundaries of the image itself. In the case where blocks go past the image boundaries, the values of the pixels in these blocks are determined either through extension or extrapolation [HDPhotoBitstreamSpec 2006].

Figure 3 shows the visual image structure hierarchy of HD Photo. In HD Photo, macroblocks are collocated into structures called tiles, and these tiles are the components that form a regular pattern on the HD Photo image [HDPhotoBitstreamSpec 2006]. All tiles in the same horizontal row have the same height and are aligned with one another, and all tiles in the same vertical column have the same width and are aligned with one another. In addition to meeting the previously stated requirement, tiles can be of arbitrary size. However, the size of the tile must be a multiple of 16 macroblocks (i.e., the tile must be macroblock aligned) [HDPhotoBitstreamSpec 2006].

Figure 3 illustrates how tiles work in HD Photo, as well as how the composition of a tile can be broken down into blocks and macroblocks. The typical HD Photo image can range from one tile to 256 tiles in width, and range from one tile to 256 tiles in height. This means an HD Photo image contains at least one tile and at most 65,536 tiles (obtained simply by multiplying 256 tiles by 256 tiles).

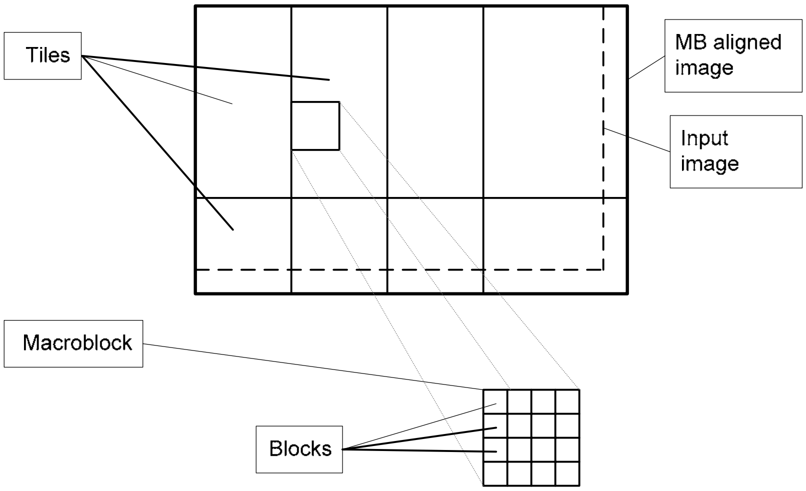


Figure 3. The Visual Image Structure Hierarchy of HD Photo, showing the composition of a tile within an HD Photo image. Taken from [HDPhotoBitstreamSpec 2006]

For the sample HD Photo image displayed in Figure 3, the big rectangle outlined with bold lines represents the macroblock aligned extrapolated HD Photo image, with the dashed lines representing the original image edges on the left and the bottom. The sample image employs a 2-by-4 tiling pattern, and the tile decomposed above is located on the first row and the second column of the image. The tile appears to be two macroblocks in width and four macroblocks in height.

When looking at the HD Photo Bitstream in great detail (i.e., the data contents that make up the HD Photo Bitstream itself), the physical bitstream layout of this file format consists of a hierarchy of layers, as documented in [HDPhotoBitstreamSpec 2006]. There are four layers attributed to this bitstream. They are the **Image Layer**, the **Tile Layer**, the **Macroblock Layer**, and the **Block Layer**. The main two layers of the HD Photo file format are the Image Layer and the Tile Layer, and these two layers can always be found in the HD Photo Bitstream.

The HD Photo Bitstream Specification explains how to interpret each syntax element found in the HD Photo Bitstream:

> The bitstream layout is defined by a set of tables. Each row of the table describes one step of pseudo-code, or the decoding of one syntax element. In the latter case, the syntax element is spelt out in the first column. Syntax elements are labeled by upper case entries, whereas functions are labeled in mixed case. The number of bits required to decode this element is specified in the 'Num bits' column. The 'Reference' column carries more information about the interpretation and semantics of the syntax element, and the decoding process when corresponding 'Num bits' is labeled as 'Variable.'
>
> The 'Descriptor' column of syntax element entries contain one of three terms: 'bool', 'uimsbf', or 'struct'. The descriptor 'bool' indicates that the syntax element is a Boolean variable represented with one bit. The 'true' value corresponds to the bit being set (i.e., equal to 1) and 'false' is otherwise. The descriptor 'uimsbf' expands to unsigned integer, most significant bit first. This refers to the decoding of bits, starting with the most significant bit being decoded first. The value of the pattern of ***k*** bits (***k*** being specified in the 'Num bits' column, or accompanying text) is interpreted as an unsigned integer and assigned to the variable in the first column. The descriptor 'struct' means that the syntax element is a structure of other component syntax elements. Such elements are further broken up in other referenced tables. [HDPhotoBitstreamSpec 2006]

In the HD Photo Bitstream Specification tables that describe the bitstream layout, the 'Reference' column serves as a directory where one can find a further explanation on the semantics and interpretation of each bitstream syntax element in the specification document. For the tables that follow, the ‘Reference’ column will refer to a syntax element’s item number (or table number) where one can find the semantics and interpretation of that specific syntax element in this very document. For the convenience of the reader, in the tables that follow, each HD Photo Bitstream syntax element is shown in bold letters.

Only the **Image Layer** and **Tile Layer** of the HD Photo Bitstream will be covered in detail in this thesis because only these two layers are useful for security testing and fuzzing experiments on the HD Photo file format, especially in regards to smart fuzzing. (Fuzzing will be covered later in this document.)

The **Image Layer** is defined in Table 1 below. The **Image Layer** consists of four syntax elements: the **Image Header** (labeled as IMAGE_HEADER), the **Image Plane Header** (labeled as IMAGE_PLANE_HEADER), the **Index Table** (labeled as INDEX_TABLE), and at least one or more instances of the **Tile Layer** (labeled as TILE). The variable name ***NumberIndexTableEntries*** is used to store the tile count, or the number of tiles in the HD Photo image. It is later used to read each instance of a Tile Layer found in the HD Photo Bitstream. If an alpha channel is present in the image, there is a separate Image Plane Header for that alpha channel in the Image Layer.

IMAGE (){	***Num bits***	***Descriptor***	***Reference***
IMAGE_HEADER	Variable	Struct	Table 2
alphaPlane = false **IMAGE_PLANE_HEADER**	Variable	struct	Table 3

If (ALPHACHANNEL_PRESENT) {
alphaPlane = true
IMAGE_PLANE_HEADER
	Variable	struct	Table 3

} **INDEX_TABLE**	Variable	struct	Table 4
For (n = 0; n < NumberIndexTableEntries; n++) { **TILE**(n)	Variable	struct	Table 6
} }			

Table 1. The Image Layer (Taken from [HDPhotoBitstreamSpec 2006])

1. Image Header

The first component of the Image Layer is the **Image Header** structure (labeled below as IMAGE_HEADER). The Image Header structure always marks the beginning of a valid HD Photo Bitstream. Originally referred to in [HDPhotoBitstreamSpec 2006], it is reproduced in this document for convenience as Table 2.

IMAGE_HEADER (){	*Num Bits*	*Descriptor*	*Reference*
GDISIGNATURE	64	uimsbf	(Item 1.1)
VERSION	4	uimsbf	(Item 1.2)
SUBVERSION	4	uimsbf	(Item 1.3)
TILING_FLAG	1	bool	(Item 1.4)
BITSTREAMFORMAT	1	uimsbf	(Item 1.5)
ORIENTATION	3	uimsbf	(Item 1.6)
INDEXTABLE_PRESENT_FLAG	1	bool	(Item 1.7)
OVERLAP	2	uimsbf	(Item 1.8)
SHORT_HEADER_FLAG	1	bool	(Item 1.9)
LONG_WORD_FLAG	1	bool	(Item 1.10)
WINDOWING_FLAG	1	bool	(Item 1.11)
TRIM_FLEXBITS_FLAG	1	bool	(Item 1.12)
TILE_STRETCH_FLAG	1	bool	(Item 1.13)
RESERVED	2	uimsbf	(Item 1.14)
ALPHACHANNEL_FLAG	1	bool	(Item 1.15)
SOURCE_CLR_FMT	4	uimsbf	(Item 1.16)
SOURCE_BITDEPTH	4	uimsbf	(Item 1.17)
if (SHORT_HEADER_FLAG) {			
WIDTH_MINUS1	16	uimsbf	(Item 1.18)
HEIGHT_MINUS1	16	uimsbf	(Item 1.19)
}			
else {			
WIDTH_MINUS1	32	uimsbf	(Item 1.18)
HEIGHT_MINUS1	32	uimsbf	(Item 1.19)
}			
If (TILING_FLAG) {			
NUM_VERT_TILES_MINUS1	12	uimsbf	(Item 1.20)
NUM_HORIZ_TILES_MINUS1	12	uimsbf	(Item 1.21)
}			
for (n = 0; n < NUM_VERT_TILES_MINUS1; n++) {			
if (SHORT_HEADER_FLAG)			
WIDTH_IN_MB_OF_TILEI[n]	8	uimsbf	(Item 1.22)
Else			
WIDTH_IN_MB_OF_TILEI[n]	16	uimsbf	(Item 1.22)
}			
for (n = 0; n < NUM_HORIZ_TILES_MINUS1; n++) {			
if (SHORT_HEADER_FLAG)			
HEIGHT_IN_MB_OF_TILEI[n]	8	uimsbf	(Item 1.23)

Else			
HEIGHT_IN_MB_OF_TILEI[n]	16	uimsbf	(Item 1.23)
}			
if (TILE_STRETCH_FLAG) {			
for (n = 0; n < NumSpatialTiles; n++)			
TILE_STRETCH[n]	8	Uimsbf	(Item 1.24)
}			
if (WINDOWING_FLAG) {			
NUM_TOP_EXTRAPIXELS	6	Uimsbf	(Item 1.25)
NUM_LEFT_EXTRAPIXELS	6	Uimsbf	(Item 1.26)
NUM_BOTTOM_EXTRAPIXELS	6	Uimsbf	(Item 1.27)
NUM_RIGHT_EXTRAPIXELS	6	Uimsbf	(Item 1.28)
}			

Table 2. The Image Header structure (Taken from [HDPhotoBitstreamSpec 2006])

Originally documented in the HD Photo Bitstream Specification, the glossary below lists each syntax element contained within the Image Header structure defined in Table 2, along with an explanation of each syntax element [HDPhotoBitstreamSpec 2006]. Thus, the definitions for each of these syntax elements are exclusively from the HD Photo Bitstream Specification, and they are reproduced here for the sake of clarity.

- **(Item 1.1) GDI Signature** (labeled as **GDISIGNATURE**): This is an 8-byte "syntax element that identifies the [HD Photo] bitstream [and] shall have the [hexadecimal] value 0x574D50484F544F00" which corresponds to the ASCII value 'WMPHOTO' [HDPhotoBitstreamSpec 2006].

- **(Item 1.2) Codec Version** (labeled as **VERSION**): This "is a 4-bit syntax element that specifies the version of the bitstream" and "shall have the value 1. All other values are Reserved" [HDPhotoBitstreamSpec 2006].

- **(Item 1.3) Codec Sub-Version** (labeled as **SUBVERSION**): This "is a 4-bit syntax element that specifies the sub-version of the bitstream. This value shall be ignored by the decoder" [HDPhotoBitstreamSpec 2006].

- **(Item 1.4) Tiling Flag** (labeled as **TILING_FLAG**): This "is a Boolean syntax element" [HDPhotoBitstreamSpec 2006]. If set to **true**, then the Tiling Flag "specifies that tiles are present in the bitstream" [HDPhotoBitstreamSpec 2006]. If set to **false**, then the Tiling Flag specifies that the image is considered untiled and will be represented by a single tile [HDPhotoBitstreamSpec 2006].

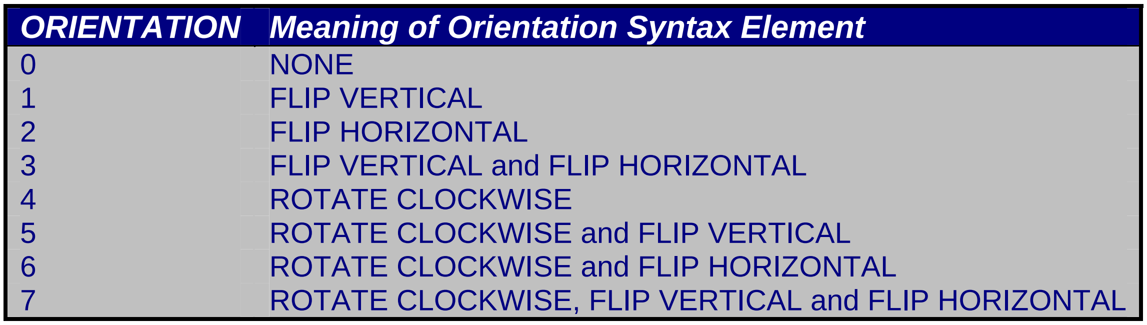
(For the remainder of this document, all Boolean syntax elements consist of 1 bit.)

- **(Item 1.5) Bitstream Format** (labeled as **BITSTREAM_FORMAT**): This "is a 1-bit syntax element" that represents the Mode of the HD Photo Bitstream, and this element can "take the value [of either] 0 or 1" [HDPhotoBitstreamSpec 2006]. If this element has a value of zero, then this "bitstream will be laid out in the Spatial Mode" [HDPhotoBitstreamSpec 2006]. If this element has a value of one, then this "bitstream will be laid out in the Frequency Mode" [HDPhotoBitstreamSpec 2006].

- **(Item 1.6) Orientation** (labeled as **ORIENTATION**): This "is a 3-bit syntax element [that] specifies the orientation of the image as defined in" Table 2 listed previously [HDPhotoBitstreamSpec 2006].

  The HD Photo Bitstream Specification notes that "the orientation syntax element" found originally in the HD Photo Container must "override the **ORIENTATION** syntax element" found in the HD Photo Bitstream, and the HD Photo decoder may therefore ignore this syntax element [HDPhotoBitstreamSpec 2006]. The purpose of the **ORIENTATION** syntax element "is to provide information regarding the desired orientation of the image, and it is desired for this field to match the orientation" syntax element found in the HD Photo Container [HDPhotoBitstreamSpec 2006].

  Listed below are the respective meanings of each value for the **ORIENTATION** element.



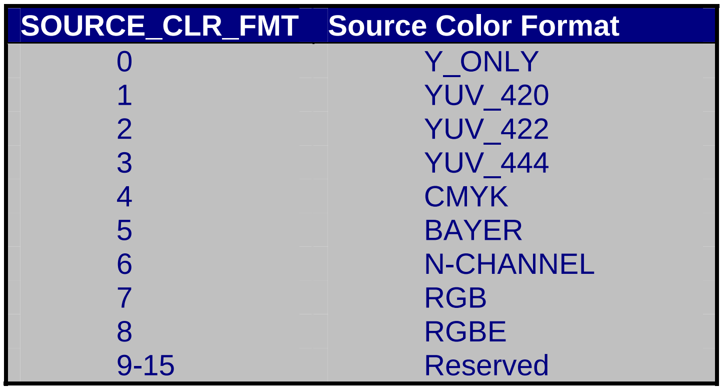
ORIENTATION	*Meaning of Orientation Syntax Element*
0	NONE
1	FLIP VERTICAL
2	FLIP HORIZONTAL
3	FLIP VERTICAL and FLIP HORIZONTAL
4	ROTATE CLOCKWISE
5	ROTATE CLOCKWISE and FLIP VERTICAL
6	ROTATE CLOCKWISE and FLIP HORIZONTAL
7	ROTATE CLOCKWISE, FLIP VERTICAL and FLIP HORIZONTAL

Table 3. ORIENTATION element values and the meaning of each respective value, taken from [HDPhotoBitstreamSpec 2006]

- **(Item 1.7) Index Table Present Flag** (labeled as **INDEXTABLE_PRESENT_FLAG)**: This "is a Boolean syntax element that specifies [whether the] index table is present in the bitstream. If INDEXTABLE_PRESENT_FLAG [is set to] true, the index table is present in the bitstream" [HDPhotoBitstreamSpec 2006]. If the bitstream is in the Frequency Mode and either the number of vertical tiles or the number of horizontal tiles is greater than zero, then this element "shall be set to true and the index table shall be present in the bitstream" [HDPhotoBitstreamSpec 2006]. Otherwise, this element being set to false signifies that "the index table shall not be present in the bitstream" [HDPhotoBitstreamSpec 2006].
- **(Item 1.8) Overlap** (labeled as **OVERLAP**): This "is a 2-bit syntax element that specifies whether overlap is present" [HDPhotoBitstreamSpec 2006]. The value assigned to this element indicates what types of filtering are performed on the image. A value of zero means "no post filtering is performed" [HDPhotoBitstreamSpec 2006]. A value of one means "only the outer post filtering shall be performed" [HDPhotoBitstreamSpec 2006]. A value of two means "both inner and outer post filtering shall be performed" [HDPhotoBitstreamSpec 2006]. For the current version of HD Photo, usage of the value of three is Reserved [HDPhotoBitstreamSpec 2006].
- **(Item 1.9) Short Header Flag** (labeled as **SHORT_HEADER_FLAG**): This "is a Boolean syntax element that specifies the bitsizes of the syntax elements that represent [the] height and width of the image and the tiles" [HDPhotoBitstreamSpec 2006]. If it is set to true, "smaller bit sizes are used" [HDPhotoBitstreamSpec 2006]. Otherwise, "longer bit sizes are used" [HDPhotoBitstreamSpec 2006].
- **(Item 1.10) Long Word Flag** (labeled as **LONG_WORD_FLAG**): This "is a Boolean syntax element and specifies whether 16 bit integers may be used for transform computations" [HDPhotoBitstreamSpec 2006]. If this element is set to false, "16 bit integers" are used; if it is set to true, "32 bit integers are used" [HDPhotoBitstreamSpec 2006].
- **(Item 1.11) Windowing Flag** (labeled as **WINDOWING_FLAG**): This "is a Boolean syntax element that specifies whether windowing is present in the bitstream"

[HDPhotoBitstreamSpec 2006]. "Windowing may be present in the bitstream" if this element is set to true; if this element is set to false, "windowing shall not be present in the bitstream" [HDPhotoBitstreamSpec 2006].

- **(Item 1.12) Trim FlexBits Flag** (labeled as **TRIM_FLEXBITS_FLAG**): This "is a Boolean syntax element that specifies if Flexbits are retained in full precision", assuming Flexbits are "present in the bitstream" [HDPhotoBitstreamSpec 2006].
- **(Item 1.13) Tile Stretch Flag** (labeled as **TILE_STRETCH_FLAG**): This "is a Boolean syntax element that specifies if tile stretching parameters are present in the bitstream" [HDPhotoBitstreamSpec 2006]. For version 1.0 of HD Photo, the value of this element is set to 0 [HDPhotoBitstreamSpec 2006]. At present, the value 1 is reserved [HDPhotoBitstreamSpec 2006].
- **(Item 1.14) Reserved Flag** (labeled as **RESERVED_FLAG**): This "is a 2-bit syntax element" that shall have a default value of zero [HDPhotoBitstreamSpec 2006]. For this syntax element, all "other values are Reserved" [HDPhotoBitstreamSpec 2006].
- **(Item 1.15) Alpha Channel Flag** (labeled as **ALPHACHANNEL_FLAG**): This "is a Boolean syntax element that specifies [if an] interleaved alpha channel is present in the bitstream" [HDPhotoBitstreamSpec 2006]. If this element is set to true, the "alpha channel is present and interleaved" [HDPhotoBitstreamSpec 2006]. If this element is set to false, either the "alpha channel shall be absent in the bitstream, or shall be carried as a separate image within the [HD Photo] container" [HDPhotoBitstreamSpec 2006].
- **(Item 1.16) Source Color Format** (labeled as **SOURCE_CLR_FMT**): This "is a 4-bit syntax element and [it] specifies the color format of the source image" [HDPhotoBitstreamSpec 2006]. The table listed below (taken from the HD Photo Bitstream Specification) shows the possible values for this element and the corresponding source color format for each value. For example, SOURCE_CLR_FMT having a value of 7 indicates that the color format of the source image is 'RGB' (Red-Green-Blue color format).



SOURCE_CLR_FMT	Source Color Format
0	Y_ONLY
1	YUV_420
2	YUV_422
3	YUV_444
4	CMYK
5	BAYER
6	N-CHANNEL
7	RGB
8	RGBE
9-15	Reserved

Table 4. SOURCE_CLR_FMT element values and the meaning of each respective value, taken from [HDPhotoBitstreamSpec 2006].

- **(Item 1.17) Source Bit Depth** (labeled as **SOURCE_BITDEPTH**): This “is a 4-bit syntax element [that] specifies the bitdepth of the source image” [HDPhotoBitstreamSpec 2006]. The table below contains the respective meanings of each value for the **SOURCE_BITDEPTH** element. The source bit depths “BD_1, BD_8, BD_16, BD_32, BD_5, and BD_10 are unsigned integer[s], corresponding to 1, 8, 16, 32, 5, and 10 bits per channel respectively” [HDPhotoBitstreamSpec 2006]. The BD_16S and BD_32S source bit depths “are signed integer[s] corresponding to 16 and 32 bits per channel respectively” [HDPhotoBitstreamSpec 2006]. BD_16F is a 16-bit Half bit depth, and BD_32F is a 32-bit Float bit depth [HDPhotoBitstreamSpec 2006]. BD_565 refers to a RGB 5:6:5 bit depth [HDPhotoBitstreamSpec 2006]

SOURCE_BITDEPTH	*Source Bit Depth*
0	BD_1, white foreground
1	BD_8
2	BD_16
3	BD_16S
4	BD_16F
5	BD_32
6	BD_32S
7	BD_32F
8	BD_5
9	BD_10
10	BD_565
11-14	Reserved
15	BD_1, black foreground

Table 5. SOURCE_BITDEPTH element values and the corresponding bit depth for each possible value, taken from [HDPhotoBitstreamSpec 2006].

- **(Item 1.18) Width** (labeled as **WIDTH_MINUS1**): This "is a syntax element" that is either 16 bits in length (if the **Short Header** flag is true) or 32 bits in length (if the **Short Header** flag is false), and it "specifies the width of the coded area" of the image, "minus 1" [HDPhotoBitstreamSpec 2006]. Thus, the formula to compute the width of the coded area is WIDTH_MINUS1 plus one [HDPhotoBitstreamSpec 2006].

- **(Item 1.19) Height** (labeled as **HEIGHT_MINUS1**): This "is a syntax element" that is either 16 bits in length (if the **Short Header** flag is true) or 32 bits in length (if the **Short Header** flag is false), and it "specifies the height of the coded area" of the image, "minus 1" [HDPhotoBitstreamSpec 2006]. Thus, the formula to compute the height of the coded area is HEIGHT_MINUS1 plus one [HDPhotoBitstreamSpec 2006].

- **(Item 1.20) Number of Vertical Tiles** (labeled as **NUM_VERT_TILES_MINUS1**): This "is a 12-bit syntax element that shall be present" in the HD Photo Bitstream "only if [the **Tiling Flag** is set to] true, and specifies the number of tiles in a [column], minus one" [HDPhotoBitstreamSpec 2006]. If this element "is not present", it defaults to a value of zero [HDPhotoBitstreamSpec 2006].

- **(Item 1.21) Number of Horizontal Tiles** (labeled as **NUM_HORIZ_TILES_MINUS1**): This "is a 12-bit syntax element that shall be present" in the HD Photo Bitstream "only if [the **Tiling Flag** is set to] true, and specifies the number of tiles in a row, minus one" [HDPhotoBitstreamSpec 2006]. If this element "is not present", it defaults to a value of zero. [HDPhotoBitstreamSpec 2006].

  Note: From the HD Photo Bitstream Specification, this syntax element and the previous syntax element (the **Number of Vertical Tiles** and the **Number of Horizontal Tiles**) are used to define "the number of image tiles" using the following formula [HDPhotoBitstreamSpec 2006]:

  NumSpatialTiles = (NUM_VERT_TILES_MINUS1 + 1) * (NUM_HORIZ_TILES_MINUS1 + 1)

- **(Item 1.22) Width in MB of Tile n** (labeled as **WIDTH_IN_MB_OF_TILEI**[n]): This "is a syntax element" that is present in the HD Photo Bitstream only if the **Number of Vertical Tiles** is greater than zero, and it "specifies the width (in macroblock units) of the nth tile along the horizontal direction, minus 1" [HDPhotoBitstreamSpec 2006]. If the Short Header flag is set to true, this syntax element is 8 bits in length; otherwise it is 16 bits in length [HDPhotoBitstreamSpec 2006].

- **(Item 1.23) Height in MB of Tile n** (labeled as **HEIGHT_IN_MB_OF_TILEI**[n]): This "is a syntax element" that is present in the HD Photo Bitstream only if the **Number of Horizontal Tiles** is greater than zero, and it "specifies the height (in macroblock units) of the nth tile along the vertical direction, minus 1" [HDPhotoBitstreamSpec 2006]. If the Short Header flag is set to true, this syntax element is 8 bits in length; otherwise it is 16 bits in length [HDPhotoBitstreamSpec 2006].

- **(Item 1.24) Tile Stretch of Tile n** (labeled as **TILE_STRETCH**[n]): This "is a 8-bit syntax element whose value should be ignored by the [HD Photo version 1.0] decoder" [HDPhotoBitstreamSpec 2006].

- **(Item 1.25) Image Offset from Top** (labeled as **NUM_TOP_EXTRAPIXELS**): This "is a 6-bit syntax element that shall be present" in the HD Photo Bitstream only if the **Windowing Flag** is set to true [HDPhotoBitstreamSpec 2006]. If this syntax element is present, **NUM_TOP_EXTRAPIXELS** "shall specify the vertical offset of the top of the image from the coded area" [HDPhotoBitstreamSpec 2006]. If this syntax element is not present, its default value will be zero [HDPhotoBitstreamSpec 2006].
- **(Item 1.26) Image Offset from Left** (labeled as **NUM_LEFT_EXTRAPIXELS**): This "is a 6-bit syntax element that shall be present" in the HD Photo Bitstream only if the **Windowing Flag** is set to true [HDPhotoBitstreamSpec 2006]. If this syntax element is present, **NUM_LEFT_EXTRAPIXELS** "shall specify the horizontal offset of the left boundary of the image from the coded area" [HDPhotoBitstreamSpec 2006]. If this syntax element is not present, its default value will be zero [HDPhotoBitstreamSpec 2006].
- **(Item 1.27) Image Offset from Bottom** (labeled as **NUM_BOTTOM_EXTRAPIXELS**): This "is a 6-bit syntax element that shall be present" in the HD Photo Bitstream only if the **Windowing Flag** is set to true [HDPhotoBitstreamSpec 2006]. If this syntax element is present, **NUM_BOTTOM_EXTRAPIXELS** "shall specify the vertical offset of the bottom of the image from the coded area" [HDPhotoBitstreamSpec 2006]. If this syntax element is not present, its default value will be zero [HDPhotoBitstreamSpec 2006].
- **(Item 1.28) Image Offset from Right** (labeled as **NUM_RIGHT_EXTRAPIXELS**): This "is a 6-bit syntax element that shall be present" in the HD Photo Bitstream only if the **Windowing Flag** is set to true [HDPhotoBitstreamSpec 2006]. If this syntax element is present, **NUM_RIGHT_EXTRAPIXELS** "shall specify the horizontal offset of the right boundary of the image from the coded area" [HDPhotoBitstreamSpec 2006]. If this syntax element is not present, its default value will be zero [HDPhotoBitstreamSpec 2006].

2. Image Plane Header

The next component that follows the Image Header in the Image Layer is the **Image Plane Header** structure (labeled below as IMAGE_PLANE_HEADER). The Image Plane Header structure can always be found immediately after the Image Header structure in a valid HD Photo Bitstream. The composition of the Image Plane Header is defined exclusively in [HDPhotoBitstreamSpec 2006], and it is reproduced for the sake of convenience in Table 6 below.

IMAGE_PLANE_HEADER (){	***Num bits***	***Descriptor***	***Reference***
CLR_FMT	3	uimsbf	(Item 2.1)
NO_SCALED_FLAG	1	bool	(Item 2.2)
BANDS_PRESENT	4	uimsbf	(Item 2.3)
if (CLR_FMT == 'YUV_444') {			
CHROMA_CENTERING	4	uimsbf	(Item 2.4)
COLOR_INTERPRETATION	4	uimsbf	(Item 2.5)
} else if (CLR_FMT == 'N_CHANNEL') {			
NUM_CHANNELS_MINUS1	4	uimsbf	(Item 2.6)
COLOR_INTERPRETATION	4	uimsbf	(Item 2.5)
}			
if (SOURCE_CLR_FMT == 'BAYER') {			
BAYER_PATTERN	2	uimsbf	(Item 2.7)
CHROMA_CENTERING_BAYER	2	uimsbf	(Item 2.8)
COLOR_INTERPRETATION	4	uimsbf	(Item 2.5)
}			
if (SOURCE_BITDEPTH ∊ {BD_16,BD_16S,BD_32,BD_32S}) {			
SHIFT_BITS	8	uimsbf	(Item 2.9)
}			
if (SOURCE_BITDEPTH == 'BD_32F') {			
MANTISSA	8	uimsbf	(Item 2.10)
EXPBIAS	8	uimsbf	(Item 2.11)
}			
DC_FRAME_UNIFORM	1	bool	(Item 2.12)
if (DC_FRAME_UNIFORM) {			
DC_QUANTIZER()	variable	struct	
}			
if (BANDS_PRESENT != 'SB_DC_ONLY') {			
USE_DC_QUANTIZER	1	bool	(Item 2.13)
if (USE_DC_QUANTIZER == false) {			
LP_FRAME_UNIFORM	1	bool	(Item 2.14)
if (LP_FRAME_UNIFORM) {			
NUM_LP_QUANTIZERS = 1			(Item 2.18)
LP_QUANTIZER()	variable	struct	
}			
}			
if (BANDS_PRESENT != 'SB_NO_HIGHPASS') {			
USE_LP_QUANTIZER	1	bool	(Item 2.15)

if (USE_LP_QUANTIZER == false) {			
HP_FRAME_UNIFORM	1	bool	(Item 2.16)
if (HP_FRAME_UNIFORM) {			
NUM_HP_QUANTIZERS = 1			(Item 2.19)
HP_QUANTIZER()	variable	struct	
}			
}			
}			
}			
FLUSHBYTE	variable		
}			

Table 6. The syntax table for the Image Plane Header structure (Taken from [HDPhotoBitstreamSpec 2006]).

Originally documented in the HD Photo Bitstream Specification and reproduced here for the sake of clarity, a brief definition for each syntax element found within the Image Plane Header structure defined in Table 6 is included in the following:

- **(Item 2.1) Color Format** (labeled as **CLR_FMT**): This "is a 3-bit syntax element that shall specify the internal color format of the coded image" (i.e., the color format of the HD Photo image created from the source image, not the color format of the source image itself) [HDPhotoBitstreamSpec 2006]. The table below shows the possible values for Color Format and the corresponding color format for each value.

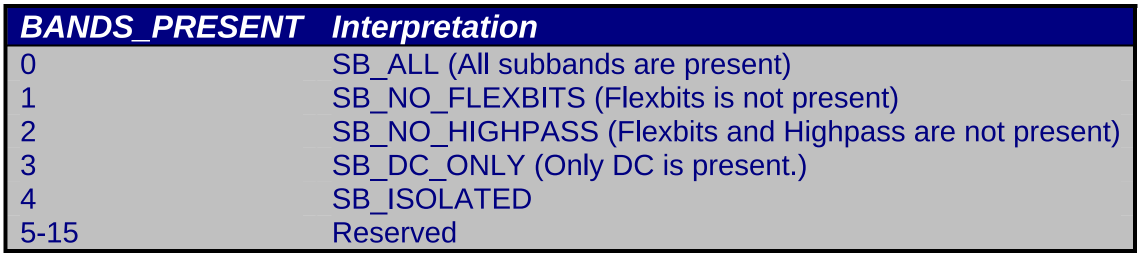
CLR_FMT	***Color Format***
0	Y_ONLY
1	YUV_420
2	YUV_422
3	YUV_444
4	CMYK
5	BAYER
6	N-CHANNEL
7	Reserved

Table 7. The values for Color Format (CLR_FMT), along with its corresponding Color Format. Taken from [HDPhotoBitstreamSpec 2006].

- **(Item 2.2) No Scaled Arithmetic Flag** (labeled as **NO_SCALED_FLAG**): This "is a Boolean syntax element that" indicates if scaling is used with respect to the transform process [HDPhotoBitstreamSpec 2006]. If this element is set to true, scaling is not used; if it is set to false, scaling is used [HDPhotoBitstreamSpec 2006].

- **(Item 2.3) Bands Present** (labeled as **BANDS_PRESENT**): This "is a 4-bit syntax element that" specifies which of the various "frequency bands are present in the bitstream" [HDPhotoBitstreamSpec 2006]. The table below shows how the value of this element indicates the presence of these bands. (The HD Photo Bitstream Specification notes that the SB_ISOLATED interpretation is an indication that "external information" is needed to decode the bitstream [HDPhotoBitstreamSpec 2006].)

-



BANDS_PRESENT	*Interpretation*
0	SB_ALL (All subbands are present)
1	SB_NO_FLEXBITS (Flexbits is not present)
2	SB_NO_HIGHPASS (Flexbits and Highpass are not present)
3	SB_DC_ONLY (Only DC is present.)
4	SB_ISOLATED
5-15	Reserved

Table 8. The values of Band Present (BANDS_PRESENT), and the interpretation for each value. Taken from [HDPhotoBitstreamSpec 2006].

- **(Item 2.4) Chroma Centering** (labeled as **CHROMA_CENTERING**): This "is a 4-bit syntax element" that is present in the HD Photo Bitstream only if the value of the Color Format element (CLR_FMT) is 'YUV_444' [HDPhotoBitstreamSpec 2006]. If this element "is not present", it has a default value of zero [HDPhotoBitstreamSpec 2006]. For version 1.0 of HD Photo, the decoder ignores this value [HDPhotoBitstreamSpec 2006].

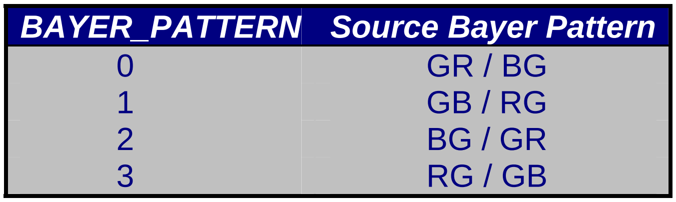
- **(Item 2.5) Color Interpretation** (labeled as **COLOR_INTERPRETATION**): This "is a 4-bit syntax element that" is present in the HD Photo Bitstream only if the value of the Color Format element (CLR_FMT) is 'YUV_444' or 'N_CHANNEL' or if the value of the Source Color Format element (SOURCE_CLR_FMT) is 'BAYER' [HDPhotoBitstreamSpec 2006]. If this element "is not present", it has a default value of zero [HDPhotoBitstreamSpec 2006]. For version 1.0 of HD Photo, the decoder ignores this value [HDPhotoBitstreamSpec 2006].

- **(Item 2.6) Number of Channels** (labeled as **NUM_CHANNELS_MINUS1**): This "is a 4-bit syntax element that" is present in the HD Photo Bitstream only if the Color

Format (CLR_FMT) syntax element is equal to ‘N_CHANNEL’ [HDPhotoBitstreamSpec 2006]. In this case, the number of channels (labeled as **Nchannels**) is NUM_CHANNELS_MINUS1 plus one [HDPhotoBitstreamSpec 2006]. For example, if the bitstream assigns the value of 3 to NUM_CHANNELS_MINUS1, then Nchannels is calculated to be 3 plus 1, for a total of 4 channels.

  - For all other color formats, a preset value is assigned to the number of channels. For instance, if the Color Format is ‘YUV_420’, ‘YUV_422’, or ‘YUV_444’, then the number of channels is set to three (or **Nchannels = 3**) [HDPhotoBitstreamSpec 2006].

- **(Item 2.7) Bayer Pattern** (labeled as **BAYER_PATTERN**): This “is a 2-bit syntax element that” is present in the HD Photo Bitstream only if the Source Color Format (SOURCE_CLR_FMT) is equal to ‘BAYER’, and it specifies “the source Bayer pattern” as shown in the table below [HDPhotoBitstreamSpec 2006]. For version 1.0 of HD Photo, the decoder ignores this value [HDPhotoBitstreamSpec 2006].



BAYER_PATTERN	*Source Bayer Pattern*
0	GR / BG
1	GB / RG
2	BG / GR
3	RG / GB

Table 9. The values of the Bayer Pattern (BAYER_PATTERN), and the interpretation for each value. Taken from [HDPhotoBitstreamSpec 2006].

- **(Item 2.8) Chroma Centering Bayer** (labeled as **CHROMA_CENTERING_BAYER**): This “is a 2-bit syntax element” that “is present” in the HD Photo Bitstream if the Source Color Format (SOURCE_CLR_FMT) is equal to ‘BAYER’ [HDPhotoBitstreamSpec 2006]. For version 1.0 of HD Photo, the decoder ignores this value [HDPhotoBitstreamSpec 2006].

- **(Item 2.9) Pre/Post Shift Bits** (labeled as **SHIFT_BITS**): This “is a 8-bit syntax element that” is present in the HD Photo Bitstream if the Source Bit Depth

(SOURCE_BIT_DEPTH) is equal to one of the following values: ‘BD_16’, ‘BD_16S’, ‘BD_32’, or ‘BD_32S’ [HDPhotoBitstreamSpec 2006]. If present, this element indicates “the number of bits by which to left-shift the reconstructed data” [HDPhotoBitstreamSpec 2006].

- **(Item 2.10) Length of Mantissa** (labeled as **MANTISSA**): This “is a 8-bit syntax element that” is present in the HD Photo Bitstream if the Source Bit Depth (SOURCE_BIT_DEPTH) is equal to ‘BD_32F’, and it indicates “the number of mantissa bits in the encoding of the floating point data” [HDPhotoBitstreamSpec 2006].

- **(Item 2.11) Exponent Bias** (labeled as **EXPBIAS**): This “is a 8-bit syntax element that” is present in the HD Photo Bitstream if the Source Bit Depth (SOURCE_BIT_DEPTH) is equal to ‘BD_32F’, and it indicates “the bias of the exponent in the encoding of the floating point data” [HDPhotoBitstreamSpec 2006].

- **(Item 2.12) DC Frame Uniform** (labeled as **DC_FRAME_UNIFORM**): This “is a Boolean syntax element” which indicates if “a single quantizer is used for the DC band” [HDPhotoBitstreamSpec 2006]. If this element is set to true, then “a single quantizer is used for the DC bands of each color plane in the image, and this quantizer is signaled in the Image Plane Header” [HDPhotoBitstreamSpec 2006]. If this element is set to false, then “multiple quantizers may be used for the DC bands of each color plane in the image, and these quantizers are signaled in the Tile Header” [HDPhotoBitstreamSpec 2006].

- **(Item 2.13) Use DC Quantizer** (labeled as **USE_DC_QUANTIZER**): This “is a Boolean syntax element” that indicates “whether the low pass band uses the same quantizer as the DC band” [HDPhotoBitstreamSpec 2006]. If this element is set to true, the “value of the low pass quantizer” is set to the value of “the DC band quantizer” [HDPhotoBitstreamSpec 2006]. If it is set to false, “the value of the low pass quantizer shall be explicitly signaled in the bitstream” [HDPhotoBitstreamSpec 2006].

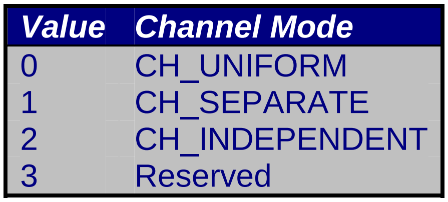
- **(Item 2.14) Low Pass Frame Uniform Flag** (labeled as **LP_FRAME_UNIFORM**): This "is a Boolean syntax element" which indicates "whether a single quantizer is used for the low pass band" [HDPhotoBitstreamSpec 2006]. If the **Low Pass Frame Uniform Flag** is set to true, then "a single quantizer shall be used for all the low pass bands of each color plane in the image, and this quantizer is signaled in the Image Plane Header" [HDPhotoBitstreamSpec 2006]. If it is set to false, then "multiple quantizers may be used for the low pass bands of each color plane in the image, and these quantizers shall be signaled in the Tile Header" [HDPhotoBitstreamSpec 2006].

- **(Item 2.15) Use Low Pass Quantizer** (labeled as **USE_LP_QUANTIZER**): This "is a Boolean syntax element that signals whether the high pass band uses the same quantizer as the low pass band" [HDPhotoBitstreamSpec 2006]. If the **Use Low Pass Quantizer** to true, "the value of the high pass quantizer shall be set [to same value as its] corresponding low pass band quantizer" [HDPhotoBitstreamSpec 2006]. If it is set to false, "the value of the high pass quantizer shall be explicitly signaled in the bitstream" [HDPhotoBitstreamSpec 2006].

- **(Item 2.16) High Pass Frame Uniform Flag** (labeled as **HP_FRAME_UNIFORM**): This "is a Boolean syntax element that signals whether a single quantizer is used for the high pass band" [HDPhotoBitstreamSpec 2006]. If the **High Pass Frame Uniform Flag** is set to true, then "a single quantizer shall be used for all the high pass bands of each color plane in the image, and this quantizer is signaled in the Image Plane Header" [HDPhotoBitstreamSpec 2006]. If it is set to false, then "multiple quantizers may be used for the high pass bands of each color plane in the image, and these quantizers shall be signaled in the Tile Header" [HDPhotoBitstreamSpec 2006].

- **(Item 2.17) Channel Mode** (labeled as **CH_MODE**): This "is a 2-bit syntax element that is present" in the HD Photo Bitstream if the number of channels (**Nchannels**) used in the image is greater than one, and "this element signals whether the color planes share a quantizer" [HDPhotoBitstreamSpec 2006]. The table listed in [HDPhotoBitstream 2006] defines the meaning of this syntax element.

    - Note: If it is present in the HD Photo Bitstream, the Channel Mode syntax element is read in during the processing of the Quantizer structures (listed in Table 3 as DC_QUANTIZER, LP_QUANTIZER, and HP_QUANTIZER) in the bitstream. This is why the CH_MODE element is not found in the syntax table for the Image Plane Header. It is found in the syntax tables provided for the Quantizer structures in the HD Photo Bitstream Specification document; these structures are not covered in detail for this thesis, and an explanation for this decision will be provided later on in this chapter.



Value	*Channel Mode*
0	CH_UNIFORM
1	CH_SEPARATE
2	CH_INDEPENDENT
3	Reserved

Table 10. The values of the Channel Mode (CH_MODE), and the interpretation for each value. Taken from [HDPhotoBitstreamSpec 2006]

- **(Item 2.18) Number of LP Quantizers** (labeled as **NUM_LP_QUANTIZERS**): This "is a 4-bit syntax element that is present if" the **Low Pass Frame Uniform Flag** is set to false, and it indicates "the number of low pass band quantizers per color plane in the [present] tile" [HDPhotoBitstreamSpec 2006]. If the Low Pass Uniform Flag is set to true, this syntax element is set to 1 by default [HDPhotoBitstreamSpec 2006].

- **(Item 2.19) Number of HP Quantizers** (labeled as **NUM_HP_QUANTIZERS**): This "is a 4-bit syntax element that is present if" the **High Pass Frame Uniform Flag** is set to false, and it indicates "the number of high pass band quantizers per color plane in the [present] tile" [HDPhotoBitstreamSpec 2006]. If the High Pass Uniform Flag is set to true, this syntax element is set to 1 by default [HDPhotoBitstreamSpec 2006].

    - **Note:** If the syntax elements NUM_LP_QUANTIZERS and NUM_HP_QUANTIZERS are present in the bitstream, they are read in as 4-bit syntax elements from the bitstream during the processing of the Tile Layer, not during the processing of the Image Plane Header structure. If they are not present, these elements are set to default values of 1 during the processing of the Image Plane Header structure.

- **(Item 2.20) Flush to Byte Operator** (labeled as **FLUSHBYTE**): This function reads in anywhere from one to seven data bits, depending on the number needed for byte-alignment purposes, and then simply throws away the data. The sole purpose of the Flush Byte function is that after this function executes, the HD Photo decoder (such as the image viewer application handling the HD Photo image) will be currently looking at the beginning of a new byte in the HD Photo Bitstream. For example, if the HD Photo decoder had read in three bits of the current data byte in the HD Photo Bitstream, the FLUSHBYTE function would have the HD Photo read in five more data bits from the bitstream. (Remember, there are 8 bits in a byte.) The result of this operation is the HD Photo decoder would now be byte-aligned with the HD Photo bitstream (i.e., the HD Photo decoder now looks at the start of a new byte in the bitstream).

  This description of the Flush To Byte Operator falls in line with the documentation in the HD Photo Bitstream Specification:

  > "FLUSHBYTE is a variable length syntax element which accounts for the flush-to-byte operation. The length of FLUSHBYTE is therefore between 0 and 7 bits, and upon reading FLUSHBYTE the bitstream 'pointer' is byte aligned. The value of FLUSHBYTE is discarded and therefore does not have any interpretation." [HDPhotoBitstreamSpec 2006]

The Quantizer structures (listed in Table 6 as DC_QUANTIZER, LP_QUANTIZER, and HP_QUANTIZER) will not be covered in detail for this thesis because they do not factor into the experimental scheme used in fuzzing HD Photo files. These structures are used for storing the values of the DC, low-pass, and high-pass quantizers used for the various color planes in HD Photo, so they are important only with respect to the visual rendering of this image file format.

3. Index Table

Next after the **Image Plane Header** in the **Image Layer** is the **Index Table** (labeled as INDEX_TABLE in Table 11 displayed on the next page). The tile offsets that will be stored in the **Index Table** are present in the bitstream only if the Index Table Present Flag (INDEXTABLE_PRESENT_FLAG) is set to true. Otherwise, there is only one tile that represents the image, and its offset will have a default value of zero. The VLWESC function used to read in the INDEX_OFFSET_TILE[n] and the SKIP_BYTES syntax elements will be described later in this thesis.

INDEX_TABLE (){	*Num bits*	*Descriptor*	*Reference*
if (INDEXTABLE_PRESENT_FLAG) {			
INDEXTABLE_STARTCODE	16	uimsbf	(Item 3.1)
for (n = 0; n < NumberIndexTableEntries; n++) {			
INDEX_OFFSET_TILE[n]	VLWESC		(Item 3.2)
}			
}			
SKIP_BYTES	VLWESC		(Item 3.3)
PADDING_DATA	SKIP_BYTES * 8		(Item 3.4)
}			

Table 11. The syntax table for the Index Table (INDEX_TABLE) structure in the Image Layer. Taken from [HDPhotoBitstreamSpec 2006].

Originally documented in the HD Photo Bitstream Specification, the glossary listing containing the meaning of each syntax element in the Index Table and pseudo-code which detail how the Index Table structure is processed is reproduced below.

- **(Item 3.1) Index Table Start Code** (labeled as **INDEXTABLE_STARTCODE**): This "16-bit length syntax element" signifies the beginning "of the Index Table. This element shall have the [hexadecimal] value **0x0001**. Other values of INDEXTABLE_STARTCODE are Reserved" [HDPhotoBitstreamSpec 2006].

- **(Item 3.2) Index Offset of Tile n** (labeled as **INDEX_OFFSET_TILE**[n]): This "is a syntax element which specifies the offset of the **n**th tile-frequency packet from the start of the coded image data" [HDPhotoBitstreamSpec 2006]. The Decode Variable Length Word operator (VLWESC) function shall be used to decode "the size and value of this syntax element" [HDPhotoBitstreamSpec 2006].

- **(Item 3.3) Number of Skipped Bytes** (labeled as **SKIP_BYTES**): This "is a syntax element which specifies the number of bytes till the start of the coded image data. The size and value of this syntax element is" determined by the VLWESC function [HDPhotoBitstreamSpec 2006].
- **(Item 3.4) Padding Data** (labeled as **PADDING_DATA**): This "is a stream of **SKIP_BYTES** number of bytes between the end of the SKIP_BYTES symbol and the start of the coded image data," hence the name [HDPhotoBitstreamSpec 2006]. The purpose of reading in **Padding Data** is to correctly get to the beginning of the coded image data (i.e., the start of the first Tile Layer), discarding the padded bytes along the way.

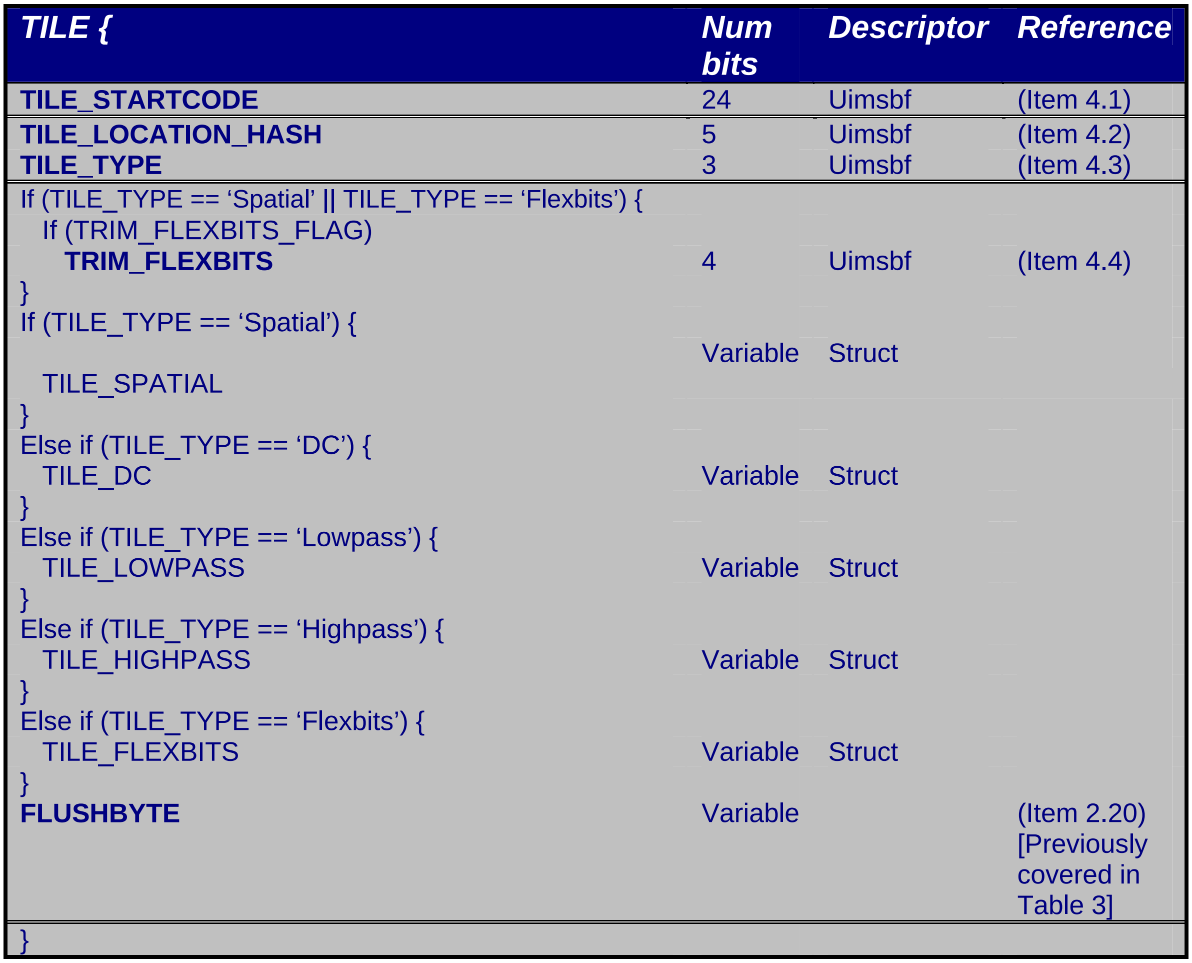
Table 12 below details the definition of the Decode Variable Length Word operator (**VLWESC**) function used in decoding the Index Table. The **VLWESC** function first reads in 8 data bits from the bitstream, referred to as the **FIRST_BYTE**. If the value of the first 8 data bits (**FIRST_BYTE**) is less than the hexadecimal value 0xFB, then the function reads in 8 more data bits from the bitstream (labeled below as **SECOND_BYTE**) and calculates the local variable **Value** to be the sum of **SECOND_BYTE** and **FIRST_BYTE** multiplied by 256. If **FIRST_BYTE** is equal to 0xFB, then the function reads in 32 data bits (stored as **FOUR_BYTES**) and sets **Value** to the hexadecimal value of **FOUR_BYTES**. If **FIRST_BYTE** is equal to 0xFC, then the function reads in 64 data bits (stored as **EIGHT_BYTES**) and sets **Value** to the hexadecimal value of **EIGHT_BYTES**. Otherwise, **Value** is set to zero. The last step of the **VLWESC** function is to return the value of **Value** to the pseudocode that called it.

<table>
<tr><th>VLWESC( ) {</th><th>Number of bits</th><th>Descriptor</th></tr>
<tr><td><b>FIRST_BYTE</b></td><td>8</td><td>Uimsbf</td></tr>
<tr><td>if (FIRST_BYTE < 0xfb) {</td><td></td><td></td></tr>
<tr><td><b>SECOND_BYTE</b></td><td>8</td><td>Uimsbf</td></tr>
<tr><td>Value = FIRST_BYTE * 256 + SECOND_BYTE</td><td></td><td></td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td>else if (FIRST_BYTE == 0x0fb) {</td><td></td><td></td></tr>
<tr><td><b>FOUR_BYTES</b></td><td>32</td><td>Uimsbf</td></tr>
<tr><td>Value = FOUR_BYTES</td><td></td><td></td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td>else if (FIRST_BYTE == 0x0fc) {</td><td></td><td></td></tr>
<tr><td><b>EIGHT_BYTES</b></td><td>64</td><td>Uimsbf</td></tr>
<tr><td>Value = EIGHT_BYTES</td><td></td><td></td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td>else { // FIRST_BYTE == 0xfd||0xfe||0xff</td><td></td><td></td></tr>
<tr><td>Value = 0 // Escape Mode</td><td></td><td></td></tr>
<tr><td>}</td><td></td><td></td></tr>
<tr><td><b>return Value</b></td><td></td><td></td></tr>
<tr><td>}</td><td></td><td></td></tr>
</table>

Table 12. The syntax table for the function VLWESC(), taken from [HDPhotoBitstreamSpec 2006].

4. Tile Layer

Located after the Image Layer, the next layer in the HD Photo Bitstream is the **Tile Layer**. Table 13 (shown on the next page) serves as a detailed specification of the **Tile Layer** in HD Photo. Since the Tile Layer is followed by one or more instances of both the Macroblock Layer and the Block Layer, this layer is the last one to be covered in detail with respect to the scope of this document.



TILE {	***Num bits***	***Descriptor***	***Reference***
TILE_STARTCODE	24	Uimsbf	(Item 4.1)
TILE_LOCATION_HASH	5	Uimsbf	(Item 4.2)
TILE_TYPE	3	Uimsbf	(Item 4.3)
If (TILE_TYPE == 'Spatial' || TILE_TYPE == 'Flexbits') {			
If (TRIM_FLEXBITS_FLAG)			
TRIM_FLEXBITS	4	Uimsbf	(Item 4.4)
}			
If (TILE_TYPE == 'Spatial') {			
	Variable	Struct	
TILE_SPATIAL			
}			
Else if (TILE_TYPE == 'DC') {			
TILE_DC	Variable	Struct	
}			
Else if (TILE_TYPE == 'Lowpass') {			
TILE_LOWPASS	Variable	Struct	
}			
Else if (TILE_TYPE == 'Highpass') {			
TILE_HIGHPASS	Variable	Struct	
}			
Else if (TILE_TYPE == 'Flexbits') {			
TILE_FLEXBITS	Variable	Struct	
}			
FLUSHBYTE	Variable		(Item 2.20) [Previously covered in Table 3]
}			

Table 13. The Tile Layer, taken from [HDPhotoBitstreamSpec 2006].

Originally documented in the HD Photo Bitstream Specification, the glossary listing containing the meaning of each syntax element in the Tile Layer relevant to the scope of this document is reproduced below.

- **(Item 4.1) Start Code of Tile** (labeled as **TILE_STARTCODE**): This "is a 24-bit syntax element that indicates the start of a tile. If [this element has a value of] 1, then the tile is considered to be a valid tile and decodable. If [this element does not have a value of 1] and [the Tile Type is] 'Flexbits', the value of [the syntax element] TRIM_FLEXBITS shall be set to 15, and all the data bits of this tile are set to zero" [HDPhotoBitstreamSpec 2006].

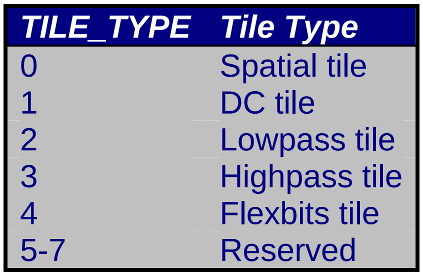
- **(Item 4.2) Hash of Tile Location** (labeled as **TILE_LOCATION_HASH**): This "is a 5-bit syntax element [whose value] shall be equal to [the following formula]":

  (VerticalTileIndex + NumberOfHorizontalTiles + HorizontalTileIndex) mod 32

  where the lone exception to this formula is the scenario where the Tile Type is equal to 'Flexbits' [HDPhotoBitstreamSpec 2006]. In this case, this syntax element (TILE_LOCATION_HASH) can have "any arbitrary value" [HDPhotoBitstreamSpec 2006].

- **(Item 4.3) Tile Type** (labeled as **TILE_TYPE**): This "is a 3-bit syntax element that shall describe the type of data contained in [the present] tile according to the [table listed below]" [HDPhotoBitstreamSpec 2006].

  For example, if TILE_TYPE has a value of zero, then it indicates that the present tile is a spatial tile.



TILE_TYPE	*Tile Type*
0	Spatial tile
1	DC tile
2	Lowpass tile
3	Highpass tile
4	Flexbits tile
5-7	Reserved

Table 14. The values of Tile Type (TILE_TYPE) and the meaning of each value. Taken from [HDPhotoBitstreamSpec 2006].

- **(Item 4.4) Trim Flexbits** (labeled as **TRIM_FLEXBITS**): This "4-bit syntax element" is present in the HD Photo Bitstream if the **Tile Type** (**TILE_TYPE**) for this current tile is either 'Spatial' or 'Flexbits', and if the Trim Flexbits Flag (TRIM_FLEXBITS_FLAG) is set to 'True' [HDPhotoBitstreamSpec 2006]. Otherwise, **Trim Flexbits** defaults to a value of zero [HDPhotoBitstreamSpec 2006].

  Although its interpretation isn't noted in the bitstream specification, this syntax element denotes the precision used to treat the **Flexbits** layer inside each instance of the **Tile** layer.

The TILE_SPATIAL, TILE_DC, TILE_LOWPASS, TILE_HIGHPASS, and TILE_FLEXBITS structures mainly consist of instances of the Macroblock Layer, instances of the Block Layer, and Quantizer structures mentioned previously in passing in earlier in this document. Since it was already mentioned that the scope of this thesis will not cover in detail the Macroblock Layer, the Block Layer, and the Quantizer structures, this document will not cover in detail the TILE_SPATIAL, TILE_DC, TILE_LOWPASS, TILE_HIGHPASS, and TILE_FLEXBITS structures as well.

Now that the HD Photo Bitstream has been covered sufficiently in detail, the concept of fuzzing can be explained, along with how it can be used to find potential security bugs in image viewer applications that can handle the HD Photo file format. The security testing technique that is called “fuzzing” is covered in the next chapter. Examples will be given on how fuzzing has been applied to testing file formats with respect to applications.

III. THE TECHNIQUE OF FILE FORMAT FUZZING

When it comes to software security testing with respect to vulnerabilities, a key aspect of this type of testing is a technique called "fuzzing." Still considered in its infancy as a formal field, fuzzing is gaining acceptance in the computer security area as a valid and efficient technique in testing software for security bugs. Fuzzing is not meant as a substitute to replace formal verification or testing procedures. Rather, it is meant as an additional method of testing to complement existing test procedures [Miller 1990].

A. ORIGINS AND DEFINITION OF FUZZING

The term "fuzzing" cannot be found in any classical dictionary, yet it is a testing technique that has been put to use in software development for many years, mostly performed by in-house testing departments [Seltzer 2006]. Peter Oehlert lends an excellent definition to the term of fuzzing:

"Fuzzing – n. a highly automatic testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. From modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines." [Oehlert 2005]

The term "fuzzing" was coined in 1990 when Miller et. al. from the University of Wisconsin-Madison wrote a paper on the technique [Miller 2005]. It was "on a dark and stormy night" when one of the authors of the said paper was connected to his workstation from home via a dial-up connection. The author knew that whenever a storm was in the area, random noise would be generated on the telephone lines as a result of the rain, so he attempted to type and transmit his commands across the dial-up connection before the noise could affect it in transit [Miller 2005]. The surprising part wasn't that the noise on the telephone lines effectively scrambled the commands that the author had remotely sent to his workstation from home. What was surprising was that because of the noise on the telephone lines, parts of the character sequences that represented his commands would turn into random characters that "actually caused programs to crash and hang" [Warnock 2007]. This observation gave the motivation for the technique known as fuzzing:

"This scenario motivated a systematic test of the utility programs running on various versions of the UNIX operating system. The project proceeded in four steps: (1) construct a program to generate random characters, plus a program to help test interactive utilities; (2) use these programs to test a large number of utilities on random input strings to see if they crash; (3) identify the strings (or types of strings) that crash these programs; and (4) identify the cause of the program crashes and categorize the common mistakes that cause these crashes." [Miller 2005]

Ever since the initial University of Madison-Wisconsin paper on fuzzing was published in 1990, fuzzing has steadily been used as a software auditing technique used by software developers as a complement to traditional testing and by third-party testers as a means of independently evaluating software safety and software security.

In academia, the closest type of testing that can be related to fuzzing is **boundary value analysis** [Sutton 2007]; boundary value analysis is a technique for test data selection where a tester would choose boundary values that lie along data extremes inside a range of known good values. Boundary values include a maximum value, a minimum value, values just inside the boundary, values just outside boundary, typical values, and error values. The key concept in boundary value analysis is that the tester is taking a "range of known good values for a particular input" and makes a series of "test values that straddle the boundary cases of known good and bad values" [Sutton 2007]. The expectation here is if the target system works correctly for the extreme values and the special values, then it should also work correctly for all the values in between. The purpose of this test is to see if the target system appropriately rejects unacceptable inputs (or the values outside the accepted range of inputs) while allowing the values located inside "the full range of acceptable inputs" [Sutton 2007]. In this sense, fuzzing can be likened to boundary value analysis, with the exception that fuzzing does not focus solely on boundary values, but also on any input values which can cause undefined or insecure behavior of the target system or software [Sutton 2007].

Fuzzing is generally categorized as a form of **black box testing**, since it is based on requirements and functionality of the target software. No knowledge of internal design or code of the target software is needed, hence the name. The fuzzer simply generates input for the target software to consume, then logs the results of any crashes that may occur. Then the fuzz testers would see what bug caused the particular crash, and what category that specific bug falls under. If the bug happens to be a security bug, the fuzz testers would determine whether the bug is exploitable or not.

The approach to fuzzing can differ from case to case, depending on the software selected for testing, the type of data used for testing, and the person (or people) conducting the testing. However, all types of fuzzing generally follow the same basic steps outlined by Sutton, Greene, and Amini:

> "1. **Identify target**. It isn't possible to select a fuzzing tool or technique until we have a target in mind. If you are fuzzing an internally developed application during a security audit, target selection will be taken care of for you. If, however, you are conducting vulnerability research to uncover vulnerabilities in third-party applications, you'll have some flexibility. When identifying a target, look at a vendor's past history with regard to previously discovered vulnerabilities. Check vulnerability aggregation sites such as SecurityFocus or Secunia. A vendor with a poor track record for past vulnerabilities is more likely to have poor coding practices that will ultimately lead to the discovery of further vulnerabilities. Beyond selecting an application, it might also be necessary to target a specific file or library within the application. If that's the case, you might want to look for binaries that are shared across multiple applications, as vulnerabilities in such targets will be of higher risk due to the expanded user base."

> "2. **Identify inputs**. Virtually all exploitable vulnerabilities are caused by applications accepting user input and processing that data without first sanitizing it or applying validation routines. Enumerating input vectors is pivotal to the success of fuzzing. Failing to locate potential sources of input or the expected input values can severely limit testing. Apply lateral thinking when looking for input vectors. Whereas some input vectors are obvious, others are subtler. In the end, anything sent from the client to the target should be considered an input

> vector. That includes headers, filenames, environment variables, registry keys, and so on. All should be considered input vectors and are therefore potential fuzz variables."
>
> "3. **Generate fuzzed data**. Once input vectors have been identified, fuzz data must be generated. The decision to use predetermined values, mutate existing data, or generate data dynamically will depend on the target and data format. Regardless of the approach selected, automation should be applied to this process."
>
> "4. **Execute fuzzed data**. This step goes hand in hand with the previous one and is where fuzzing becomes a verb. Execution could involve the act of sending a data packet to the target, opening a file, or launching a target process. Again, automation is crucial. Without it, we're not really fuzzing."
>
> "5. **Monitor for exceptions**. A vital but often overlook step during fuzzing is the execution or fault monitoring process. Transmitting 10,000 fuzz packets to a target Web server, for example, and ultimately causing that server to crash is a useless endeavor if we are unable to pinpoint the packet responsible for the crash. Monitoring can take many forms and will be dependent on the target application and type of fuzzing being used."
>
> "6. **Determine exploitability**. Once a fault is identified, depending on the goals of the audit, it might also be necessary to determine if the uncovered bug can be further exploited. This is typically a manual process that requires specialized security knowledge. It might therefore be a step performed by someone other than the person conducting the initial fuzzing." [Sutton 2007]

For the steps listed above, steps 3, 4, 5, are iterated over and over by an automated fuzzing tool. The fuzzing tool would perform step 3 (generate fuzzed data) by taking a set of supplied data that is valid, and use some kind of method to malform the data. It would complete step 4 (execute fuzzed data) by submitting the malformed data generated in step 3 to the target application. If the application deviates from its specified behavior, the resulting exception is logged and the malformed data that caused the crash is saved for auditing purposes. If the application behaves normally, the results are discarded. In either case, the fuzzing tool starts the process again from step 3. Figure 4 below illustrates the iteration process of an automated fuzzing tool, taken from [Oehlert 2005].

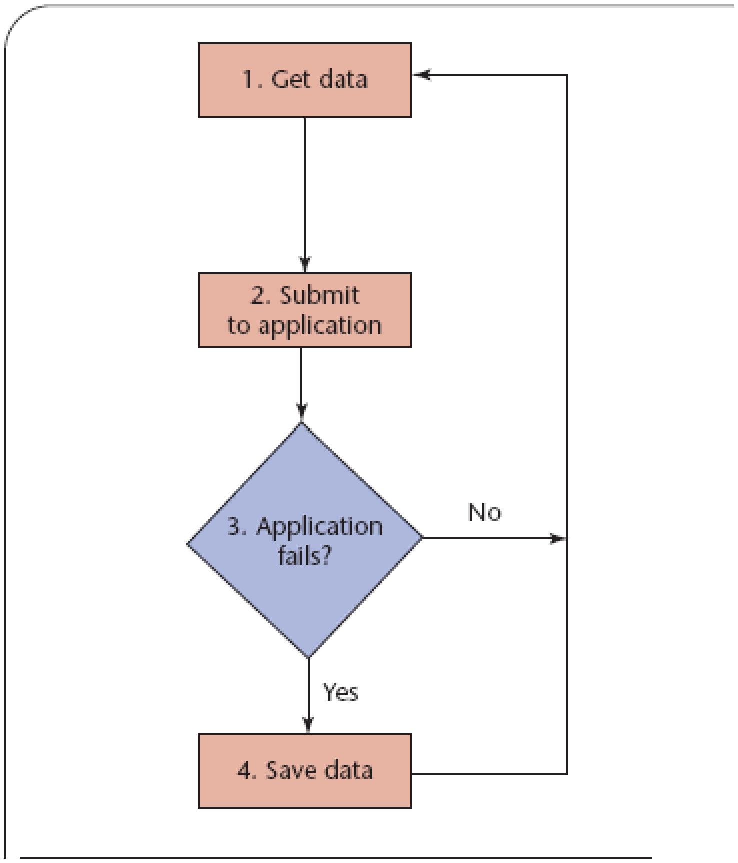


Figure 4. "A complete fuzzer iteration, starting from generation. The fuzzer begins by getting semivalid data via one of the two main methods for use in testing: generation or mutation. The fuzzer then submits the data and tracks whether the erroneous input causes the application to crash (in which case, it saves the data for later analysis). If not, the fuzzer automatically proceeds to the next iteration" [Oehlert 2005]. (Figure 4 taken from [Oehlert 2005])

The software targeted by fuzzing techniques is generally referred to as a parser. A parser is "a set of code that takes data as input and converts it to a form that a program can use. This often means deserializing binary or text data formats into structures or classes that a program can interpret easier" [Oehlert 2005]. In general, parsers analyze data structures, and fall into one of three categories: **file format parsers**, **network protocol parsers**, and **miscellaneous parsers**. **File format parsers** include sets of code that handle graphic image formats (such as JPEG, BMP, WMF, and TIFF) and executable files (such as DOC, PDF, PE, ELF, and SWF) [Howard 2006]. **Network protocol**

parsers are generally those that adhere to “certain standards and rules understood by both sending and receiving parties” that are agreed upon “to communicate data in a meaningful way” [Sutton 2007]. These include sets of code that handle common network protocols such as TCP/IP, SMB, NFS, SSL/TLS, and SNMP. With respect to network protocol parsers, the chronological order in which the network operations are performed can be fuzzed, such as “performing a response before a request” [Howard 2006]. **Miscellaneous parsers** include APIs (application program interfaces) and protocol handlers that can be plugged into a browser [Howard 2006]. Parsers that do not fall under the previous two categories can be grouped into this category as well.

B. THE BENEFITS AND USEFULNESS OF FUZZING

Due to its remarkable ability in uncovering a vast number of security bugs, fuzzing has been applied as a valuable complement to traditional techniques to “discover untested combinations of code and data by combining the power of randomness, protocol knowledge, and attack heuristics.” [DeMott 2006]. Fuzzing provides a remarkable cost-to-benefit ratio partially because the test design is very simplistic. This is particularly useful for software developers who might not have such an accommodating budget for black-box testing. As such, they may need to turn to testing tools that generally have a great track record in discovering faults in software and can be developed at a relatively low cost. Fuzzing also provides for a convenient means of testing because it is a technique that can be readily applied by third-party testers who do not have access to the source code listings of the target software or system that they are testing.

Fuzzing is convenient with respect to time because it is a semi-automated process and it minimizes the necessity for human interaction. Having an automated tool that generates malformed data, provide the data to the target application, and monitoring the target application for exceptions is far more beneficial than having a tester create these test cases by hand and watching the behavior of the target application himself (or herself). Automating the process of finding faults in a target application with as little human interaction as possible is a priority for testers “because the problem of finding bugs is difficult and time consuming [DeMott 2006].” In practice, it is infeasible and far less productive to conduct manual fuzzing. At present, human interaction is only needed "to determine best uses and results" [DeMott 2006].

The knack of finding critical software bugs is what makes fuzzing so valuable. It is a rather inexpensive and simple mechanism that can be assembled with minimal time and resources, yet the majority of the bugs that it finds are security related. Since some of these bugs are exploitable, a system can be compromised as a result of these bugs. This is important because software producers know that producing and delivering patches to fix buggy software is more costly than adhering to quality development from the beginning [Lipner 2005].

C. TYPES OF FUZZING TECHNIQUES

It was mentioned earlier in this document that the automated fuzzing tool would take the set of valid data that was supplied to it, and use some kind of method to malform the data in such a way so that it would be regarded as semivalid (data that is correct enough so that parsers that handle the data will not immediately reject it, yet still invalid enough to trigger undesired or insecure behavior [Oehlert 2005]). Once the target application and the target data type(s) have been chosen, the next step in the fuzzing process is to select how to malform the data that is to be sent.

The **input space** for any data type is defined as "the entire set of all possible permutations that could be sent to the target" [DeMott 2006]. The input space of the data type can be infinite, and it is impossible to try all possible inputs. Furthermore, having more test cases means more audit logs; testers need to have audit logs created to keep track of test cases attempted because they want to detect when a bug or fault has been found, what test case caused each fault, and when the fault occurred [DeMott 2006].

This stresses the importance in determining the method of fuzzing the data because it is desired that the size of the input space and the number of audit logs are minimized as much as possible. The smaller the total number of test cases that need to be generated, the better.

Besides limiting the size of the input space, the fuzzing tool should take into account the efficiency of the selected method in malforming data that results in causing the target application to fail. In general, there are two methods for malforming data to send to the target application: **mutation-based fuzzing** and **generation-based fuzzing.**

In **generation-based fuzzing**, the fuzzer builds the malformed data from scratch "by modeling the target protocol or file format" [Sutton 2007]. This type of fuzzing typically requires researching the specification of the data being malformed. Knowledge of the data format specification enables the testers to construct the fuzzer to "generate data based on a specification for how it should look" [Oehlert 2005]. For a fuzzing tool to successfully generate a malformed set of data, the tool itself will need to be supplied with information about the specification of the data so that it can generate an instance of this data type that is semivalid.

For example, a **generation-based fuzzer** in practice can typically rely upon a set of configuration files provided by the user as a guideline on how to generate semivalid data for the target application to consume. These configuration files, or templates, contain metadata about the structural make-up of the data types used in the fuzzing process. These templates can be thought of "as lists of data structures, their positions relative to each other, and their possible values" [Sutton 2007]. Using these templates as a baseline, the fuzzer can then generate sets of malformed data for each data type.

For instance, suppose the fuzzing tool is working with the BMP image file format, a bitmapped graphics format widely known to Windows users. The basic structure of a BMP image file consists of four data structures. They are listed below, sequentially ordered with respect to its position in the file structure:

- The Bitmap Header – it stores general information about the bitmap file.
- The Bitmap Information Header – it stores detailed information about the bitmap image.
- The Color Palette – it stores the definition of the colors being used for indexed color bitmaps.
- The Bitmapped Data – it stores the actual image, pixel by pixel. [Wikipedia-BitmapImage 2007]

Armed with this information about the BMP structure, the fuzzing tool can generate these data structures of the BMP file, since all four data structures are simply blocks of data containing a series of data elements, and can be easily implemented and generated by a good fuzzing tool. However, the fuzzing tool should be built so that it generates these structures in such a way that the resulting BMP file is malformed to some extent. For example, the fuzzing tool could generate the Bitmap Header with some invalid values, then generate the rest of the data structures with valid values. The Bitmap Header has an data element called "bfOffBits" which stores the specified offset from the beginning of the file to the bitmapped data. Assigning an interesting value to "bfOffBits", such as a very large value, could potentially result in overflowing the buffer of an application that fails to validate image data of BMP files (i.e., checking the size of the image itself), or it could also just simply crash the application. The fuzzer tool could also create a BMP file with a Bitmap Header, but nothing else afterwards (no Bitmapped Information Header, Color Palette, or Bitmapped Data).

In a similar manner, these attack methods can be applied to network protocols and other file formats as well. Like file formats, network protocols can be fuzzed by generated-based fuzzing tools if enough information is provided on the specification of these protocols themselves. In this scenario, not only can generation-based fuzzing create malformed network operations of a specific network protocol, but it can also fuzz the order in which these operations are executed (such as transmitting a response before a request, a situation which some network protocol parsers might not expect). Similar methods used for fuzzing a BMP file can also apply to other file formats, especially image file formats. Since a typical file format often consists of a header (such as the BMP format), strategies can be formed on how to specifically generate malformed instances of a given file format type, such as deciding which values inside the header to fuzz (such as the "bfOffBits" element in the Bitmap Header) and which values should be left intact (such as the "bfType" element in the Bitmap Header, which serves as a signature that identifies the file as a BMP file). It is important that the methods used in generating malformed files should be selected such that it could force the target application to make parsing errors when it handles the malformed files. The goal is to test as much of the functionality of the target application's source code as possible. The more functionality that is tested, the better the chance that a fault can be found, if any such flaws exist.

Mutation-based fuzzing is most often associated with fuzzing file formats. In **mutation-based fuzzing**, the fuzzer takes an existing file that is deemed to be "valid" and mutates in some form or fashion. This method requires gathering several different kinds of files of the target file type used in fuzzing the target application. Essentially, the tester should collect a repository of valid files for this file type; the more different kinds of files found, the better. The ideal scenario is that the library of sample files is a representation of "a broad spectrum of content" for that file type [Howard 2006]. Successfully covering the spectrum of all potential kinds of files for a given file type goes a long way in fulfilling the goal of testing as much of the functionality of the target application's source code as possible.

Once the library of files has been collected, the tester then chooses how to mutate these files. Each mutation of the file is then fed to the target application to see if the file is accepted as valid, rejected the file as invalid, or triggers an exception in the application itself. Of course, it is desired that the mutation-based fuzzer finds a fault in the target.

There are two ways to conduct mutation-based fuzzing: **dumb fuzzing** and **smart fuzzing**. The concept of **dumb fuzzing** is just that; the fuzzer takes a valid file and randomly corrupts it in some form or fashion. Sometimes known as brute force fuzzing, methods mentioned in [Howard 2006] that fall in the category of dumb fuzzing include:

- "Filling the entire file with random data"
- "Filling portions of the file with random data"
- "Searching for null-terminated strings (in ASCII and Unicode) and setting the trailing null to non-null"
- "Setting numeric data types to negative values"
- "Exchanging adjacent bytes"

- “Setting numeric data types to zero”
- “Toggling, setting, or clearing high bits (0x80, 0x8000, and so on)”
- “Performing an exclusive OR (XOR) operation on all bits in a byte, one bit at a time”
- “Setting numeric data types to 2^N ± 1” [Howard 2006]

In smart fuzzing, the mutation-based tool is given some knowledge about the structural make-up of the file format it is fuzzing. The fuzzing tool then uses that information to take a valid file and mutate certain portions of that file. Take for example the PNG (Photographic Networks Group) image file format. A PNG file begins with the mandatory file header, which is an 8-byte signature that identifies itself as a PNG file and contains the hexadecimal value 0x89504E470D1A0A. Following the file header is a series of chunks, with each chunk containing certain information on the image itself [Wikipedia-BitmapImage 2007]. Each chunk in the PNG file format is composed of four structural elements. They are as follows:

- The number of bytes in the data field. It is represented by a 4-byte value to indicate the length.
- The name of the chunk (such as IHDR or IDAT). It is represented by a 4-byte value to indicate the type.
- The data, the format of which depends on the chunk. It is represented by **n** bytes of data.
- A CRC-32 (cyclical redundancy check, 4 bytes) calculated from the data. It is represented by a 4-byte value. [Howard 2006]

The first chunk that follows the file header is always of the IHDR chunk type. This chunk contains the image header and “specifies image dimensions and color information” [Howard 2006]. It is composed of the data elements listed below:

- The image width in pixels, represented by a 4-byte value.
- The image height in pixels, represented by a 4-byte value.

- The number of bits per pixel (1, 2, 4, 8, or 16 bits per pixel), represented by a 1-byte value.
- The color type used by the PNG file (0 for grayscale, 2 for RGB, 3 for palette, 4 for gray with alpha channel, and 6 for RGB and alpha), represented by a 1-byte value.
- The compression mode, always set to zero, represented by a 1-byte value.
- The filter mode, always set to zero, represented by a 1-byte value.
- The interlace mode, 0 for one and 1 for Adam-7 format, represented by a 1-byte value. [Howard 2006]

Using this information about the image file format, a mutation-based fuzzer can be very specific in how it can mutate a valid PNG file. Thus, the mutation-based fuzzing tool can be built to take a valid PNG file and malform it using the following methods:

- "Set the chunk length to a bogus value."
- "Create random chunk names. (They are case sensitive, and the case has specific meaning.)"
- "Build a file with no IHDR chunk."
- "Build a file with more than one IHDR chunk."
- "Set the width, height, or color depth to invalid values (zero, negative values, $2^N \pm 1$, little-endian, and so on)."
- "Set invalid compression, filter, or interlace modes."
- "Set an invalid color type." [Howard 2006]

The goal of **smart fuzzing** is the same as generation-based fuzzing: to test as much of the functionality of the target application's source code as possible, while keeping the total amount of test cases to a minimum. In practice, this is accomplished by trying "to cover every meaningful test case (without too much duplication or unneeded sessions) and to log the ones that succeed in causing the target to fail in some way" [DeMott 2006]. However, it is desired that fuzzed data stays semivalid; the testers want a given test case to not be immediately rejected by the target as invalid data, but they also want the test case to be invalid enough such that it can cause the target to fail.

For instance, since some file formats have built-in safeguards to defend against malformation (such as digital signatures or data checksums), it is desired that mutation-based fuzzing toolsets have some sort of "smart fuzzing" built into the tools themselves, or at least have some methods that specifically attack specific sections of a valid file while keeping the digital signatures and the checksums inside the file intact. With the PNG file format example, the testers have to make sure that the mutation-based fuzzer creates a valid CRC for each chunk inside the PNG file. Otherwise, the target application handling the PNG file will recognize an invalid CRC for a given chunk and simply reject the malformed image.

D. FUZZING APPLIED TO FILE FORMATS

Described in the previous section of this document, fuzzing tools aim at targets that are either file format parsers, network protocol parsers, or miscellaneous parsers. For this thesis, the focus will be on fuzzing file format parsers. As with fuzzing in general, in file format fuzzing the desired result "is to find an exploitable flaw in the way that an application parses a certain type of file" [Sutton 2007]. File format fuzzing is typically narrow in scope, with most of the targets being client-side applications such as image viewers, Web browsers, media players, and office productivity suites [Sutton 2007]. Since these targets are often on the client-side, file format fuzzing can be performed on a single host.

Discovering successful file format attacks tend to be more of an art than a science, as evidenced by the trial-and-error process commonly used in file format fuzzing. The fuzzing techniques covered earlier in this chapter, generation-based fuzzing and mutation-based fuzzing, are the main methods used in the art of fuzzing file formats. Applying fuzzing to local applications that handle file formats is relatively easy. In a manner that is very similar to the iteration fuzzing process described previously in this chapter, the typical file fuzzer performs the following steps:

"1. Prepare a test case, either via mutation or generation" (as described previously in this document)

"2. Launch the target application and instruct it to load the test case."

"3. Monitor the target application for faults, typically with a debugger."

"4. In the event a fault is uncovered, log the finding. Alternatively, if after some period of time no fault is uncovered, manually kill the target application."

"5. Repeat." [Sutton 2007]

To illustrate an example of a file format fuzzer, Figure 5 shows the relatively simple composition of a JPEG image format fuzzing tool that targets the Microsoft Paint image viewer application . The JPEG generator component of the fuzzing tool is built to generate semivalid JPEG images based on the mutation-based and generation-based methods provided by the testers. A target launcher program is used to spawn an instance of Microsoft Paint that will load the current malformed JPEG image created by the JPEG generator. If an exception occurs as a result of the fuzzing, an error detection engine (such as a debugger) monitoring for exceptional conditions will recognize that such an event has occurred and record this exception (i.e., which malformed image was responsible, what type of fault was recorded, and so on).

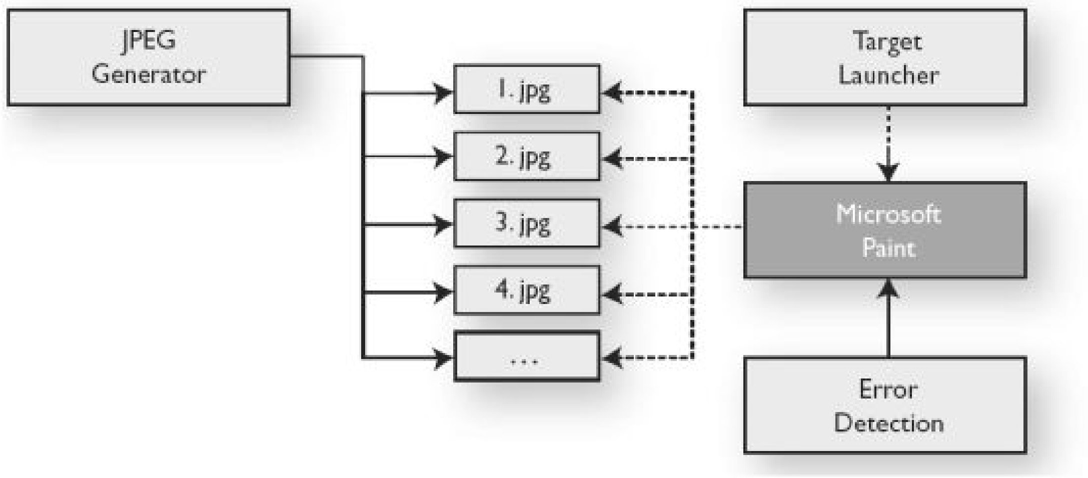


Figure 5. "Fictitious file format fuzzer breakdown and overview" [Sutton 2007]. (Figure 5 taken from [Sutton 2007])

The security faults that are frequently discovered in poorly coded applications that parse file formats generally fall into these category of vulnerabilities [Sutton 2007]:

- **Denial of Service, or DoS** (crash or hang) – Common DoS issues with respect to parsing file formats include "out of bound reads, infinite loops, and NULL pointer dereferences. A common error leading to infinite loops is trusting offset values in files that specify the locations of other blocks within the file. If the application does not make sure this offset is forward in relation to the current block, an infinite loop can occur causing the application to repeatedly process the same block or blocks ad infinitum." [Sutton 2007].
- **Integer handling problems –** If the target program makes the assumption that a certain variable will always contain a positive value, it is a good candidate for being vulnerable to integer overflows. "If the variable has a signed integer type, an overflow can causes its value to wrap and become negative, violating the assumption contained in the program and perhaps leading to unintended behavior." In a similar manner, "subtracting from a small unsigned value may cause it to wrap to a large positive value that can also lead to unexpected behavior" [Wikipedia-IntegerOverflow 2007].
- **Simple stack/heap overflows –** These type of faults are possible when the functionality of the target application fits this following situation when reading in a malformed file: "A fixed size buffer is allocated, whether it be on the stack or on the heap. Later, no bounds checking is performed when copying in oversized data from the file. In some cases, there is some bounding checking, but it is done incorrectly. When the copy occurs, memory is corrupted, often leading to arbitrary code execution" [Sutton 2007].
- **Logic errors –** If there are logic errors in the target application's source code, chances are that the functionality of the target does not exactly match the intended specification. This may result in the target to deviate from its normal behavior when it parses a malformed file. The ability to make an application perform in an

unexpected manner due to semivalid data could lead to possible exploitation, provided that the attacker can transfer control from the application to arbitrary code of the attacker's choice.

- **Format strings –** If the target application uses a format function to print a format string (a series of characters and special format tokens) without performing some kind of input validation on the format string itself, it can be vulnerable to a format string attack. The problem lies in the usage of the format functions themselves. For example, the printf() function uses a form in which "the programmer specifies how the function is to behave explicitly using a format string" [Harris 2005]. If it is used incorrectly, it can result in the function executing in an unexpected manner, which can lead to a vulnerability that is exploitable.
- **Race conditions –** Target applications that typically may have this type of vulnerability are complex multithreaded applications [Sutton 2007]. The vulnerability is present when the target application simultaneously uses uninitialized memory and uses memory that is in use by another thread [Sutton 2007].

When it comes to applying fuzzing to image viewer applications, the target formats must first be identified. Normally, a fuzzer could target several formats belonging to a general category, such as various formats that are all network protocols, or various formats that are all image file formats. In the case of this thesis, the target format is obviously the HD Photo image format. The next step is identifying the target application or software. In the case of this thesis, the target software would be an image viewer application that is compatible with the HD Photo image format.

With respect to Figure 5 shown previously, for this thesis the fuzzer that will be built to conduct security testing will obviously generate and produce HD Photo images instead of JPEG images. Determining if the fuzzing is successful in triggering the application behavior will require attaching a debugger to the target application to see which exceptions have occurred during testing. Once the required components have been acquired for the fuzzing tool, the next step is to brainstorm specific fuzzing strategies with respect to the HD Photo file format. For mutation-based fuzzing, a good start would be to complete a walkthrough, or handtrace, of a sample HD Photo image. This is conducted in the next chapter of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DEVELOPMENT METHODOLOGY OF TEST CASES FOR FUZZING HD PHOTO IMAGE FILES

A. HANDTRACE EXAMPLE OF A VALID HD PHOTO IMAGE

To get an idea of what the internal structure of a valid HD Photo file would look like in practice, it would be best to perform a walkthrough (or handtrace) of an existing HD Photo image. A good start would be to create a very simple instance of an image and see how the specification information released for the HD Photo format matches up with the actual data contents in the file. Then the knowledge learned from the results of the HD Photo file format walkthrough can be used to develop very specific techniques for smart fuzzing and develop other meaningful test cases to attempt with respect to fuzzing this image file format.

A simple example for the HD Photo sample walkthrough would be an image consisting of a single black pixel. This sample image was chosen for its simplicity, since the focus is on fuzzing the header layers of HD Photo, not the image data itself. In this case, the image data represents the black pixel of this image. This image was originally created and saved as a 24-bit Bitmap file using the Microsoft Paint application in Windows XP. A conversion utility to convert between popular image formats and HD Photo was used to convert the image from its native Bitmap format to the HD Photo format, thanks in part to source code made available by Andrei Pociu via the Geekpedia website [Geekpedia 2007]. The source code, which can convert an image file from most of the common image formats used today and save it as an HD Photo file, is written in the C# language and is run inside the Visual Studio 2005 development environment.

The resulting HD Photo image (saved as "hdphoto-1x1blackpixel.wdp") that represents the original 24-bit bitmapped image of a black square (1 pixel wide and 1 pixel high) is 198 bytes in length. The data contents of the HD Photo file will be covered and explained in the pages that follow. The best way to demonstrate the data contents of the HD Photo image is by tracing the HD Photo file by hand. Here, the handtrace is done by traversing the image from the beginning of the file to the end of the file.

Of course, each component of the HD Photo file format will be covered, from the HD Photo Image File Header to the HD Photo Bitstream. They will not be covered all at the same time, since it is easy to be overwhelmed with all the information about the HD Photo image. Hence, the walkthrough approach is written such that it sequentially covers each single piece of data inside the image file from beginning to end, one piece at a time. For each piece of data within the file, there will be an accompanying file offset which tells the reader where it is located with respect to the beginning of the file, along with an explanation of what that piece of data means. Thus, the file offset 0x00 will indicate the data contained at the first byte of the HD Photo image file, and 0xC5 will indicate the data at the last byte of the file (i.e., the 198th byte). For the sake of clarity, an explanation will include the binary equivalent of a data byte's value if one or more data elements in the bitstream are associated with that byte.

After the walkthrough of this sample HD Photo image is completed, it can then be applied as a sample blueprint for fuzzing the HD Photo file format. This can apply to far more complex images as well, since they use pretty much the same data contents from the beginning of the HD Photo File Header to the start of the Tile Layer. Images that are more complex mainly differ from the sample HD Photo image in that they use far more macroblocks to define the visual representation of the image. There are other differences between complex HD Photo images and simple HD Photo images with respect to the data contents, but this will be covered later.

1. Handtrace - HD Photo Image File Header

<table>
<tr><td colspan="2"><b>HD Photo Image File Header</b></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x00 to 0x01</td><td><b>49 49</b></td></tr>
<tr><td colspan="2">This data represents the byte order, or “endian-ness,” used within the image file. As expected, it contains the ASCII characters “II” which corresponds to the TIFF header convention for little-endian byte order for multi-byte numerical formats. As noted in Chapter 2, HD Photo only supports little-endian encoding, so the first two bytes of an HD Photo image file must always be “II”.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x02</td><td><b>BC</b></td></tr>
</table>

This data represents the unique byte value that distinguishes this file as an HD Photo image and not a TIFF 6.0 or some other TIFF-variant file. As expected, it contains the hexadecimal value 0xBC.	
File Offset	**Data contents, in hexadecimal format**
0x03	**01**
This data represents the version number of the HD Photo file structure. Here, the value “01” denotes this HD Photo image to be of version type 1, meaning that this HD file is in compliance with the requirements defined in the released 1.0 HD Photo specification.	
File Offset	**Data contents, in hexadecimal format**
0x04 to 0x07	**08 00 00 00**
This data represents the offset of the first IFD (Image File Directory), in bytes. Here, the hexadecimal value is 0x00000008 (the order is reversed since little-endian encoding is used), or simply 0x08 for short. Thus, the first IFD can be found at file offset 0x08.	

2. Handtrace – HD Photo Image File Directory

The first Image File Directory (IFD) of this HD Photo image file starts at file offset 0x08 and ends at file offset 0x79. The only value in the IFD table longer than 4 bytes is the first IFD entry’s value, the 128-bit GUID which identifies the pixel format for this image. It starts at 0x07A and ends at 0x89.

HD Photo Image File Directory	
File Offset	**Data contents, in hexadecimal format**
0x08 to 0x09	**09 00**
This data represents the first value in an Image File Directory (IFD), a 2-byte count of the number of directory entries, or fields, in the IFD. Here, the number of directory entries in this IFD is 9, as indicated by the hexadecimal value 0x0009 (after reversing the order due to the little-endian encoding).	

a. First IFD Entry

HD Photo Image File Directory, First IFD entry	
File Offset	**Data contents, in hexadecimal format**
0x0A to 0x0B	**01 BC**
This data represents the field’s **Tag** value, 0xBC01, which identifies this directory entry as the ***PixelFormat*** HD Photo field. Thus, the value in this IFD entry represents a 128-bit Globally Unique Identifier (GUID) that specifies the image pixel format.	
File Offset	**Data contents, in hexadecimal format**

<table>
<tr><td>0x0C to 0x0D</td><td><b>01 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Type</b>. Here, the hexadecimal value is 0x0001, which corresponds to a “BYTE” field type, or an 8-bit unsigned integer.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x0E to 0x11</td><td><b>10 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the number of values of the indicated Type, simply referred to as the field <b>Count</b>. Here, the hexadecimal value 0x00000010 indicates a count of sixteen. This means that we should expect sixteen 8-bit unsigned integers, which is the equivalent of 128 bits, the required amount of space necessary to accommodate the 128-bit GUID.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x12 to 0x15</td><td><b>7A 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field’s <b>Value/Offset</b>, which in this case is the 128-bit GUID, which identifies the image pixel format. Since the 128-bit GUID does not fit into the allotted 4-byte space, the hexadecimal value that is provided indicates the <b>file offset</b> where the 128-bit GUID is located within the file. Here, the value 0x0000007A means that the PixelFormat GUID can be found at offset 0x0000007A.</td></tr>
</table>

b. Second IFD Entry

<table>
<tr><td colspan="2"><b>HD Photo Image File Directory, Second IFD Entry</b></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x16 to 0x17</td><td><b>02 BC</b></td></tr>
<tr><td colspan="2">This data represents the field’s <b>Tag</b> value, 0xBC02, which identifies this directory entry as the <b><i>Transformation</i></b> HD Photo field. This optional metadata tag specifies the transformation to be applied when decoding the image to present the desired representation.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x18 to 0x19</td><td><b>04 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Type</b>. Here, the hexadecimal value is 0x0004, which corresponds to a “LONG” field type, or a 32-bit (4-byte) unsigned integer.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x1A to 0x1D</td><td><b>01 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Count</b>. The hexadecimal value 0x00000001 indicates the number of values of the indicated Type, so this means that a single 32-bit (4-byte) unsigned integer is to be expected. Thus the value of the <b><i>Transformation</i></b> <b>Tag</b> can fit within four bytes and will immediately follow this value.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x1E to 0x21</td><td><b>00 00 00 00</b></td></tr>
</table>

This data represents the field's **Value**, which indicates how the transformation is to be applied when decoding the image to display the desired representation.

"There are 8 different possible image orientations as the result of the combination of a ninety-degree clockwise rotation, a horizontal flip, and a vertical flip. The ***Transformation*** **Tag** represents the required transformation to achieve each orientation, as shown in the following table" [HDPhotoFeatureSpec 2006]. Table 15 illustrates the corresponding ID value for each possible orientation of the image.

The meaning of each column are explained in [HDPhotoFeatureSpec 2006] as follows:

"ID: The tag value that specifies the transformation."

"RCW: A value of one specifies that a 90-degree clockwise rotation is applied as the first step of the transformation."

"FlipV: A value of one specifies that vertical flip is applied as part of the transformation, following the rotation."

"FlipH: A value of one specifies that horizontal flip is applied as part of the transformation, following the rotation."

"Orient: This graphically shows the resulting orientation as a result of the transformation."

"Fill: This is an alternate way to describe the requested transformation, specifying the associated two-dimensional fill order of the bitmap as follows:"

> "TL: The 0th row represents the top edge of the image and the 0th column represents the left edge of the image."
>
> "BL: The 0th row represents the bottom edge of the image and the 0th column represents the left edge of the image."
>
> "TR: The 0th row represents the top edge of the image and the 0th column represents the right edge of the image."
>
> "BR: The 0th row represents the bottom edge of the image and the 0th column represents the right edge of the image."
>
> "RT: The 0th row represents the top right of the image and the 0th column represents the top edge of the image."
>
> "RB: The 0th row represents the right edge of the image and the 0th column represents the bottom edge of the image."
>
> "LT: The 0th row represents the left edge of the image and the 0th column represents the top edge of the image."
>
> "LB: The 0th row represents the left edge of the image and the 0th column represents the bottom edge of the image."

"TIFF: This is the value of the TIFF-compatible *Orientation* tag that would result in the same transformation. HD Photo does not use the *Orientation* tag." [HDPhotoBitstreamSpec 2006]

ID	*RCW*	*FlipH*	*FlipV*	*Orient*	*Fill*	*TIFF*
0	0	0	0	P	TL	1
1	0	0	1	b	BL	4

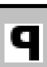
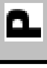
2	0	1	0		TR	2
3	0	1	1		BR	3
4	1	0	0		RT	6
5	1	0	1		RB	7
6	1	1	0		LT	5
7	1	1	1		LB	8

Table 15. Transformation HD Photo Tag Data Table, taken from [HDPhotoFeatureSpec 2006]

For this image, the hexadecimal value of 0x00000000 is given as the value for ID, the tag value that specifies the transformation. A zero value for ID indicates that a value of zero is applied to RCW, a value of zero is applied to FlipV, and a value of zero is applied to FlipH. This means that during the transformation, no 90-degree clockwise rotation is applied in the first step, no vertical flip is applied as part of the transformation following the first step, and no horizontal flip is applied following the first step. Thus, when this image is decoded, no changes are made with respect to how the image is displayed, so the resulting image should remain untransformed. The HD Photo Feature Specification makes an interesting note about the information pertaining to the Transformation IFD tag:

> Whenever an image is encoded, it should be stored in an un-transformed orientation and the Transformation tag should be reset to zero. If an application changes the *Transformation* metadata tag (effectively requesting a transform to be performed when the image is decoded) then the application must also make sure the values for ImageWidth and ImageHeight are correct (swap them if the transform includes a rotation.) [HDPhotoFeatureSpec 2006].

c. Third IFD Entry

HD Photo Image File Directory, Third IFD Entry	
File Offset	**Data contents, in hexadecimal format**
0x22 to 0x23	**04 BC**
This data represents the field's **Tag** value, 0xBC04, which identifies this directory entry as the ***ImageType*** HD Photo field. This optional metadata tag specifies the image type of each individual image in a multi-image file. This tag should not be used when a file contains a single image.	

File Offset	**Data contents, in hexadecimal format**
0x24 to 0x25	**04 00**
This data represents the field **Type**. Here, the hexadecimal value is 0x0004, which corresponds to a “LONG” field type, or a 32-bit (4-byte) unsigned integer.	
File Offset	**Data contents, in hexadecimal format**
0x26 to 0x29	**01 00 00 00**
This data represents the field **Count**. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single 32-bit (4-byte) unsigned integer is to be expected. Thus the value of the ***ImageType*** **Tag** can fit within four bytes and will immediately follow this value.	
File Offset	**Data contents, in hexadecimal format**
0x2A to 0x2D	**00 00 00 00** (00000000 00000000 00000000 00000000 in binary)
This data represents the field’s **Value**, which indicates how to specify the image type of each individual image in this file. The value of this tag is interpreted as a bit field, with a specific meaning associated with each of the 32 bits. In Version 1.0 of the Microsoft HD Photo Feature Specification, only two bits are defined, with the remaining bits being reserved for future use and thus should be either ignored or set to zero. The meaning of each of the two bits is explained in the immediate paragraphs that follow.	

Bit 0 is the **Preview** bit. If this bit is set, then the image represented by this bitstream is a preview (i.e., an alternate representation) of another image in this HD Photo file and is typically encoded as a reduced resolution thumbnail or preview, often for the purpose of providing a simplified preview of the image that is not computation intensive.

The value provided for ***ImageType*** has Bit 0 cleared (set to zero), so the image specified by this IFD table is not an alternative representation of another image in the file. This makes logical sense, since there is only one IFD table that exists in the entire file. This means that this file contains a single image, even though the IFD table is using this tag. The HD Photo Feature Specification document provides further explanation about a Preview image in HD Photo.

> A HD Photo file should contain only one Preview image. If multiple images contain the ImageType metadata tag with this bit set, only the first image should be used and any subsequent Preview images should be ignored. The Preview image must be encoded in a basic pixel format. Ideally, it should be encoded in one of these two pixel formats:

> WICPixelFormat24bppBGR or WICPixelFormat8bppGray. If any advanced pixel format is used, there is no guarantee that all decoders will be able to access the preview image.
>
> To minimize the preview image size, an appropriate lossy compression is also recommended. A Preview image can be used in a number of different ways. It provides a high performance method to retrieve a displayable representation, especially for very large images. A preview can provide a basic pixel format representation of an image stored in an advanced pixel format. A preview image can also be used to present an alternative view of a larger image, such as a cropped region.
>
> Applications should only create a Preview image and set this metadata tag bit when it makes logical sense. Small images in basic pixel formats may not require a preview since it is just as easy to decode the entire image [HDPhotoFeatureSpec 2006].

Bit 1 is the **Page** bit. The image specified by this IFD table is an individual page in a sequence of pages within file. However, in HD Photo version 1.0, the use of multiple images within a single container file is not supported. Even though this HD Photo tag is documented in the HD Photo Feature Specification (version 1.0), it is recommended that this tag not be used in a HD Photo 1.0 file. The following documentation is in anticipation of its use in potential future updates to the HD Photo file format.

As an alternative to using the Page bit, an encouraged option for describing individual image pages in an HD Photo file for multiple images is including the TIFF 6.0 metadata tags PageNumber and PageName. These tags are capable of describing the individual image pages.

The value provided for ***ImageType*** has Bit 1 cleared (set to zero), so the image specified by this IFD table is **not** an individual page in a sequence of pages within the file.

d. Fourth IFD Entry

<table>
<tr><th colspan="2">HD Photo Image File Directory, Fourth IFD entry</th></tr>
<tr><th>File Offset</th><th>Data contents, in hexadecimal format</th></tr>
<tr><td>0x2E to 0x2F</td><td><b>80 BC</b></td></tr>
<tr><td colspan="2">This data represents the field’s <b>Tag</b> value, 0xBC80, which identifies this directory entry as the <b><i>ImageWidth</i></b> HD Photo field. This metadata tag specifies the number of columns in the transformed photo, or the number of pixels per scan line. However, because there is no actual transformation taking place, this tag simply indicates the number of columns in the original photo. The <b><i>ImageWidth</i></b> field is a mandatory metadata tag for all HD Photo image files, and has no default value. This value must be explicitly set.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><th>File Offset</th><th>Data contents, in hexadecimal format</th></tr>
<tr><td>0x30 to 0x31</td><td><b>04 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Type</b>. Here, the hexadecimal value is 0x0004, which corresponds to a “LONG” field type, or a 32-bit (4-byte) unsigned integer.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><th>File Offset</th><th>Data contents, in hexadecimal format</th></tr>
<tr><td>0x32 to 0x35</td><td><b>01 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Count</b>. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single 32-bit (4-byte) unsigned integer is to be expected. Thus the value of the <b><i>ImageWidth</i></b> <b>Tag</b> can fit within four bytes and will immediately follow this value.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><th>File Offset</th><th>Data contents, in hexadecimal format</th></tr>
<tr><td>0x36 to 0x39</td><td><b>01 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field’s <b>Value</b>, which indicates the image’s number of columns, or its width in pixels. The hexadecimal value 0x00000001 indicates that the width of the image is one pixel.</td></tr>
</table>

e. Fifth IFD Entry

<table>
<tr><th colspan="2">HD Photo Image File Directory, Fifth IFD entry</th></tr>
<tr><th>File Offset</th><th>Data contents, in hexadecimal format</th></tr>
<tr><td>0x3A to 0x3B</td><td><b>81 BC</b></td></tr>
<tr><td colspan="2">This data represents the field’s <b>Tag</b> value, 0xBC81, which identifies this directory entry as the <b><i>ImageHeight</i></b> HD Photo field. This metadata tag specifies the number of rows of pixels in the transformed photo, or the number of scan lines. However, because there is no actual transformation taking place, this tag simply indicates the number of rows of pixels in the original photo. The <b><i>ImageHeight</i></b> field is a mandatory metadata tag for all HD Photo image files, and has no default value. This value must be explicitly set.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><th>File Offset</th><th>Data contents, in hexadecimal format</th></tr>
<tr><td>0x3C to 0x3D</td><td><b>04 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Type</b>. Here, the hexadecimal value is 0x0004, which corresponds to a “LONG” field type, or a 32-bit (4-byte) unsigned integer.</td></tr>
</table>

<table>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x3E to 0x41</td><td><b>01 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Count</b>. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single 32-bit (4-byte) unsigned integer is to be expected. Thus the value of the <b><i>ImageHeight</i></b> <b>Tag</b> can fit within four bytes and will immediately follow this value.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x42 to 0x45</td><td><b>01 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field's <b>Value</b>, which indicates the image's number of scan lines, or its height in pixels. The hexadecimal value 0x00000001 indicates that the height of the image is one pixel.</td></tr>
</table>

f. Sixth IFD Entry

<table>
<tr><td colspan="2"><b>HD Photo Image File Directory, Sixth IFD Entry</b></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x46 to 0x47</td><td><b>82 BC</b></td></tr>
<tr><td colspan="2">This data represents the field's <b>Tag</b> value, 0xBC82, which identifies this directory entry as the <b><i>WidthResolution</i></b> HD Photo field. This optional metadata tag specifies the horizontal resolution of a transformed image expressed in pixels per inch. If this value is zero or this optional tag is not present, then a default resolution of 96dpi (abbreviation for 96 dots per inch) is to be assumed.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x48 to 0x49</td><td><b>0B 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Type</b>. Here, the hexadecimal value is 0x000B, which corresponds to a "FLOAT" field type, or a number that follows a single precision (4-byte) IEEE format in little-endian (LSB [Least Significant Byte] first) byte order.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x4A to 0x4D</td><td><b>01 00 00 00</b></td></tr>
<tr><td colspan="2">This data represents the field <b>Count</b>. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single number in the 4-byte single precision IEEE format is to be expected. Thus the value of the <b><i>WidthResolution</i></b> <b>Tag</b> can fit within four bytes and will immediately follow this value.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x4E to 0x51</td><td><b>00 00 C0 42</b></td></tr>
<tr><td colspan="2">This data represents the field's <b>Value</b>, which indicates the image's horizontal resolution in pixels per inch. The hexadecimal value 0x42C00000 (reversed because of the little-endian encoding used by HD Photo) represents that number in 4-byte single precision IEEE format.</td></tr>
</table>

In the IEEE standard for binary floating-point arithmetic, the single-precision (32-bit) format can be computed in the following manner. In single-precision mode, binary floating-point numbers are treated as signed numbers, or numbers which are represented by a magnitude (its absolute value) and a sign (positive or negative). The most significant bit represents the sign bit, which designates whether the number is positive or negative. The biased exponent, or "exponent," is represented by the next eight bits after the most significant bit. The rest of the bits in the 32-bit binary value represent the "fraction" value. Thus the first bit is the sign bit, the next eight bits form the exponent, and the remainder of the 32-bit number (the last 23 bits) forms the "fraction". [Wikipedia-IEEE754 2007]

The number has the value **V**, computed using the formula:

$\mathbf{V} = S \times 2^{e} \times M$

where

S = +1 (this condition is met when the sign bit equals 0, so the number is positive)

S = -1 (this condition is met when the sign bit equals 1, so the number is negative)

e = Exponent – 127

M = 1.fraction in binary (i.e., "the binary number 1 followed by the radix point followed by the binary bits of the fraction" [Wikipedia-IEEE754]).

So it should follow that M is always between the values of 1 and 2. ($1 < M < 2$)

Breaking down the previously described hexadecimal value into its 32-bit binary equivalent, we get the following:

Hexadecimal value: 0x42C00000

Binary equivalent: 0100 0010 1100 0000 0000 0000 0000 0000

	Sign	Exponent (8 bits)	Fraction (23 bits)
0x42C00000 →	0	10000101	10000000000000000000000

The sign bit is 0 → S = +1

e = Exponent – 127 = (100000101 in binary) – 127 = $(2^7 + 2^2 + 2^0) – 127$

$$= (128 + 4 + 1) - 127$$

$$= 133 - 127 = 6$$

$M = 1.\text{fraction}$ (in binary form) = (1.1 in binary) = $(2^{0} + 2^{-1}) = 1 + 0.5 = 1.5$

With all the essential elements known, we can compute the single-precision IEEE number:

$$V = S \times 2^{e} \times M$$

$$V = (+1) \times 2^{6} \times (1.5) = (+1) \times 64 \times (1.5) = \mathbf{96}$$

Thus, the horizontal resolution of this image is 96 dots per inch. If this HD Photo tag were not present, the horizontal resolution would have a default value of 96dpi. This means that this HD Photo metadata tag probably was not needed for this particular image.

g. Seventh IFD Entry

HD Photo Image File Directory, Seventh IFD Entry	
File Offset	**Data contents, in hexadecimal format**
0x52 to 0x53	**83 BC**
This data represents the field's **Tag** value, 0xBC83, which identifies this directory entry as the ***HeightResolution*** HD Photo field. This metadata tag is optional, and it specifies the vertical resolution of a transformed image expressed in pixels per inch. If this value is zero or this optional tag is not present, then a default resolution of 96dpi (abbreviation for 96 dots per inch) is to be assumed.	
File Offset	**Data contents, in hexadecimal format**
0x54 to 0x55	**0B 00**
This data represents the field **Type**. Here, the hexadecimal value is 0x000B, which corresponds to a "FLOAT" field type, or a number that follows a single precision (4-byte) IEEE format in little-endian (LSB [Least Significant Byte] first) byte order.	
File Offset	**Data contents, in hexadecimal format**
0x56 to 0x59	**01 00 00 00**
This data represents the field **Count**. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single number in the 4-byte single precision IEEE format is to be expected. Thus the value of the ***HeightResolution*** **Tag** can fit within four bytes and will immediately follow this value.	
File Offset	**Data contents, in hexadecimal format**
0x5A to 0x5D	**00 00 C0 42**

This data represents the field’s **Value**, which indicates the image’s vertical resolution in pixels per inch. The hexadecimal value 0x42C00000 (reversed because of the little-endian encoding used by HD Photo) represents that number in 4-byte single precision IEEE format.

Referring to the previous example in computing the horizontal resolution of this image, the vertical resolution can be calculated using the same method. In this case, using the result from the previous calculation performed for the horizontal resolution, the vertical resolution is 96 dots per inch. If this HD Photo tag were not present, the vertical resolution would have a default value of 96dpi. This means that this HD Photo metadata tag probably was not needed for this particular image.

For this image, the horizontal resolution and the vertical resolution would have remained unchanged even without the use of the ***WidthResolution*** and the ***HeightResolution*** tags by this IFD to explicit define the resolution values.

h. Eighth IFD Entry

HD Photo Image File Directory, Eighth IFD Entry	
File Offset	**Data contents, in hexadecimal format**
0x5E to 0x5F	**C0 BC**
This data represents the field’s **Tag** value, 0xBCC0, which identifies this directory entry as the ***ImageOffset*** HD Photo field. This metadata tag specifies the byte offset pointer to the beginning of the photo data, relative to the beginning of this HD photo file. The ***ImageOffset*** field is a mandatory metadata tag for all HD Photo image files, and has no default value. This value must be explicitly set.	
File Offset	**Data contents, in hexadecimal format**
0x60 to 0x61	**04 00**
This data represents the field **Type**. Here, the hexadecimal value is 0x0004, which corresponds to a “LONG” field type, or a 32-bit (4-byte) unsigned integer.	
File Offset	**Data contents, in hexadecimal format**
0x62 to 0x65	**01 00 00 00**
This data represents the field **Count**. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single 32-bit (4-byte) unsigned integer is to be expected. For the ***ImageOffset*** **Tag**, the provided value for this tag will not immediately follow this value because the ***ImageOffset*** field is always provided as an offset.	
File Offset	**Data contents, in hexadecimal format**
0x66 to 0x69	**8A 00 00 00**
This data represents the field’s **Offset**, which in this case is a hexadecimal value that indicates the **file offset** where the photo data is located within the file. Here, the value 0x0000008A means that the photo data of the image described by this IFD can be found at **offset 0x0000008A** with respect to the beginning of the HD Photo image file.	

i. Ninth IFD Entry

HD Photo Image File Directory, Ninth IFD Entry	
File Offset	**Data contents, in hexadecimal format**
0x6A to 0x6B	**C1 BC**
This data represents the field's **Tag** value, 0xBCC1, which identifies this directory entry as the ***ImageByteCount*** HD Photo field. This metadata tag specifies the size of the photo data in bytes. The ***ImageByteCount*** field is a mandatory metadata tag for all HD Photo image files, and has no default value. This value must be explicitly set.	
File Offset	**Data contents, in hexadecimal format**
0x6C to 0x6D	**04 00**
This data represents the field **Type**. Here, the hexadecimal value is 0x0004, which corresponds to a "LONG" field type, or a 32-bit (4-byte) unsigned integer.	
File Offset	**Data contents, in hexadecimal format**
0x6E to 0x71	**01 00 00 00**
This data represents the field **Count**. The hexadecimal value 0x00000001 indicates that the number of values of this field is one, so a single 32-bit (4-byte) unsigned integer is to be expected. Thus the value of the ***ImageOffset*** **Tag** can fit within four bytes and will immediately follow this value.	
File Offset	**Data contents, in hexadecimal format**
0x72 to 0x75	**3C 00 00 00**
This data represents the field's **Value**, which indicates the size of the photo data in bytes. The hexadecimal value 0x0000003C (reversed because of the little-endian encoding used by HD Photo) represents that number. Converting the hexadecimal value into its decimal equivalent: 0x0000003C → $(3 * 16^1) + (12 * 16^0) = (3 * 16) + (12 * 1) = (48) + (12) =$ **60 bytes** For the image described by this IFD, the size of the photo data is **60 bytes.**	

This marks the end of 1st Image File Directory table in the sample HD Photo file. After all the field entries of an Image File Directory in a HD Photo file are processed, either one of the two things can happen when reading in the next four bytes that follow the last field entry.

The first scenario is that there is at least one more IFD contained inside the HD Photo file that has not been processed yet. In this case, the last IFD entry is followed by a 32-bit byte offset to the next IFD to be processed. This offset must be non-zero.

The second scenario is that this is the last IFD in the file. In this case, the next four bytes should all be null bytes, indicating that there are no more Image File Directories to be processed. (This implies the assumption that no IFD starts at the beginning of the HD photo file.)

End of First IFD in HD Photo File, Offset to Next IFD	
File Offset	**Data contents, in hexadecimal format**
0x76 to 0x79	**00 00 00 00**

As mentioned previously, this data represents either the offset to the next Image File Directory (if it exists) or four null bytes to indicate that this IFD is the last one in the HD Photo image file. The four null bytes after the ninth (and last) entry in the Image File Directory indicates that the IFD previously processed was indeed the last IFD in the file. Thus, there is only one image in this HD Photo file. This is what should be expected, since version 1.0 of HD Photo does not support image files that contain more than one image.

128-bit GUID of HD Photo Image Described By First IFD	
File Offset	**Data contents, in hexadecimal format**
0x7A to 8x89	**24 C3 DD 6F 03 4E FE 4B B1 85 3D 77 76 8D C9 0C**

This is the 128-bit Globally Unique Identifier of the first image in this as previously described in the PixelFormat Tag defined in the first field entry of the IFD. In reading in the GUID, the first four bytes (represented by the initial hexadecimal value 0x24C3DD6F) are reversed before being processed. The next two bytes (0x034E) are reversed before being read-in, and so are the next two bytes (0xFE4B). The remaining eight bytes in the GUID are read as-is, so no reversing is done. Then the resulting value for the GUID is: **6FDDC324-4E03-4BFE-B1853D77768DC90C**.

Referring to Table 16 (originally from the HD Photo Feature Specification Version 1.0 document), this GUID is linked to the PixelFormat name **WICPixelFormat24bppBGR**. This indicates that the image is represented by a RGB format. The table further indicates that the image uses three channels, with 8 bits per channel, for a total of 24 bits per pixel.

PixelFormat Name ***GUID***	***Ch***	***BPC***	***BPP***	***Num***	***Color***	***A***	***B***
WICPixelFormat24bppRGB 6fddc324-4e03-4bfe-b1853d77768dc90d	3	8	24	UINT	RGB		✓

WICPixelFormat24bppBGR 6fddc324-4e03-4bfe-b1853d77768dc90c	3	8	24	UINT	RGB		✓
WICPixelFormat32bppBGR 6fddc324-4e03-4bfe-b1853d77768dc90e	3	8	24	UINT	RGB		
WICPixelFormat48bppRGB 6fddc324-4e03-4bfe-b1853d77768dc915	3	16	48	UINT	RGB		✓
WICPixelFormat48bppRGBFixedPoint 6fddc324-4e03-4bfe-b1853d77768dc912	3	16	48	SINT	RGB		✓
WICPixelFormat48bppRGBHalf 6fddc324-4e03-4bfe-b1853d77768dc93b	3	16	48	Float	RGB		
WICPixelFormat96bppRGBFixedPoint 6fddc324-4e03-4bfe-b1853d77768dc918	3	32	96	Float	RGB		
WICPixelFormat128bppRGBFloat 6fddc324-4e03-4bfe-b1853d77768dc91b	3	32	128	Float	RGB		
WICPixelFormat32bppBGRA 6fddc324-4e03-4bfe-b1853d77768dc90f	4	8	32	UINT	RGB	✓	
WICPixelFormat64bppRGBA 6fddc324-4e03-4bfe-b1853d77768dc916	4	16	64	UINT	RGB	✓	
WICPixelFormat64bppRGBAFixedPoint 6fddc324-4e03-4bfe-b1853d77768dc91d	4	16	64	SINT	RGB	✓	
WICPixelFormat64bppRGBAHalf 6fddc324-4e03-4bfe-b1853d77768dc93a	4	16	64	Float	RGB	✓	
WICPixelFormat128bppRGBAFixedPoint 6fddc324-4e03-4bfe-b1853d77768dc91e	4	32	128	SINT	RGB	✓	
WICPixelFormat128bppRGBAFloat 6fddc324-4e03-4bfe-b1853d77768dc919	4	32	128	Float	RGB	✓	
WICPixelFormat32bppPBGRA 6fddc324-4e03-4bfe-b1853d77768dc910	4	8	32	UINT	RGB	✓	
WICPixelFormat64bppPRGBA 6fddc324-4e03-4bfe-b1853d77768dc917	4	16	64	UINT	RGB	✓	
WICPixelFormat128bppPRGBAFloat 6fddc324-4e03-4bfe-b1853d77768dc91a	4	32	128	Float	RGB	✓	

Table 16. Listing of all the RGB image pixel formats supported by the HD Photo 1.0 codec, taken from [HDPhotoFeatureSpec 2006]

3. Handtrace – HD Photo Bitstream

The last part of the sample HD Photo image file is the HD Photo Bitstream. For this sample image, it is represented by the range of data from file offset 0x8A to 0xC5. The Image Header starts at file offset 0x8A and ends at file offset 0x99. The Image Plane Header starts at 0x9A and ends at 0xA3. The Index Table and the Tile Layer make up the rest of the bitstream. Each component of this image's bitstream is covered in the sections that follow, in the order that they appear in the bitstream.

a. Image Header

<table>
<tr><td colspan="2"><b>HD Photo Image Header (IMAGE_HEADER)</b></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x8A to 0x91</td><td><b>57 4D 50 48 4F 54 4F 00</b></td></tr>
<tr><td colspan="2">This data is the first eight bytes of the HD Photo Bitstream that represents the 64-bit <b>GDI Signature</b> (<b>GDISIGNATURE</b>) element in the Image Header. Its purpose is to identify the bitstream. As expected, the hexadecimal value found here is 0x574D50484F544F00, which corresponds to "WMPHOTO" in ASCII.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x92</td><td><b>10</b> (0001 0000 in binary)</td></tr>
<tr><td colspan="2">VERSION 4 bits 0001
SUBVERSION 4 bits 0000</td></tr>
<tr><td colspan="2">The first four bits of this byte (0001) is the <b>Codec Version</b> (<b>VERSION</b>), which specifies the version of the bitstream. It should have a value of 1, with all other values being reserved for future purposes. As seen here, this element does indeed have a value of 1.
The last four bits of this byte (0000) the <b>Codec Sub-version</b> (<b>SUBVERSION</b>), which specifies the sub-version of the bitstream. Here, it is assigned a value of 0. According to the HD Photo Bitstream Specification 1.0, this value is ignored by the decoder.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x93</td><td><b>00</b> (0000 0000 in binary)</td></tr>
<tr><td colspan="2">TILING_FLAG 1 bit 0
BITSTREAM_FORMAT 1 bit 0
ORIENTATION 3 bits 000
INDEXTABLE_PRESENT_FLAG 1 bit 0
OVERLAP 2 bits 00</td></tr>
<tr><td colspan="2"><b>TILING_FLAG</b> is set to a zero value, so this image is said to be untiled.
<b>BITSTREAM_FORMAT</b> is set to 0, so the bitstream for this image is laid out in the Spatial mode.
<b>ORIENTATION</b> is set to 0, so no transformation is done to the image.
<b>INDEXTABLE_PRESENT_FLAG</b> is set to 0, so no index table is present. This
<b>OVERLAP</b> is set to 0, so no post filtering was performed on this image.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td><b>File Offset</b></td><td><b>Data contents, in hexadecimal format</b></td></tr>
<tr><td>0x94</td><td><b>C0</b> (1100 0000 in binary)</td></tr>
</table>

SHORT_HEADER_FLAG	1 bit	1
LONG_WORD_FLAG	1 bit	1
WINDOWING_FLAG	1 bit	0
TRIM_FLEXBITS_FLAG	1 bit	0
TILE_STRETCH_FLAG	1 bit	0
RESERVED	2 bits	00
ALPHACHANNEL_FLAG	1 bit	0

SHORT_HEADER_FLAG is set to 1, so smaller bit sizes are used to represent the height and width of the image.
LONG_WORD_FLAG is set to 1, so 16 bit integers are used for transform computations.
WINDOWING_FLAG is set to 0, so windowing is not present in this bitstream.
TRIM_FLEXBITS_FLAG is set to 0, so Flexbits are not retained in full precision. For this image, we don't care about this value, since Flexbits isn't present in this bitstream.
TILE_STRETCH_FLAG is set to 0. We will ignore this value, since this element is always set to 0 for version 1.0 of HD Photo.
RESERVED_FLAG is set to 0. We will ignore this value, since this element always defaults to a zero value for this version of HD Photo.
ALPHACHANNEL_FLAG is set to 0, so no alpha channel is present in the bitstream.

File Offset	Data contents, in hexadecimal format		
0x95	**71** (0111 0001 in binary)		
	SOURCE_CLR_FMT	4 bits	0111
	SOURCE_BITDEPTH	4 bits	0001

SOURCE_CLR_FMT is assigned a value of 0111 in binary, or 7 in decimal form.
Using the table provided for SOURCE_CLR_FMT in the Bitstream Specification and again in Chapter 2, this corresponds to the "**RGB"** source color format representation.
SOURCE_BITDEPTH is assigned a value of 0001 in binary, or 1 in decimal form.
Using the table provided for SOURCE_BITDEPTH in the Bitstream Specification, this corresponds to the unsigned integer **"BD_8"**, which corresponds to 8 bits per channel.

File Offset	Data contents, in hexadecimal format
0x96 to 0x99	**00 00 00 00** (00000000 00000000 00000000 00000000 in binary)

WIDTH_MINUS1	**16 bits**	**0000 0000 0000 0000**
HEIGHT_MINUS1	**16 bits**	**0000 0000 0000 0000**

At this point in reading in the photo data, there is a check to see if the SHORT_HEADER_FLAG is set to 1. If it is, then the next two elements to read in are the 16-bit **Width** element (**WIDTH_MINUS1**) and the 16-bit **Height** element (**HEIGHT_MINUS1**). The **WIDTH_MINUS1** element specifies the width of the coded area, minus 1. The **HEIGHT_MINUS1** element specifies the height of the coded area minus 1. Thus, the width and height of the coded area is defined as follows:

The first sixteen bits represent the value for **WIDTH_MINUS1**, which is zero.
The next sixteen bits represent the value for **HEIGHT_MINUS1**, which is zero.
Actual width of image in pixels: **Width** = **WIDTH_MINUS1** + 1 = 0 + 1 = **1**
Actual height of image in pixels: **Height** = **HEIGHT_MINUS1** + 1 = 0 + 1 = **1**
If the SHORT_HEADER_FLAG were cleared (initialized to 0), then the **WIDTH_MINUS1** and **HEIGHT_MINUS1** elements would have been 32-bit elements instead of 16-bit elements.

The TILING_FLAG element is set to 0 (FALSE). Since the elements **NUM_VERT_TILES_MINUS1** and **NUM_HORIZ_TILES_MINUS1** are present only if TILING_FLAG is set to 1 (TRUE), these elements are declared not present and are both inferred to have a value of zero.
Since the elements **NUM_VERT_TILES_MINUS1** and **NUM_HORIZ_TILES_MINUS1** both have zero values, the elements **WIDTH_IN_MB_OF_TILEI[n]** and **HEIGHT_IN_MB_OF_TILEI[n]** are not present.
The **NUM_VERT_TILES_MINUS1** and **NUM_HORIZ_TILES_MINUS1** elements are used to determine the number of image tiles in the image. Here is the formula:
NumSpatialTiles = (NUM_VERT_TILES_MINUS1 + 1) * (NUM_HORIZ_TILES_MINUS1 + 1) = (0 + 1) * (0 + 1) = 1 * 1 = **1**

The TILE_STRETCH_FLAG is not set (has a zero value), so the TILE_STRETCH[n] element is not used.
The WINDOWING_FLAG is set to 0 (FALSE), so the 6-bit syntax elements NUM_TOP_EXTRAPIXELS, NUM_LEFT_EXTRAPIXELS, NUM_BOTTOM_EXTRAPIXELS, and NUM_RIGHT_EXTRAPIXELS are not used and their values are set to zero.
The height and width of the image area deemed as "viewable" are derived as follows:
ImageWidth = Width – NUM_LEFT_EXTRAPIXELS – NUM_RIGHT_EXTRAPIXELS
= 1 – 0 – 0 = 1
ImageHeight = Height – NUM_TOP_EXTRAPIXELS – NUM_BOTTOM_EXTRAPIXELS
= 1 – 0 – 0 = 1

b. HD Photo Image Plane Header

HD Photo Image Plane Header (IMAGE_PLANE_HEADER)	
File Offset	**Data contents, in hexadecimal format**
0x9A	**60** (0110 0000 in binary)

CLR_FMT	3 bits	011
NO_SCALED_FLAG	1 bit	0
BANDS_PRESENT	4 bits	0000

CLR_FMT has a binary value of 011, or a decimal value of 3.
From the Color Format table provided in the Bitstream Specification, a value of 3 corresponds to the **YUV_444** color format.
CLR_FMT serves another purpose; it is used to define the number of channels (**NChannels**). Here, **CLR_FMT** having an assigned value of “YUV_444” corresponds to having the number of channels set to three (**NChannels** = 3).

NO_SCALED_FLAG has a binary value of 0, indicating a Boolean value of FALSE. Thus, this image uses scaling.

BANDS_PRESENT has a binary value of 0000, or a decimal value of 0. From the Bands Present table provided in the Bitstream Specification, a value of 0 corresponds to the **SB_ALL** interpretation. This means that all subbands are present.

The element NumSubBandsPresent represents the number of bands present in the bitstream. According to the Bitstream Specification, the value of **BANDS_PRESENT** determines the value of NumSubBandsPresent. Since the BANDS_PRESENT element corresponds to the value **SB_ALL**, the NumSubBandsPresent element is assigned a decimal value of 4. So there are 4 bands present in the bitstream.

File Offset	**Data contents, in hexadecimal format**
0x9B	**01 (0000 0001 in binary)**
	CHROMA_CENTERING 4 bits 0000 COLOR_INTERPRETATION 4 bits 0001

Since the Color Format is “YUV_444”, the next two elements in the Image Plane Header are CHROMA_CENTERING and COLOR_INTERPRETATION.
These elements are read in this order in the bitstream only for this type of color format.
CHROMA_CENTERING is assigned a value of zero.
COLOR_INTERPRETATION is assigned a value of one.
As of version 1.0 of HD Photo, both elements are ignored by the decoder.

File Offset	**Data contents, in hexadecimal format**
0x9C to 0x9F	**A0 00 0A 00** (1010 0000 0000 0000 0000 1010 0000 0000 in binary)
0xA0 to 0xA3	**00 A0 00 00** (0000 0000 1010 0000 0000 0000 0000 0000 in binary)

Data elements read in from bitstream while processing bytes 0x9C to 0xA3, part 1:

DC_FRAME_UNIFORM	1 bit	1
CH_MODE	2 bits	01
DC_QUANT_Y	8 bits	00000000
DC_QUANT_UV	8 bits	00000000

Data elements read in from bitstream while processing bytes 0x9C to 0xA3, part 2:

USE_DC_QUANTIZER	1 bit	0
LP_FRAME_UNIFORM	1 bit	1
CH_MODE	2 bits	01
LP_QUANT_Y[q] (for q=1)	8 bits	00000000
LP_QUANT_UV[q] (for q=1)	8 bits	00000000

Data elements read in from bitstream while processing bytes 0x9C to 0xA3, part 3:

USE_LP_QUANTIZER	1 bit	0
HP_FRAME_UNIFORM	1 bit	1
NUM_HP_QUANTIZERS is set to 1		
CH_MODE	2 bits	01
HP_QUANT_Y[q] (for q=1)	8 bits	00000000
HP_QUANT_UV[q] (for q=1)	8 bits	00000000
(Bits read in from bitstream while processing the FLUSHBYTE function)	5 bits	00000

Explanation of processing bytes 0x9C to 0xA3 in the Image Plane Header of the example HD Photo file, part 1:

- A check is made to see if CLR_FMT is set to “N_CHANNEL”, as seen in the syntax table of the Image Plane Header (listed in this thesis as Table 2). It is not set to “N_CHANNEL”, so the code is not executed for this condition (i.e., NUM_CHANNELS_MINUS1 is not read in from the bitstream).
- A similar check is made to see if SOURCE_CLR_FMT is set to “BAYER”. It is not, so the code is not executed for this condition.
- A similar check is made to see if SOURCE_BITDEPTH is equal to either “BD_16”, BD_16S”, “BD_32”, or “BD_32S”. Since SOURCE_BITDEPTH is set to “BD_8”, the code is not executed for this condition.
- Another check is made on SOURCE_BITDEPTH to see if it is equal to “BD_32F”. Since it is not, the code is not executed for this condition.
- The 1-bit element DC_FRAME_UNIFORM is read from the bitstream. For this image, this element is set to the binary value of 1, or “True”. Thus, a single quantizer shall be used for the DC bands of each color plane in the image, and this quantizer shall be signaled in the image plane header.
- At this juncture in processing the Image Plane Header, the DC_QUANTIZER structure is then read from the bitstream. The DC_QUANTIZER function is called to read in the DC_QUANTIZER structure in the bitstream.

- The start of the process in reading the DC_QUANTIZER structure is to check the value assigned to the element “NChannels”. From [HDPhotoBitstreamSpec 2006], this element is set to 3. This means that the 2-bit element CH_MODE is determined from the bitstream. From reading the next two bits in the bitstream, this element is set to the binary value 01. From Chapter 2, a binary value of 01 assigned to CH_MODE translates to the “CH_SEPARATE” mode, as seen in Table 10. So the next step is to execute the code for the condition that the CH_MODE is equal to “CH_SEPARATE”.

  Note: For the sake of completeness, this handtrace includes an explanation of how each Quantizer structure is processed. This means including reading in the CH_MODE element in DC_QUANTIZER to understand how the processing of the Image Plane Header in the bitstream works. This also applies to the LP_QUANTIZER and HP_QUANTIZER structures, since the CH_MODE element can be present in the other two Quantizer structures as well, and not just the DC_QUANTIZER structure. However, even though the CH_MODE element is a part of the Quantizer structures, the elements found inside these structures are not a focus of the fuzzing scheme with respect to HD Photo.
- As a result of CH_MODE being equal to “CH_SEPARATE”, the next step is to read in the 8-bit DC_QUANT_Y element in the DC_QUANTIZER structure from the bitstream. It is set to the binary value 00000000.
- Read in the 8-bit DC_QUANT_UV element in the DC_QUANTIZER structure in the bitstream. It is set to the binary value 000000000. After this element is read, the processing of the DC_QUANTIZER structure is complete.

Explanation of processing bytes 0x9C to 0xA3 in the Image Plane Header of the example HD Photo file, part 2:

- Upon completion of reading the DC_QUANTIZER structure from the bitstream, the next step in processing the Image Plane Header is to check the value of the BANDS_PRESENT element, since it determines what elements are the next ones to be read in from the bitstream. For this image, BANDS_PRESENT is set to the value “SB_ALL”. From the syntax table for the Image Plane Header (listed in Chapter 2 as Table 6), the first condition checked is to see if BANDS_PRESENT is not equal to “SB_DC_ONLY”. Since it is not, the syntax table dictates that the Boolean element USE_DC_QUANTIZER should be read from the bitstream. For this image, this element has a binary value of 0, meaning that it is set to “false”.
- Since USE_DC_QUANTIZER is set to “false”, the next step is to read the Boolean element LP_FRAME_UNIFORM from the bitstream. Since this element has a binary value of 1, this means that it is set to “true” and the next element in the bitstream is the LP_QUANTIZER structure. Before calling the LP_QUANTIZER function to read in the LP_QUANTIZER structure, the NUM_LP_QUANTIZERS element is set to 1.
- The first step in the LP_QUANTIZER function is to see if the value of the NChannels element is equal to 1. It was seen earlier that NChannels was set to 3, so this condition is not met, and this means that the next element to read from the bitstream is CH_MODE. Note that this element might vary between the different Quantizer structures in HD Photo. For the LP_QUANTIZER structure, it is determined from the bitstream that this element has the binary value of 01. As stated earlier, this translates to the “CH_SEPARATE” mode, so the next step is to execute the code for the condition that the CH_MODE is equal to “CH_SEPARATE”. This means that the next two elements to read in from the bitstream are LP_QUANT_Y[q] and LP_QUANT_UV[q].
- From the bitstream, it is determined that LP_QUANT_Y[q] (for q=0) has an 8-bit binary value of 00000000, and that LP_QUANT_UV[q] (for q=0) has an 8-bit binary value of 00000000.
- After these elements are read, the processing of the LP_QUANTIZER structure is complete.

Explanation of processing bytes 0x9C to 0xA3 in the Image Plane Header of the example HD Photo file, part 3:

- Upon completion of reading the LP_QUANTIZER structure from the bitstream, the next step in processing the Image Plane Header is to again check the value of the BANDS_PRESENT element, since it determines what elements are the next ones to be read in from the bitstream. For this image, BANDS_PRESENT is set to the value “SB_ALL”. From the syntax table for the Image Plane Header (listed in Chapter 2 as Table 6), the second condition checked is to see if BANDS_PRESENT is not equal to “SB_NO_HIGHPASS.” Since it is not, the syntax table dictates that the Boolean element USE_LP_QUANTIZER should be read from the bitstream. For this image, it is determined from the bitstream that this element has a binary value of 0, meaning that it is set to “false”. Since USE_LP_QUANTIZER is set to “false”, the next step is to read the Boolean element HP_FRAME_UNIFORM from the bitstream.
- From the bitstream, the HP_FRAME_UNIFORM has a binary value of 1, so this means that it is set to “true” and the next element in the bitstream is the HP_QUANTIZER structure. Before calling the HP_QUANTIZER function to read in the HP_QUANTIZER structure, the NUM_HP_QUANTIZERS element is set to 1.
- In a very similar manner as the LP_QUANTIZER function, the first step in the HP_QUANTIZER function is to see if the value of the NChannels element is equal to 1. It was seen earlier that NChannels was set to 3, so this condition is not met, and this means that the next element to read from the bitstream is CH_MODE. For the HP_QUANTIZER structure, it is determined from the bitstream that this element has the binary value of 01. As stated earlier, this translates to the “CH_SEPARATE” mode, so the next step is to execute the code for the condition that the CH_MODE is equal to “CH_SEPARATE”. This means that the next two elements to read in from the bitstream are HP_QUANT_Y[q] and HP_QUANT_UV[q].
- From the bitstream, it is determined that HP_QUANT_Y[q] (for q=0) has an 8-bit binary value of 00000000, and that HP_QUANT_UV[q] (for q=0) has an 8-bit binary value of 00000000.
- After these elements are read, the processing of the HP_QUANTIZER structure is complete.
- The last processing step in the Image Plane Header structure is to call the FLUSHBYTE operator. As stated earlier in Chapter 2, FLUSHBYTE is a variable length syntax element that accounts for the flush-to-byte operation. The length of this element is somewhere between 0 and 7 bits, depending on how many bits need to be read in order to align the bitstream ‘pointer’ with the HD Photo bitstream. For this image, the FLUSHBYTE operator reads in 5 bits (all zeroes), and discards the bits. The result of running the FLUSHBYTE operator is that the bitstream ‘pointer’ is now byte-aligned, and it is now looking at file offset 0xA4 of this sample HD Photo image file.

c. HD Photo Index Table

Index Table (INDEX_TABLE)	
File Offset	**Data contents, in hexadecimal format**
0xA4	**FF** (1111 1111 in binary)
FIRST_BYTE	**8 bits 1111 1111**

In processing the Index Table structure in the bitstream of this sample HD Photo image, we refer to the syntax table for this structure, previously listed in Chapter 2 as Table 4.
The first step is to check if the INDEXTABLE_PRESENT_FLAG element is set to true or false. In the Image Header structure, this element was set to false. Since the INDEXTABLE_STARTCODE and INDEX_OFFSET_TILE[n] elements would be read from the bitstream only if this element was set to true, the code in Table 4 that would read in these two elements does not execute (i.e., the "if-loop" is skipped).

After the "if-loop" set of code, the next step is to read in the SKIP_BYTES element. This means calling the VLWESC function, previously documented in Chapter 2 of this thesis. The value of SKIP_BYTES depends on the value returned by the VLWESC function.
The VLWESC function reads in the next 8 bits from the HD Photo bitstream and assigns this data as the value to the local variable FIRST_BYTE. As seen above, FIRST_BYTE is set to the binary value of 11111111, the hexadecimal equivalent of 0xFF.
The VLWESC function then executes code based on the value of FIRST_BYTE.
Since FIRST_BYTE is greater than the hexadecimal value 0xFC, the VLWESC function executes the last piece of code, which sets the local variable "Value" to zero. The function then returns the value of "Value" back to INDEX_TABLE, which then assigns this value to the SKIP_BYTES element.
Thus, SKIP_BYTES is set to zero.

The last step in processing the Index Table is to read in the PADDING_DATA element. The bit length of this element is equal to the value of SKIP_BYTES multiplied by 8. Since SKIP_BYTES is zero, so is the bit length of PADDING_DATA, and no data is read from the bitstream for this element. This concludes the processing of the Index Table Structure, and begins the processing of the Tile Layer for this image.

d. Tile Layer

Tile Layer (TILE)	
File Offset	**Data contents, in hexadecimal format**
0xA5 to 0xA7	**00 00 01** (0000 0000 0000 0000 0000 0001 in binary)
TILE_STARTCODE	**24 bits 00000000 00000000 00000001**

TILE_STARTCODE is a 24-bit syntax element that indicates the start of a tile. Here, TILE_STARTCODE is equal to 1, so this tile is considered to be a valid tile and decodable.	
File Offset	**Data contents, in hexadecimal format**
0xA8	**00** (0000 0000 in binary)
TILE_LOCATION_HASH 5 bits 00000 TILE_TYPE 3 bits 000	
TILE_LOCATION_HASH is a 5-bit syntax element whose value is determined by this formula: (VerticalTileIndex * NumberOfHorizontalTiles + HorizontalTileIndex) % 32 There is one exception to this formula. In the case where the TYLE_TYPE is equal to “Flexbits”, TILE_LOCATION_HASH can take on any value. This value, however, should match with its respective index table entry. TILE_TYPE, the element that describes the type of data contained in this tile, has a value of 0 for this image, which corresponds to the Tile Type value of “Spatial”. Hence this is a “Spatial tile”.	

Macroblock Layer / Block Layers inside Tile Layer	
File Offset	**Data contents, in hexadecimal format**
0xA9 to 0xAF	**CE 01 00 00 00 00 00**
0xB0 to 0xBF	**00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00**
0xC0 to 0xC5	**00 00 00 00 8E 40**
The rest of the data in the sample image file, from file offset 0xA9 to 0xC5, correspond to the Macroblock Layer and Block Layers that exist in the Tile Layer. Essentially, it represents the visual presentation of the image itself, complete with color information. This data includes the values set for the Macroblock DC coefficients, the macroblock-level AC coefficients (or “lowpass” coefficients), and the lower-level AC coefficients. **For this image, this data translates to a completely black background. This is what we should expect, since the original source image is a single jet-black pixel.**	

This completes the handtrace of the sample HD Photo image. Using the walkthrough, a general blueprint (or template) can be made to fuzz the HD Photo file format against selected image viewer applications. This is made possible by looking at the walkthrough itself and seeing how each of the data elements in the sample HD Photo file correspond to the specifications listed for the HD Photo file structure detailed in Chapter II. Once these observations can be made, various test cases can be developed for use in generation-based fuzzing and mutation-based fuzzing with respect to this file format. This is covered in the next section of this chapter.

Before going on to developing fuzzing test cases, this walkthrough can also be analyzed with respect to steganography. From the handtrace conducted on this sample HD Photo image, can it be determined whether a secret message can be embedded into the HD Photo file without altering the visual representation of the image itself? One key requirement to successfully hide information into a digital image is that a person cannot notice or detect any visual differences between the original image (before embedding the secret inside the image) and the altered image (after embedding the secret).

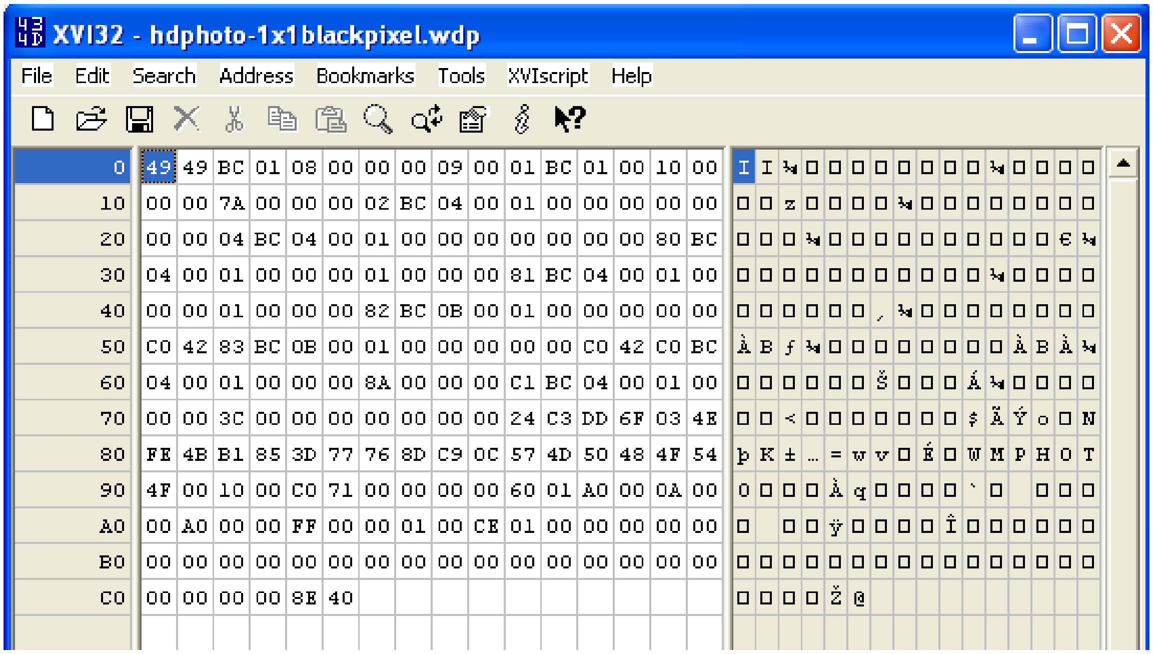
There is one interesting case to consider that is related to the sample HD Photo image that was just previously handtraced: instead of just a single black pixel, what if the image used in the walkthrough was a completely black macroblock, 16 jet black pixels wide and 16 pixels high? What would be the key differences in the resulting HD Photo image? A similar handtrace was done on an HD Photo image of a 16-pixel-wide, 16-pixel-high black image. Just like the previous image, this image was initially created as a 24-bit bitmap image and converted into an HD Photo image. A similar walkthrough was then performed on the resulting HD Photo file.

From comparing the two HD Photo files described above, it turns out that these two images are very similar in terms of file structure. Both images have the same number of total bytes (each HD Photo file is 198 bytes in length), and both have the same identical set of IFD entry tags (such as the PixelFormat tag, the Transformation tag, the ImageType tag, and so on). The only differences between the single-pixel image (1 pixel wide and high) and the macroblock image (16 pixels wide and high) were the values for the ImageWidth and ImageType tags in the HD Photo Image File Directory, and the values for the WIDTH_MINUS1 and the HEIGHT_MINUS1 data elements in the HD Photo Bitstream. For the single-pixel image, the value of both the ImageWidth and ImageType tag was 0x00000001 (which translates to a width and height value of 1). For the macroblock image, the value of both the ImageWidth and ImageType tag was 0x00000010 (which translates to a width and height value of 16). For the single-pixel image, the values assigned to the WIDTH_MINUS1 element and the HEIGHT_MINUS1 element are 0x00000000, while the other image had both WIDTH_MINUS1 and HEIGHT_MINUS1 set to 0x0000000F.

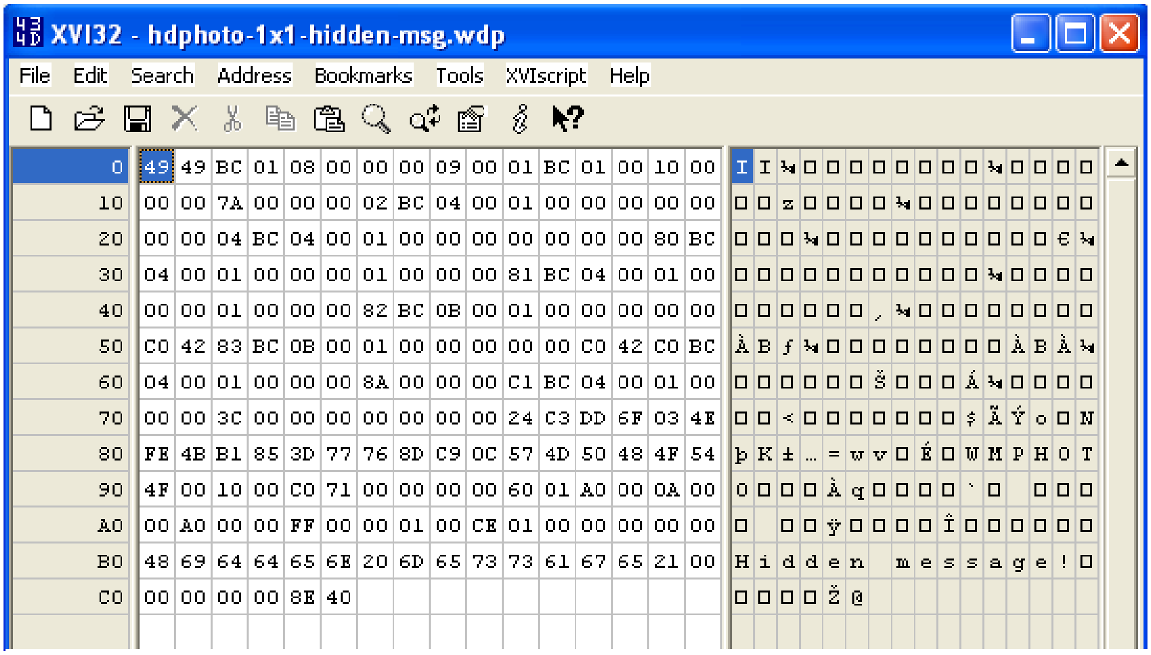
One key observation from comparing the two images is that (apart from the WIDTH_MINUS1 and HEIGHT_MINUS1 data elements) the actual image data for the macroblock image is the same as the image data of the single-pixel image that was handtraced earlier. This implies that not all of the image data factors into the actual image shown. Remember from Chapter 2 that HD Photo images are formed in multiples of macroblocks. For the singe black pixel image, only a portion of the macroblock is actually shown (since it is just a single pixel).

From observing the differences between the 1-pixel-wide, 1-pixel-high black image and the 16-pixel-wide, 16-pixel-high black image, one can take advantage of the fact that blocks do go past the image boundaries for HD Photo images whose height and width are not multiples of 16, and use the portions of the blocks not shown in the picture to embed hidden data (steganography). Thus, at least for certain conditions, steganography is indeed possible in HD Photo.

To shown an example of how one can use the HD Photo image of the solitary black pixel to embed a hidden message, we employ the use of a hex editor. A standard hex editor program can take any binary file and edit it in some form or fashion, such as an HD Photo file. The Freeware Hex Editor XVI32 program, which can be downloaded at http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm, for no charge, was used to edit the HD Photo file of the 1-pixel-wide, 1-pixel-high black image. Figure 6 shows the comparison of the contents of the original single-pixel HD Photo file, and the contents of the same file after embedding hidden data inside the file. As seen in Figure 6, the string "Hidden message!" was inserted in the original image in the range of bytes from offset 0xB0 to offset 0xBE, and was saved as "hdphoto-1x1-hidden-msg.wdp". Despite the modification of the original image by embedding this message inside the file, the visual presentation of the original image is not affected in any way. It is still the same black pixel presented in the HD Photo file format. This means that either the image data represented by the bytes used to store the hidden message represents the color coefficients of the portions of the macroblock that go past the image boundaries of the original image, or it simply does not factor into the visual presentation of the image itself (i.e., used for other purposes such as padding).



```
 0  49 49 BC 01 08 00 00 00 09 00 01 BC 01 00 10 00   I I ¼ □ □ □ □ □ □ □ □ ¼ □ □ □ □
10  00 00 7A 00 00 00 02 BC 04 00 01 00 00 00 00 00   □ □ z □ □ □ □ ¼ □ □ □ □ □ □ □ □
20  00 00 04 BC 04 00 01 00 00 00 00 00 00 00 80 BC   □ □ □ ¼ □ □ □ □ □ □ □ □ □ □ € ¼
30  04 00 01 00 00 00 01 00 00 00 81 BC 04 00 01 00   □ □ □ □ □ □ □ □ □ □ □ ¼ □ □ □ □
40  00 00 01 00 00 00 82 BC 0B 00 01 00 00 00 00 00   □ □ □ □ □ □ ‚ ¼ □ □ □ □ □ □ □ □
50  C0 42 83 BC 0B 00 01 00 00 00 00 00 C0 42 C0 BC   À B ƒ ¼ □ □ □ □ □ □ □ □ À B À ¼
60  04 00 01 00 00 00 8A 00 00 00 C1 BC 04 00 01 00   □ □ □ □ □ □ Š □ □ □ Á ¼ □ □ □ □
70  00 00 3C 00 00 00 00 00 00 00 24 C3 DD 6F 03 4E   □ □ < □ □ □ □ □ □ □ $ Ã Ý o □ N
80  FE 4B B1 85 3D 77 76 8D C9 0C 57 4D 50 48 4F 54   þ K ± … = w v □ É □ W M P H O T
90  4F 00 10 00 C0 71 00 00 00 00 60 01 A0 00 0A 00   O □ □ □ À q □ □ □ □ ` □   □ □ □
A0  00 A0 00 00 FF 00 00 01 00 CE 01 00 00 00 00 00   □   □ □ ÿ □ □ □ □ Î □ □ □ □ □ □
B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
C0  00 00 00 00 8E 40                                 □ □ □ □ Ž @
```



```
 0  49 49 BC 01 08 00 00 00 09 00 01 BC 01 00 10 00   I I ¼ □ □ □ □ □ □ □ □ ¼ □ □ □ □
10  00 00 7A 00 00 00 02 BC 04 00 01 00 00 00 00 00   □ □ z □ □ □ □ ¼ □ □ □ □ □ □ □ □
20  00 00 04 BC 04 00 01 00 00 00 00 00 00 00 80 BC   □ □ □ ¼ □ □ □ □ □ □ □ □ □ □ € ¼
30  04 00 01 00 00 00 01 00 00 00 81 BC 04 00 01 00   □ □ □ □ □ □ □ □ □ □ □ ¼ □ □ □ □
40  00 00 01 00 00 00 82 BC 0B 00 01 00 00 00 00 00   □ □ □ □ □ □ ‚ ¼ □ □ □ □ □ □ □ □
50  C0 42 83 BC 0B 00 01 00 00 00 00 00 C0 42 C0 BC   À B ƒ ¼ □ □ □ □ □ □ □ □ À B À ¼
60  04 00 01 00 00 00 8A 00 00 00 C1 BC 04 00 01 00   □ □ □ □ □ □ Š □ □ □ Á ¼ □ □ □ □
70  00 00 3C 00 00 00 00 00 00 00 24 C3 DD 6F 03 4E   □ □ < □ □ □ □ □ □ □ $ Ã Ý o □ N
80  FE 4B B1 85 3D 77 76 8D C9 0C 57 4D 50 48 4F 54   þ K ± … = w v □ É □ W M P H O T
90  4F 00 10 00 C0 71 00 00 00 00 60 01 A0 00 0A 00   O □ □ □ À q □ □ □ □ ` □   □ □ □
A0  00 A0 00 00 FF 00 00 01 00 CE 01 00 00 00 00 00   □   □ □ ÿ □ □ □ □ Î □ □ □ □ □ □
B0  48 69 64 64 65 6E 20 6D 65 73 73 61 67 65 21 00   H i d d e n   m e s s a g e ! □
C0  00 00 00 00 8E 40                                 □ □ □ □ Ž @
```

Figure 6. A comparison of the file contents between the original image (a single black pixel saved as "hdphoto-1x1blackpixel.wdp", shown at the top) and the same image embedded with the hidden message "Hidden message!" (saved as "hdphoto-1x1-hidden-msg.wdp", shown at the bottom). The contents of each file is shown in hexadecimal format on the left-hand side, and its ASCII equivalent is shown on the right-hand side.

With the steganography aspect of this thesis addressed (albeit in a brief manner), the focus can now turn to the fuzzing aspect of the HD Photo file format.

B. DEVELOPMENT OF WINDOWS MEDIA PHOTO FUZZING BLUEPRINT AND FUZZING TEST CASES

Now that an example handtrace has been performed on a sample valid HD Photo image, the next step is to take the results of the handtrace and use it as a guideline in constructing a fuzzing blueprint for the HD Photo file format. This blueprint can be used as the foundation for developing test cases for both mutation-based fuzzing test cases and generation-based fuzzing test cases. This is ideal, since the best approach would be using both mutation-based fuzzing and generation-based fuzzing to test this image file format. Pertaining to the mutation-based fuzzing aspect, it would be easy to just perform dumb fuzzing on this file format, but it would be of greater benefit if we have some test cases to perform smart fuzzing instead of just randomly manipulating data across a valid HD Photo file.

For mutation-based fuzzing, the first step is to take a completely valid HD Photo image and use it as a foundation to build use cases on top of it. The key is to start with a known good HD Photo file and find ways to malform it in such a way that we are actually doing boundary value analysis with respect to this file format. We want the resulting HD Photo files that we create to be semi-valid; valid in the sense that the target application does not reject the input HD Photo file outright, but invalid such that (as a result of handling this semi-valid HD Photo file) the target application operates in a way that deviates from its normal and specified behavior (such as failing to terminate gracefully). Thus, the goal of the HD Photo fuzzer is to test the target viewer application and verify that it does not do something it shouldn’t do, rather than do something it should do [Oehlert 2005].

A good starting point needs to be determined to prevent an HD Photo file from being rejected outright by image viewer applications. It would be best to begin with keeping fields that are scrutinized by hard checks, such as the first three bytes of the HD Photo File Header (0x4949BC) and the GDI signature of the Image Header in the HD Photo Bitstream. Thus, all our test cases should have these checks built in.

It is of little benefit to exclusively use completely random input, or dumb fuzzing, as test cases if almost all of them will be invalidated by the target application due to failing the built-in required validation checks inside the application itself (such as having the required three-byte signature 0x4949BC at the beginning of the file, or the mandatory GDI signature “WMPHOTO” at the start of the bitstream). It would not be feasible anyway to run and analyze the number of test cases of this particular type in a reasonable amount of time because the required amount of test cases based purely on random input to find a single bug would be at an order of magnitude far too high to be practical. By using a completely valid image, manipulating parts of the file, and sending the semi-valid file as input to the target application, we have a better chance of finding possible bugs in the way that the target application reads in an HD Photo file and goes through its method of either (1) rejecting the file as an invalid image and informing the user via an error message, or (2) accepting the file as a valid image and then attempting to display the image to the user.

Since an ideal approach to fuzzing HD Photo files would be a combined approach of mutation-based and generation-based test cases, with a combination of “smart fuzzing” and “dumb fuzzing” techniques used for the mutation-based test cases, it would be best to develop test scenarios for both mutation-based fuzzing and generation-based fuzzing. This includes developing a number of “smart fuzzing” cases as well as “dumb fuzzing” cases, since it is desired to have the benefits of both types of mutation-based fuzzing.

1. Mutation-Based Fuzzing Test Cases Using “Dumb Fuzzing”

Looking back at the HD Photo sample handtrace conducted previously, the following “dumb fuzzing” cases described below were developed for mutation-based fuzzing of the HD Photo file format. These cases were built based on the techniques described for mutation-based “dumb fuzzing” in Chapter 3 of this document.

- Case 1: Abruptly truncate a valid HD Photo file. This is to test how a parser handles a HD Photo file that is unexpectedly incomplete.
- Case 2: Replace a randomly-selected range of bytes inside a valid HD Photo file with random values. This includes:

    - (a) Filling the entire file with random data. This tests a parser’s ability to reject a HD Photo file that is completely scrambled with random data.
    - (b) Filling portions of the file with random data. This tests a parser’s ability to reject a HD Photo file that is partially scrambled with random data.
- Case 3: Find an ASCII string value inside the IFD of a valid HD Photo file and replace it with an unexpected value. This includes:
    - (a) Searching for null-terminating strings (in ASCII and Unicode) and setting the trailing null to non-null (maybe for IFD entries using “ASCII” as the data type). This could be employed with Adobe Photoshop metadata tags that are compatible with the TIFF, and they may be compatible with the HD Photo format as well. This tests how the parser handles invalid ASCII values provided in an IFD tag.
- Case 4: Toggle on or off the most significant bits of a series of bytes inside a valid HD Photo file. This includes:
    - (a) Toggling, setting, or clearing high bits (0x80, 0x8000, and so on). This tests the sensitivity of the parser in regards to whether certain bits in a HD Photo file are set or cleared.
- Case 5: Perform an exclusive OR (XOR) operation on a range of bytes (selected at random) inside a valid HD Photo file with random values. Similar to Case 4, this also tests the sensitivity of the parser in regards to whether certain bits in a HD Photo file are set or cleared.
- Case 6: Switch adjacent bytes inside a valid HD Photo file. This is to test the order in which the parser reads in a HD Photo file byte by byte.

2. Mutation-based Fuzzing Test Cases Using “Smart Fuzzing”

Looking back at the HD Photo sample handtrace conducted previously, the following “smart fuzzing” cases were developed for mutation-based fuzzing of the HD Photo file format. These cases were developed based on what parts of the HD Photo file would be most beneficial to the testing process if they were fuzzed. In the cases that follow, the application being tested is referred to as a parser.

- Case 1: Fuzzing the value of the Version Number in the HD Photo File Header. This would test how the application would handle a HD Photo file with a future version (not version 0 or version 1).

- Case 2: Fuzzing the number of entries in the IFD. This tests the image viewer application's ability to correctly parse the actual number of valid entries inside the Image File Directory.

- Case 3: Fuzzing the offset to the PixelFormat field in the IFD. This value is mandatory in HD Photo, and it indicates the offset where one can find the unique pixel format of the rendered image. This tests how the parser determines if the offset to the PixelFormat field is valid.

- Case 4: Fuzzing the contents of the IFD Transformation field. This tests the parser's ability to handle valid transformations (especially if the transformation does not match the given values for the image's height and width).

- Case 5: Fuzzing the contents of the IFD ImageType field. This tests how the parser interprets the value provided by this optional IFD tag. It also tests how the parser handles a HD Photo file containing a single image; any HD Photo file with just one image should not have the ImageType tag for version 1.0 of this format.

- Case 6: Fuzzing the contents of the IFD ImageWidth field. Fuzzing the value of the "ImageWidth" IFD tag in an HD Photo file may affect the way in which a parser handles an HD Photo file if the parser trusts this value to be correct. This value is mandatory in HD Photo, and it indicates the width of the rendered image.

- Case 7: Fuzzing the contents of the IFD ImageHeight field. Fuzzing the value of the "ImageHeight" IFD tag in an HD Photo file may affect the way in which a parser handles an HD Photo file if the parser trusts this value to be correct. This value is mandatory in HD Photo, and it indicates the height of the rendered image.

- Case 8: Fuzzing the contents of the IFD WidthResolution field. This tests the parser's ability to handle any given value to the image's width resolution and determine if it is valid or not. Thus, it should reject ridiculous values such as positive infinity or negative infinity.

- Case 9: Fuzzing the contents of the IFD HeightResolution field. This tests the parser's ability to handle any given value to the image's height resolution and determine if it is valid or not. Thus, it should reject ridiculous values such as positive infinity or negative infinity.

- Case 10: Fuzzing the contents of the IFD ImageOffset field (image data offset). This tests the parser's ability to determine if an offset provided to the image data actually contains a valid HD Photo Bitstream.

- Case 11: Fuzzing the contents of the IFD ImageByteCount field (i.e. the image data in bytes). This value is mandatory in HD Photo, and it indicates the length of the image data in bytes. This could possibly used by an image viewer application in prematurely expecting the number of bytes it should read from the HD Photo Bitstream.

- Case 12: Fuzzing the contents of the IFD offset to next IFD (instead of the four trailing null bytes). This tests the parser's ability to determine if a non-zero offset provided to the next IFD actually contains a valid secondary Image File Directory.

- Case 13: Fuzzing the contents of the HD Photo Bitstream Image Width syntax element. This tests whether the parser checks if the actual image width matches the information provided by the IFD ImageWidth tag.

- Case 14: Fuzzing the contents of the HD Photo Bitstream Image Height syntax element. This tests whether the parser checks if the actual image height matches the information provided by the IFD ImageHeight tag.

2. Generation-Based Fuzzing Test Use Cases

Based on the walkthrough results, the following test cases listed below were developed for use with generation-based fuzzing of the HD Photo file format. Unless otherwise noted, the physical portions of the HD Photo file that are not mentioned in a specific test case can be assumed to be unmodified, with the exception of the actual image data located after the Tile Layer in the HD Photo Bitstream (because it contains the Macroblock and Block layers). The test cases are developed using the 198-byte sample image as the foundational blueprint (i.e., the template) for generation-based fuzzing. For example, for the first test case described (using a version number other than zero or one in the HD Photo File Header), just the third byte in the 198-byte image is changed for that fuzzed file, and the rest of the file is left unchanged (with the exception noted above for the actual image data).

- Case 1: Generate a HD Photo file with a wrong version number, that is, any number greater than one in the third byte of the HD Photo file. This tests the parser to see if it rejects any HD Photo file with a version number other than zero or one.

- Case 2: Generate a HD Photo file that omits one of the mandatory IFD tags (e.g., PixelFormat tag and the ImageWidth tag) required by any HD Photo file in order to be considered "valid." From the handtrace example performed earlier, we see that there are at least seven mandatory IFD entries used by HD Photo. Fuzz testing should center on how the image viewer application handles mandatory IFD entries and optional IFD entries. This tests the parser to see how it handles a HD Photo file that is generated in this manner; a good parser should reject any HD Photo file that has required tags missing in the IFD.

- Case 3: Generate a HD Photo file with all IFD tags made semi-valid by fuzzing the value field. This tests the parser to see if it can handle a HD Photo file that may contain IFD tags with invalid values in the Value field.

- Case 4: Generate a HD Photo file with an invalid GDI signature, that is, a signature that is not equal to the ASCII string "WMPHOTO". This partially tests how the HD Photo Bitstream is validated by the image viewer application.

- Case 5: Generate a HD Photo file with valid values for every component up to the HD Photo Bitstream (excluding the GDI signature), but with the entire HD Photo Bitstream missing. The purpose of this test case is to check if the parser can recognize all possible forms that a valid HD Photo Bitstream can take. Variations of this case include:
    - The case described above, but with both the Image Header and Image Plane Header missing from the HD Photo Bitstream, instead of the entire HD Photo Bitstream.
    - The case described above, but just the Image Plane Header missing from the HD Photo Bitstream, instead of the entire HD Photo Bitstream.
- Case 6: Generate a HD Photo file with the number of directory entries in the IFD set to a random value. This tests the parser's ability to reject a HD Photo file which contains an incorrect value for the total number of entries for an IFD.
- Case 7: Generate a HD Photo file with the ImageWidth and ImageHeight IFD tags not being in ascending order. This tests the parser's ability to read in the ImageWidth and ImageHeight container tags correctly.
- Case 8: Generate a HD Photo file with at least one IFD Entry with Type set to FLOAT (IEEE format), but set the value of the tag to a random value which may be valid or invalid. This tests the parser's ability that it can validate any value for any given IFD tag for all possible data types.
- Case 9: Generate a HD Photo file with valid HD Photo IFD tags, but with randomly generated values for each of the tag's data Type. This tests how the parser handles invalid or unexpected values in the Type field for each IFD tag.
- Case 10: Generate a HD Photo file with the first IFD advertising an offset to a second IFD, but only one IFD exists in the file. This case reveals how the image viewer application handles an HD Photo file that does have a non-zero offset after the first IFD, but does not contain a valid IFD at that location. Not handling this exception properly may result in a dangling pointer (or an invalid pointer reference) in the application's execution. A variation of this test case is:

  - Generate a HD Photo file with the offset to the next (second) IFD pointing to the beginning of the first Image File Directory. This tests to see if the parser goes into an infinite loop by reading in the same IFD over and over again.

- Case 11: Generate a HD Photo file with the Image File Directory missing. This tests a parser's ability to reject all HD Photo files that do not have an IFD.

- Case 12: Generate a HD Photo file with the BITSTREAM_FORMAT syntax element set, even though the sample image has this element originally set to zero. This tests a parser's ability to handle a HD Photo file which advertises that it the HD Photo image is laid out in Frequency mode, even though the image itself was created in Spatial mode.

- Case 13: Generate a HD Photo file with the alpha channel flag syntax element set, even though the sample image has this element originally set to zero. This tests a parser's ability to handle a HD Photo file which advertises that it has an alpha channel present in its bitstream, even though the image itself does not have such a channel.

- Case 14: Generate a HD Photo file, while replacing all the bytes in the elementary HD Photo bitstream with random values. Together with Case 5, the purpose of this test case is to further test if the parser can recognize all possible forms that a valid HD Photo Bitstream can take.

- Case 15: Generate a HD Photo file with the PixelFormat IFD tag set to a different value (which may be a valid or invalid value) in place of the original value. This tests the parser to see if it can handle a HD Photo file which has a possibly corrupt value in the PixelFormat IFD tag.

- Case 16: Generate a HD Photo file and set the width, height, and the number of bytes of image data to random values in order to see if this results in a HD Photo file that is accepted by the parser: This test case fuzzes the field value of the ImageHeight, ImageWidth, and the ImageByteCount tags. These values should be validated by the parser when processing a HD Photo file.

- Case 17: Generate a HD Photo file with a single image that is X pixels high and Y pixels wide, such that X and Y are values where one dwarfs the other. In this test case, the Transformation tag should be set to generate a 90-degree clockwise rotation, but the same values for the ImageWidth and ImageHeight tags should be kept. As noted in the handtrace described in Chapter IV, the values for ImageWidth and ImageHeight should be updated if the value in the Transformation IFD tag is changed. This tests the image viewer application's ability to parse the Transformation IFD tag for all possible values.
- Case 18: Generate a HD Photo File where the WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements have incorrect values, or do not match the actual size of the image. This test case should also modify the ImageWidth and ImageHeight HD Photo IFD container tags so that they contain incorrect values as well. This tests how the parser validates the width and height of the HD Photo image file.
- Case 19: Generate a HD Photo File such that the tiling flag is set, even though the sample image does not use tiling. This tests how the image viewer application deals with an untiled HD Photo file that has the tiling flag set.
- Case 20: Generate a HD Photo File where the Image Plane Header uses a CLR_FMT value of 7, which is reserved. This tests how the parser reads in the syntax element CLR_FMT, which should not have a value of 7 for the current version of HD Photo.

With the test cases developed for both mutation-based fuzzing and generation-based fuzzing, the focus can now shift to how the fuzzing is implemented for testing HD Photo. To carry out this kind of testing, especially for generation and mutation-based fuzz testing, an existing generic file format fuzzer can be used and modified to suit our implementation needs.

The next chapter describes the image viewer applications to be tested using the fuzzing toolset and why these applications were chosen as the candidates for fuzz testing.

At the time of this writing, these viewer applications described in the next chapter are capable of handling and rendering image files saved in the HD Photo file format. The next chapter will also cover the steps taken in the successful setup for the experimental procedures performed in conducting fuzz testing against each of the chosen image viewer applications using HD Photo.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENTAL SETUP AND JUSTIFICATION OF SELECTED TOOLSETS AND IMAGE VIEWER APPLICATIONS

A. SELECTION OF TOOLSET FOR TESTING HD PHOTO FILES

For this thesis, the toolset used to test image viewer applications via the HD Photo file format will be a general file format fuzzer. In addition to a high benefit-to-cost ratio in terms of time and resources compared to other common software testing techniques, the advantages of using a general file format fuzzer is that the fuzzing toolset built in testing the HD Photo file format can be applied from one image viewer application to another without having to make any or little changes to the toolset itself. A quality general file format fuzzer can also be customized and modified without too much difficulty to incorporate both mutation-based fuzzing and generation-based fuzzing.

Most of the existing fuzzing frameworks used today are built for testing network protocols, particularly with web browsers and web servers as the primary targets. In addition, since a lot of the fuzzing frameworks are intended for use with the UNIX operating systems, many of them are either simply not suited for use with the Windows platforms or require a lot of modification to be compatible with the Windows operating system. Thus, the fuzzing framework skeleton called “MiniFuzz” provided by Michael Howard was chosen as the foundation to build the fuzzer ultimately used for testing image viewer applications for security bugs via the HD Photo file format.

B. JUSTIFICATION OF CHOSEN HD PHOTO FUZZING TOOLSET

There are several reasons why the customized Visual Studio fuzzing toolset based on C++ code was chosen to carry out the fuzzing of the HD Photo file format. The main reason is that it is very user-friendly with respect to the Windows XP and the Windows Vista platforms, since these were the operating systems in which the security testing was performed. The image viewer applications chosen for security testing (discussed later in this chapter) are all run on the Windows operating system. Since it is natively Windows-based, MiniFuzz can easily launch Windows-based applications for fuzz testing.

Another reason why the MiniFuzz framework was chosen because it is written in C++ source code and compiled using the Visual Studio .NET 2005 integrated development environment. Under the Visual Studio development environment, management of multiple C++ header files and source files is made convenient. Since the MiniFuzz framework consists of several C++ header files and source files, making frequent changes to the framework was easier to do during the coding phase of this thesis. This cut down significantly in the amount of time required for coding, compiling, and debugging.

The MiniFuzz fuzzing framework, which was customized into a fuzzing toolset and for this thesis was modified to conduct HD Photo fuzz testing, also exhibits a great deal of flexibility. With this toolset, the test cases developed in the previous chapter for mutation-based fuzzing can easily be written and implemented for both “dumb fuzzing” and “smart fuzzing”. Thus, the framework can be easily filled in with the fuzzing test cases we wish to carry out for each of image viewer applications to be tested. The “MiniFuzz” skeleton can be extended to perform generation-based fuzzing as well. If other test cases need to be tested in the future, they can easily be added to the template of the fuzzing toolset.

C. SELECTION OF IMAGE VIEWER APPLICATIONS TESTED USING THE HD PHOTO FILE FORMAT

At the time of this writing, there were only a select handful of image viewer applications that were able to open and display HD Photo image files. One of them, of course is the default application for handling HD Photo image files in Windows Vista, called Windows Photo Gallery. Originally called Windows Fax and Picture Viewer in Windows XP, the default image viewer for Windows Vista was renamed to Windows Photo Gallery and serves as a tool for managing, editing, and tagging digital photos [Wikipedia-WPG]. Since this application uses the Windows Imaging Component framework, which is part of the .NET Framework 3.0 software component natively supported by Windows Vista, it gives Windows Photo Gallery the ability to support the HD Photo image file format.

The second image viewer application tested against the HD Photo fuzzer is Paint.NET. Paint.NET is an open-source image and photo editing software that runs on the Windows operating system, and can be downloaded without charge at the Paint.NET home site http://www.getpaint.net. Although this application was not originally written to support the HD Photo format, a plug-in made available by one of the lead developers of Paint.NET allows a user to open a HD Photo image file using this application. As of April 2007, only versions 3.10 and beyond of Paint.NET are able to open HD Photo image files [Paintdotnet 2007]. In addition to the HD Photo plug-in, Paint.NET requires the installation of .NET Framework 3.0. Thus, it is possible for someone to open an HD Photo image file in Windows XP using Paint.NET (version 3.10 or later) with .NET Framework 3.0 installed.

The third image viewer application being tested is XnView. Created by Pierre-e Gougelet and currently maintained by a team of developers, XnView is a utility for viewing, converting, and manipulating graphic files, and it can be run on multiple operating systems (including several versions of the Windows and Linux platforms). A copy of the XnView image viewer application is available for free at the official XnView site, http://www.xnview.com. A format plug-in to make XnView read and write to HD Photo image files is available in the "Download" section of the XnView site, under the "Plugins" subsection page. Unlike the Paint.NET application, XnView does not require .NET Framework 3.0 in order to be compatible with the HD Photo format.

D. JUSTIFICATION OF CHOSEN IMAGE VIEWER APPLICATIONS

The image viewer applications described in the previous section have all been selected for a couple of reasons. The primary reason is that these applications are part of only a select handful of image viewer applications available at the time of this writing that are compatible with HD Photo. As of 2007, not many applications exist that are able to open and display HD Photo image files. Thus, there aren't very many alternatives to choose from to use for HD Photo fuzz testing. The other reason is that these applications can be run on the Windows platform, which is the only platform that the HD Photo fuzzing toolset can be run. If the fuzzing toolset can't be run on the same platform as the target application, then it is obvious that no fuzz testing can be done to the application using that fuzzing toolset.

E. EXPERIMENTAL SETUP

The setup of the experiments conducted in HD Photo fuzz testing is rather simple. A working copy of the Windows Vista (Business Edition) operating system, along with Visual Studio .NET 2005, was installed on a workstation to be used for fuzz testing Windows Photo Gallery. The other image viewer applications, XnView and Paint.NET, were then installed on the other machine, which uses the Windows XP operating system. The Windows XP machine was also provided with an installation of .NET Framework 3.0, which is a requirement of the Paint.NET application in order to open HD Photo files. As of July 2007, .NET Framework 3.0 is available for download from Microsoft at http://www.microsoft.com/downloads/details.aspx?FamilyId=10CC340B-F857-4A14-83F5-25634C3BF043, free of charge.

Some additional steps were taken to have Paint.NET and XnView open HD Photo files. To meet the other requirement of making Paint.NET compatible with HD Photo, the HD Photo beta plug-in package had to be downloaded at the official Paint.NET site; it can be found at http://www.getpaint.net/files/zip/HDPhotoBETA.0.3.zip. The package was then unzipped to the FileTypes folder in the directory where Paint.NET was installed. For example, the Windows XP workstation had Paint.NET installed at the directory "C:\Program Files\Paint.NET", so the HD Photo plug-in was copied to the "C:\Program Files\Paint.NET\FileTypes" folder. To make XnView compatible with HD Photo, the HD Photo plug-in from the XnView official site was downloaded from the hyperlink http://www.xnview.com/download/plugins/wmp.zip. In a manner very similar to the steps taken for Paint.NET, this file was then unzipped to the "PlugIns" directory contained within the directory where XnView was installed. For the Windows XP machine, the plug-in was copied to the "C:\Program Files\XnView\PlugIns" folder.

Once the above steps were accomplished, the testing implementation of HD Photo file fuzzing can now begin with respect to the chosen image viewer applications. The next chapter covers how the HD Photo file fuzzing was implemented and conducted, as well as the results of the fuzz testing.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. TESTING IMPLEMENTATION AND RESULTS ANALYSIS

A. TESTING IMPLEMENTATION AND PROCEDURE

Once the experimental setup phase described in the previous chapter was completed, the fuzzing toolset described in Chapter 5 was modified to accommodate both mutation-based fuzzing and generation-based fuzzing specifically for HD Photo image viewer applications. Code was added to the fuzzing toolset to carry out the test cases for both "dumb fuzzing" and "smart fuzzing" for mutation-based fuzzing and the described generation-based fuzzing test cases in Chapter 5. Once the code was written for these test cases, the next step was figuring out how to implement the combination of the two types of fuzz testing in the HD Photo fuzzing toolset, since it is desired to accommodate both fuzzing approaches.

The general operation of the HD Photo fuzzing toolset is shown in Figure 7 as a flow diagram. The fuzzer is first given a target image viewer application and a directory containing a set of valid HD Photo image files. At the start of an iteration, the fuzzer selects at random whether to do generation-based fuzzing or mutation-based fuzzing. If it selects generation-based fuzzing, the fuzzer will generate a semi-valid HD Photo file based on at least one of the generation-based fuzzing test cases (also selected at random) described in Chapter 4 of this document. If it selects mutation-based fuzzing, the fuzzer will select at random one of the HD Photo files in the provided file directory, which is then mutated in some fashion (based on which of the mutation-based fuzzing test cases, described in Chapter 4 of this document, is randomly selected by the fuzzer). Once the fuzzed HD Photo file is produced through either of the two fuzzing approaches, the HD Photo fuzzer then spawns an instance of the target application and observes how it consumes the fuzzed HD Photo file. If the malformed HD Photo file causes an exception, the fuzzer saves that file in a folder created inside the directory that contains the set of valid HD Photo files. The fuzzing toolset names this folder "badfiles" and it contains any fuzzed HD Photo file created by the fuzzer that caused an exception in the target application. Thus, any fuzzed file which successfully causes the target application to crash can be saved for further observation. Otherwise, the fuzzer deletes the file.

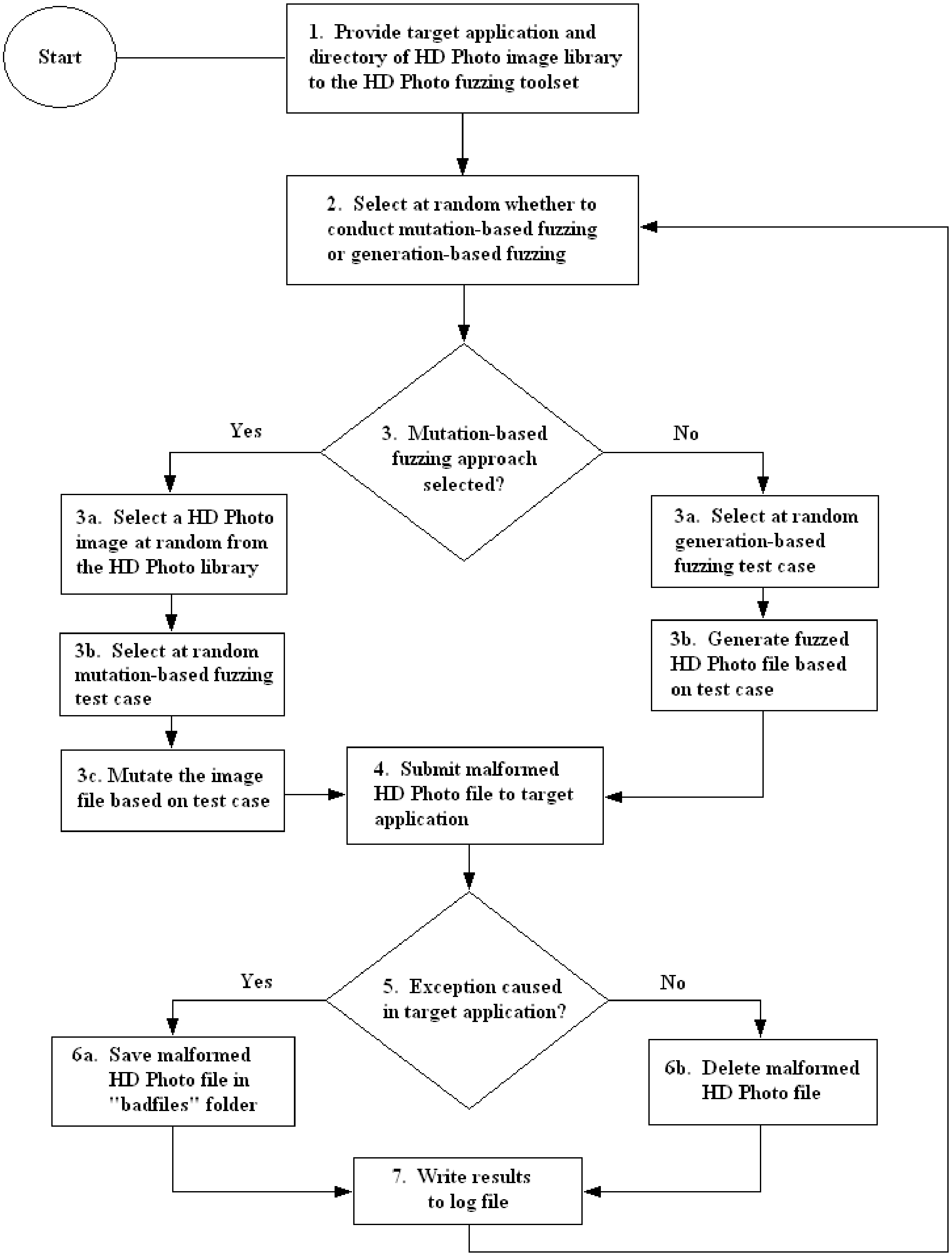

Start
1. Provide target application and directory of HD Photo image library to the HD Photo fuzzing toolset
2. Select at random whether to conduct mutation-based fuzzing or generation-based fuzzing
3. Mutation-based fuzzing approach selected?
Yes
No
3a. Select a HD Photo image at random from the HD Photo library
3b. Select at random mutation-based fuzzing test case
3c. Mutate the image file based on test case
3a. Select at random generation-based fuzzing test case
3b. Generate fuzzed HD Photo file based on test case
4. Submit malformed HD Photo file to target application
5. Exception caused in target application?
Yes
No
6a. Save malformed HD Photo file in "badfiles" folder
6b. Delete malformed HD Photo file
7. Write results to log file

Figure 7. The flow diagram of the HD Photo file fuzzing toolset used in testing the chosen image viewer applications Windows Photo Gallery, Paint.NET, and XnView.

The HD Photo fuzzing toolset is capable of monitoring for exceptions because it is a mini-debugger itself through the use of debugging APIs (or application program interfaces) [Howard 2006]. In either case (whether or not the fuzzed file succeeds in crashing the target application), the results are written to a log file by the fuzzer, indicating the fuzzed file that was created and the time of creation. The process is then repeated until the user of the fuzzing toolset wishes to stop the testing.

Using the MiniFuzz framework as its foundation, the HD Photo fuzzing toolset consists of several C++ header and source files and was compiled as a project in the Visual Studio 2005. (The C++ source code files which make up the fuzzing toolset can be found in Appendix B of this document.) The result is an executable called "MiniFuzz" which takes in two arguments: the name of the image viewer application to be tested, and the name of the directory that contains the library of valid HD Photo files to be used in mutation-based fuzzing. To run the fuzzing toolset using the command line utility in Windows, the user simply has to enter in the following command:

```
MiniFuzz <target_directory> <target_application>
```

This runs the fuzzing toolset against the chosen target application using the HD Photo files in the target directory. For example, if the desired target application is XnView and the HD Photo files used in fuzz testing are stored in the directory "C:\HDPhoto-Library" then the command would be:

```
MiniFuzz C:\HDPhoto-Library "C:\Program Files\XnView\xnview.exe"
```

In order to provide broad coverage for fuzz testing, a small library of valid HD Photo image files was assembled by collecting an assortment of image files from various online sources. This was necessary because there are very few HD Photo image files available online at the time of this writing, mainly due to HD Photo being a fairly new format and HD Photo not being compatible with two of the most popular web browser applications: Microsoft's Internet Explorer and Mozilla Firefox.

The library of HD Photo image files had to be created using images from the Internet originally in JPEG format. The Geekpedia conversion tool described in Chapter

4 of this thesis was then used to convert these images from its native file format to HD Photo. The images found online that were converted to HD Photo are as follows:

- The “msn-logo.jpg” image from the site “Canada’s Michael Smith Genome Sciences Centre” at http://www.bcgsc.ca/project/bomge/sockeye/images/msn-logo.JPG/view
- The test images of a factory office, at different resolutions and quality settings, from the site http://www.dnull.com/jtest/ saved as “small-15.jpg”, “small-35.jpg”, “small-75.jpg”, “medium-15.jpg”, “medium-35.jpg”, “medium-75.jpg”, “large-15.jpg”, “large-35.jpg”, and “large.jpg”.
- Standard test images often used for conducting classical and modern image processing testing, made available by the Signal and Image Processing Institute at the University of Southern California, and reproduced as PNG images at the Wikipedia website. These images can be found at the “Standard test image” article at http://en.wikipedia.org/wiki/Standard_test_image, under the section “Gallery of common test images”. These images are:
    - “Lenna.png”
    - “Baboon.png”
    - “Tiffany24.png”
    - “Peppers.png”
    - “Peppers2.png”
    - “House24.png”
    - “Sailboat24.png”
    - “Masuda1.png”
    - “Masuda2.png”
    - “Barbara24.png”
    - “Zelda24.png”
    - “Mouse.png”

- “Goldhill24.png”
- “Boats24.png”
- “Airplane24.png”

These images were converted to HD Photo (using the code in Appendix A) and resaved under the same filename, except with the extension being “.wdp” instead of “.jpg” or “.png”. These images served as the library of HD Photo files used in the mutation-based fuzzing part of testing the HD Photo image viewer applications.

The fuzzing toolset was run against each of the three chosen image viewer applications, using the previously described combined approach of mutation-based fuzzing and generation-based fuzzing. With all the test cases incorporated into the fuzzing tool, the tool was then run against each application for a time period of 24 hours, or a single day. This is to ensure that the fuzzer tests as much as possible the many variations that exist for each of the test cases developed for both types of fuzzing.

B. RESULTS AND ANALYSIS

After running the HD Photo fuzzing toolset against each of the select image viewer applications for a period of 24 hours, it was discovered that for all three test runs, all of the fuzzed HD Photo files created by the fuzzer were either accepted by the target application or rejected outright. Not a single malformed HD Photo file was developed by the HD Photo fuzzing toolset that could cause an exception in any of the three target applications that is worth investigating. The test runs were repeated again for each application, but the test results obtained were the same as the original: no exceptions were noted in the log files created by the fuzzing toolset, and no “bad” HD Photo files were saved in the “badfiles” folder.

To ensure that the cause of not finding any security bugs in any of the target applications was the fuzzing toolset itself, the mechanism that deletes all the fuzzed files that failed to cause the application to crash was turned off. To check the test cases generated for both mutation fuzzing and generation fuzzing, the hex editor program XVI32 (first described in Chapter 4 of this document) was used to see if the fuzzed files were created correctly by the fuzzing toolset. With the deletion mechanism of unsuccessful fuzzed HD Photo files turned off in the fuzzing toolset, it was confirmed that the malformed files created by the fuzzer were correctly developed according to specification.

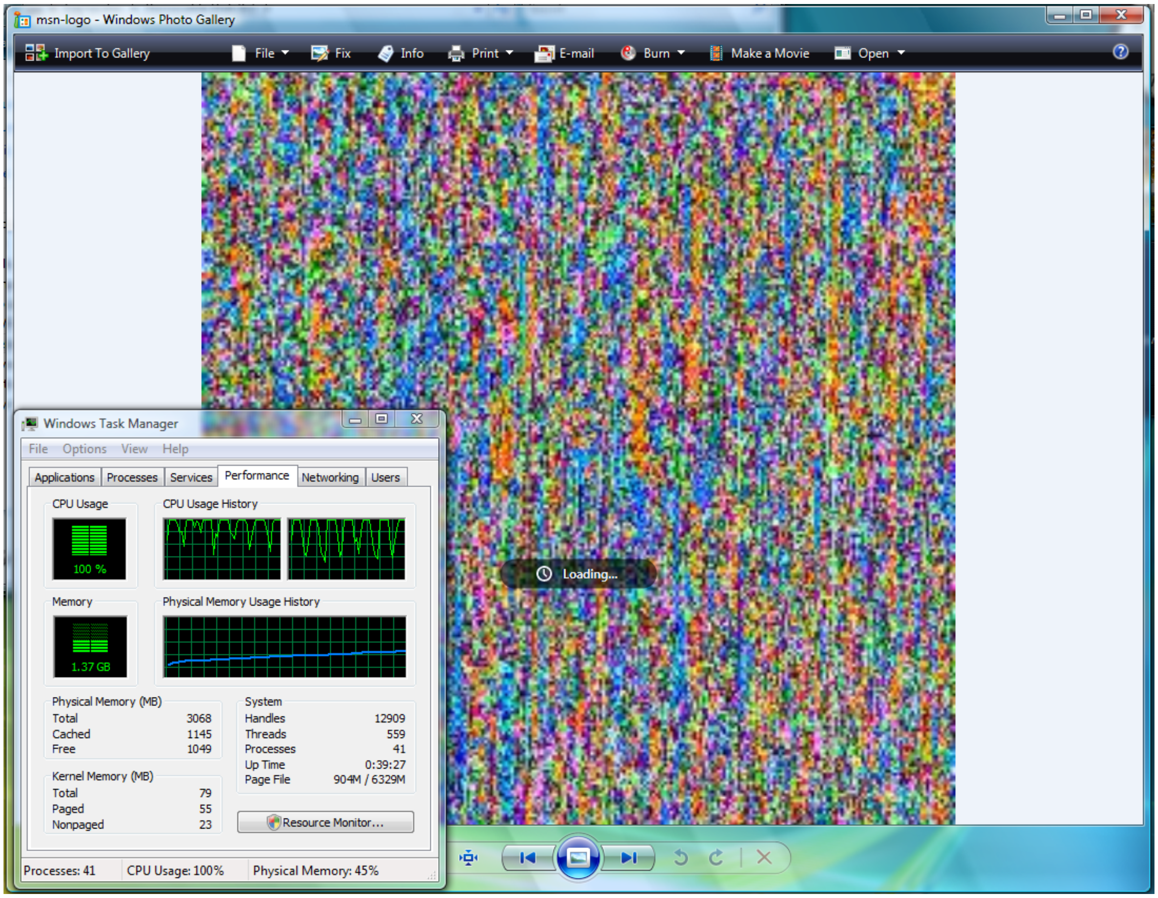
Multiple test runs were then conducted against each of the target applications in the hope of finding any security bugs. Modifications were made for some of the test runs, such as temporarily turning off generation-based fuzzing in the fuzzer so that only mutation-based fuzzing was conducted (and vice versa) and lengthening the period of time allowed for the fuzzer to run. Despite such measures, efforts to cause the target parser to crash were unsuccessful on each run.

As a last resort, the fuzzer toolset was run once again, this time with the deletion of failed malformed files turned off. Many of these malformed HD Photo files created by the fuzzing toolset were then manually opened with each of the chosen target image viewer applications. Despite hours spent opening each of the fuzzed files by hand for each of the target applications, the results were the same: the target application either rejected the fuzzed file or it accepted the file if it were actually legitimately valid.

After going over the results (or lack thereof in finding any security bugs), it was decided that it would be best to take another look at the HD Photo file format specification. One important observation that was noted about HD Photo is that in the HD Photo file format, the range of values for vital fields such as the width and height of the image (determined by the syntax elements WIDTH_MINUS1 and HEIGHT_MINUS1 in the HD Photo Bitstream) are set such that the possible values assigned to these fields are never below the absolute minimum value or above the absolute maximum value, since these syntax elements are treated as unsigned integers. For the lowest possible value that can be set for either syntax element (which is zero), the parser (i.e., the image viewer application) interprets this as a value of one, since it automatically increments by one the provided value for WIDTH_MINUS1 or HEIGHT_MINUS1. This prevents the situation of setting these elements to a zero or negative value. Having these syntax elements as unsigned integers helps to avoid the pitfalls of signed integer overflow, which often leads to unexpected behavior, or (even worse) a vulnerability that can be exploited). This is good programming practice.

HD Photo image viewer applications would most likely store the image data of an HD Photo file through the allocation of buffers, so corrupting the image’s height and width values would have been one of the best ways to attack this file format. Since the size of the buffers allocated by the target application would likely depend on the advertised height and width of the HD Photo image, it would be ideal to attempt the trigger of an integer overflow or buffer overflow by fuzzing the WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements. This is especially true for Windows Vista’s image viewer application, which exhibited somewhat strange behavior when opening one of the malformed files created by the fuzzer.

For Windows Photo Gallery in Windows Vista Business Edition (out of the box with no security updates installed), one of the closest times that a malformed file came to causing that application to deviate from its normal behavior was when the fuzzer took the “msn-logo.wdp” image file and set each of the WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements in the HD Photo Bitstream to the hexadecimal value 0xBFFF, even though the ImageWidth and ImageHeight IFD container tags advertised values of 29 pixels and 28 pixels respectively. When opened by Windows Photo Gallery, the image viewer application took a very long time to process the image file. After several hours, Windows Photo Gallery eventually finished rendering the HD Photo file, although the end result was an image that looked like gobbledygook. While it was processing the image, the CPU usage and the physical memory usage spiked considerably, as seen in Figure 8. In practice, if opening a file using an application results in a memory spike or CPU usage spike, this “would indicate that malicious data could cause a denial of service” for this application [Weinstein 2007]. Seeing a memory spike or CPU usage spike are two classic cases that vulnerability researchers look for in order to see if fuzzing had made an impact on the target application [Weinstein 2007]. For Windows Photo Gallery, handling the malformed HD Photo file resulted in 100 percent CPU usage and a significant spike (45 percent) in physical memory usage, which continued to slowly but continually increase. Other high hexadecimal values were used in place of 0xBFFF for both WIDTH_MINUS1 and HEIGHT_MINUS1 (such as 0xFFFE and 0x7FFF), which cause similar results when Windows Photo Gallery attempts to open the malformed file.



Physical Memory (MB)	
Total	3068
Cached	1145
Free	1049

Kernel Memory (MB)	
Total	79
Paged	55
Nonpaged	23

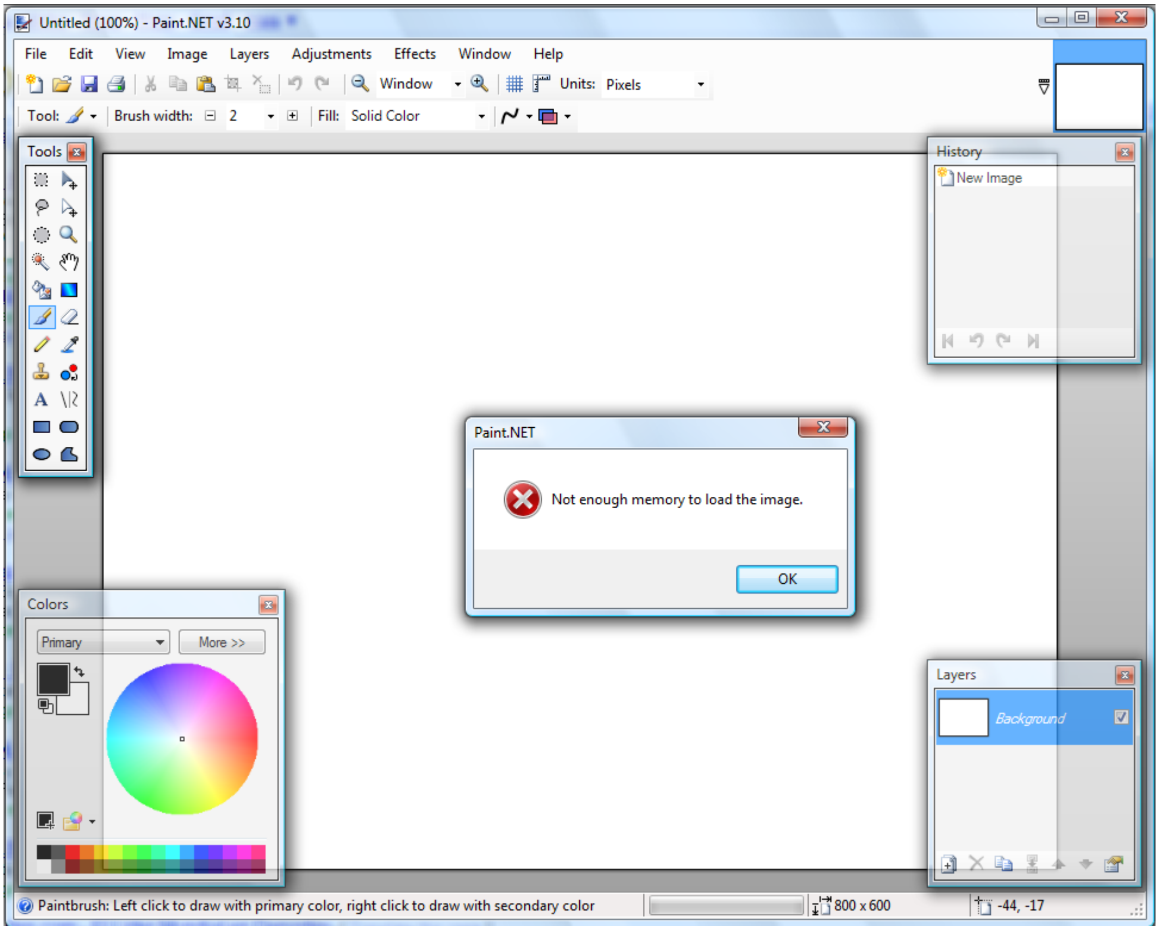
System	
Handles	12909
Threads	559
Processes	41
Up Time	0:39:27
Page File	904M / 6329M

Processes: 41 | CPU Usage: 100% | Physical Memory: 45%

Figure 8. Screenshot of Windows Photo Gallery handling malformed “msn-logo.wdp” file in Windows Vista Business, with Windows Task Manager showing the current CPU usage and memory. The HD Photo file “msn-logo.wdp” is modified with WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements set to the hexadecimal value 0xBFFF.

Although this was technically an invalid HD Photo file, Windows Photo Gallery handling the malformed “msn-log.wdp” file took up a lot of resources on Windows Vista. However, it did not cause the application to hang, since the user still has control of Windows Photo Gallery, such as selecting the option to view the next picture in the current directory. What this observation tells us about Windows Photo Gallery is that it is consuming far more resources that usual to process the malformed HD Photo file, which causes the operating system to slow down to a crawl. This is due to the amount of memory that Windows Photo Gallery has allocated in anticipation of loading this image file because of the altered values for the WIDTH_MINUS1 and HEIGHT_MINUS1 syntax elements. Replacing the actual values of both the image width and height syntax elements of a HD Photo file with a high enough hexadecimal value seems to be the main culprit for both the memory spike and CPU spike when Windows Photo Gallery attempts to open that file. Even though this does not result in a complete denial of service attack, the ability to craft a HD Photo file that (when opened) results in Windows Photo Gallery causing Windows Vista to run slowly is something that should be looked at by testers in the near future.

This malformed file was then opened using the other applications, Paint.NET and XnView. Upon opening "msn-logo.wdp" and observing the Paint.NET application (this time running on Windows Vista Business, to compare performance with Windows Photo Gallery) after thirty to forty seconds, the application displayed an error message, as seen below in Figure 9. The error message reads "Not enough memory to load the image"; which corroborates the earlier claim that Windows Photo Gallery behaved the way it did because it allocated a massive amount of memory to load the malformed file. While it was trying to load the image, Paint.NET also consumed 100 percent of the CPU and a significant amount of memory in attempting to do so. However, Paint.NET released the CPU and memory resources when it stopped its attempt to load the malformed file after thirty to forty seconds, and informed the user of the error. Thus, Paint.NET was able to abort the process of opening the malformed file cleanly. In a similar manner, upon attempting to open "msn-logo.wdp", XnView pops up a window that indicates to the user that the program needs to quit, so it exits cleanly.



Untitled (100%) - Paint.NET v3.10

Units: Pixels

Brush width: 2

Fill: Solid Color

Paint.NET

Not enough memory to load the image.

Paintbrush: Left click to draw with primary color, right click to draw with secondary color

800 x 600

-44, -17

Figure 9. Screenshot of Paint.NET showing an error message to the user, after attempting to open the malformed “msn-logo.wdp” HD Photo file in Windows Vista Business; “msn-logo.wdp” is the same file as the one previously mentioned in Figure 8.

For the most part, the results indicate that the programmers who wrote the code for the tested image viewer applications seem to have followed good code practices in parsing file formats. For one, they went through great lengths to check the validity of the files being parsed. One of these practices is listed in documents such as [HDPhotoFeatureSpec], where the implementers are cautioned to perform a data pointer check for structures such as pointers to IFD entries in HD Photo. It avoids having possible pointer error bugs by warning implementers never to have two pointers that reference the same location in memory. Another good practice championed by [HDPhotoFeatureSpec] is the ability to read a certain file format unambiguously. This is implemented in HD Photo through the use of the PixelFormat IFD tag. Attempts to fuzz the value provided in the PixelFormat IFD entry either resulting in the fuzzed file being rejected by the target parser, or the parser being able to render the image successfully.

The security likely rests on the implementation of HD Photo (such as managed code such as .NET Framework 3.0) and on the image viewer applications that handle the image. This includes performing the necessary input validation routines to correctly identify valid HD Photo image files and reject invalid (and possibly malformed) images. The managed code that the Windows operating system uses to handle and render HD Photo, .NET Framework 3.0, uses a virtual machine to execute code, instead of passing it to the kernel itself. This allows for checking to ensure that malicious code is not executed on behalf of the current user.

What cannot be concluded is whether the applications tested in this thesis are completely secure with respect to all possible variations of HD Photo image files. The results can only indicate that the applications are very resistant to easy-to-find security bugs often discovered by fuzzers. Fuzzing is used to find what is referred to as the low-hanging fruit of the vulnerabilities. Sutton et. al. liken vulnerabilities as fish and a vulnerability researcher as a fisherman; they explain that fuzzing is “highly effective at capturing the ‘easy’ fish” while the art of reverse engineering an application “in search of a vulnerability may be referred to as ‘deep sea fishing’” [Sutton 2007]. While fuzzing has a knack for finding security bugs in any application, reverse engineering these image viewer applications might be the only way to discover what vulnerabilities may exist in them.

While the fuzzing toolset not being able to find vulnerabilities in the chosen HD Photo image viewer applications does not completely guarantee their security, it strongly indicates that each application’s ability to parse HD Photo files is relatively strong in security and requires a malicious user to go long lengths to find such a vulnerability relating to HD Photo when targeting these applications.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION

A. CONCLUSION

The failure to find any security bugs in the sample of image viewer applications that accept HD Photo data as input, despite exhaustive fuzz testing conducted using the HD Photo fuzzing toolset, indicate at the very least that there does not exist any “low-hanging fruit” in terms of security vulnerabilities associated with using the file format. Despite not being able to break the security of any of the chosen applications in this thesis, the only way to be completely sure of the absence of security bugs or vulnerabilities is a formal mathematical proof of the application in handling of HD Photo. Testing on the other hand can reveal software faults, but not indicate the complete absence of software bugs except under exhaustive testing which is impractical for anything but the most trivial software systems.

The lesson learned in this thesis is that in general, any form of data that can be altered by any user should never be assumed to be automatically valid. This means that binary parsers, especially HD Photo image viewer applications, should take the steps necessary to validate all fields contained within a binary file (such as a HD Photo file). Because it is unsafe to assume that all data provided by a given binary file is valid, programmers should take meticulous care in writing parsers that can successfully handle as many variations of a binary of a given file format as possible, without deviating from its intended specification. One of the best ways to accomplish this is to keep the source code simple enough such that it will “accept only valid input [and] deny all variants” [Padilla 2005]. This not only enhances security, but it also saves the heartache and trouble of having to patch the application in order to fix possible bugs that may be reported in the not-too-distant future. Although the applications tested in this thesis seem to demonstrate strong security concerning the HD Photo file format, care should be taken when handling malformed HD Photo files. This includes HD Photo files in which the data provided by the ImageWidth and ImageHeight IFD tags do not match the data given by the HD Photo Bitstream syntax elements WIDTH_MINUS1 and HEIGHT_MINUS1, in which case the data should be immediately rejected by the image viewer applications.

Although it is necessary to ensure that image viewer applications will correctly handle HD Photo images, the risk of possible malicious use of HD Photo itself will not be a top priority until the web browsers used today are updated to render and display images of the HD Photo file format. Once the most commonly used web browsers of today are updated to handle, render, and show image files of the HD Photo image format, that is when we should worry about the proliferation of HD Photo online, since the Internet is the most probable venue through which malicious non-executable files (especially crafted malformed images) can be disseminated widely in an expeditious manner.

B. RECOMMENDATIONS AND SUGGESTIONS FOR FUTURE WORK

There are some areas of research that should be given near-term attention. One of these is to further test the security of image viewer applications compatible with HD Photo. This would also include applications using future versions of HD Photo. As of July 2007, the HD Photo file format has been submitted for review to the Joint Expert Photographers Group as an image standard. Should the file format be successfully standardized, this would pave the way for others to update their image viewer applications to become compatible with HD Photo. It would also create more incentive for current web browsers to be updated for this file format. Thus, applications that will be updated (such as web browsers) to accommodate this file format in the future will need to be tested as well.

Another research avenue is the detection of steganography inside HD Photo image files. If image files of the HD Photo format are to be used in multilevel security systems (MLS), steganalysis methods and techniques will need to be developed to prevent users from embedding data at a higher classification level (i.e., the payload) in an HD Photo image (i.e., the carrier) of a lower classification level inside HD Photo image files which would violate no-write-down security policies.

If there is off-nominal behavior (such as security bugs or vulnerabilities) discovered in any of the existing viewer applications compatible with HD Photo, the next step would be determining whether exploitation is possible with these applications. Good references for determining exploitation are [Harris 2005] and [Wysopal 2007].

APPENDIX A. HD PHOTO IMAGE CONVERSION SOURCE CODE LISTING

```
/************************************************************
 * Geekpedia's main code listing for converting common image
 * graphics formats to HD Photo.
 * It is available at the Geekpedia website in the form of
 * a .zip archive file, with the link included below.
 *
 * Provided/uploaded by Andrei Pociu, May 27 2007
 * http://www.geekpedia.com/tutorial208_Converting-Graphics-to-HD-
Photo-(WDP-or-HDP).html
 *
 * Installation steps for HD Photo conversion software,
 * assuming the machine is running Windows with
 * Visual Studio .NET 2005 and .NET Framework 3.0 installed:
 *
 * (1) Download the "HD Photo graphics conversion" project
 * for Visual Studio 2005 at:
 * http://www.geekpedia.com/Samples/ConvertToWdp/ConvertToWdp.zip
 * (2) Unzip the file to a directory of your choice.
 * (3) Open the project in Visual Studio 2005.
 * (4) Compile the project by selecting "Build Solution" from
 *     the "Build" menu (or press Ctrl+Shift+B as a shortcut)
 * (5) Run the project by selecting "Start Debugging" from the
 *     "Debug" menu (or press F5 as a shortcut).
 *
 *  When the source code is run successfully, a pop-up window
 *  appears with the message "It's as simple as that:" and a
 *  mouse-click button labeled "Convert".  When the "Convert"
 *  button is clicked, the user can select which image file
 *  (not in HD Photo format) that he/she wishes to convert,
 *  and the filename of the HD Photo equivalent of that image
 *  file.
 *  The second file will be the HD Photo version of
 *  the original image selected.
 *
 *  Note: To reproduce the 198-byte HD Photo image of a single
 *        black pixel (saved in Microsoft Paint as a 24-bit .bmp
 *        file) handtraced in this thesis, in the original code
 *        the following line in the “FileToWmp” method
 *
 *           wbeFile.ImageQualityLevel = 0.9f;
 *
 *        should be changed to
 *
 *           wbeFile.ImageQualityLevel = 1.0f;
 *
*************************************************************/


using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Text;
using System.Windows.Forms;
using System.Windows.Media.Imaging;

namespace ConvertToWmp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnConvert_Click(object sender, EventArgs e)
        {
            // Prompt to open the file
            if (openPicture.ShowDialog() == DialogResult.OK)
            {
                // Prompt to save the file
                if (saveWdp.ShowDialog() == DialogResult.OK)
                {
                    // Call the method that does all the work
                    FileToWmp(openPicture.FileName, saveWdp.FileName);
                }
            }
        }

        public static void FileToWmp(string inFile, string outFile)
        {
            // Container for bitmap frames
            BitmapDecoder bdFile = null;
            // Read the source file into a FileStream object
            FileStream readFile = File.OpenRead(inFile);
            // Set the BitmapDecoder object from the source file
            bdFile = BitmapDecoder.Create(readFile,
BitmapCreateOptions.PreservePixelFormat, BitmapCacheOption.None);
            // Prepare the output file
            FileStream writeFile = File.OpenWrite(outFile);
            // All the magic done by WmpBitmapEncoder
            WmpBitmapEncoder wbeFile = new WmpBitmapEncoder();
            // Quality level set for lossless compression
            wbeFile.ImageQualityLevel = 1.0f;
            // Add the bitmap frame to the encoder object
            wbeFile.Frames.Add(bdFile.Frames[0]);
            // Write the output file
            wbeFile.Save(writeFile);
            writeFile.Close();
            readFile.Close();
        }
    }
}
```

APPENDIX B. HD PHOTO FUZZING TOOL SOURCE CODE LISTINGS (USING MINIFUZZ)

A. FUZZING TOOLSET C++ HEADER FILES

1. stdafx.h

```
/************************************************************
Filename:  stdafx.h
Original author:  Michael Howard

Auxiliary header file used by MiniFuzz
************************************************************/

#pragma once

#ifndef _M_IX86
#     error MiniFuzz has only been tested on Intel/AMD CPUs
#endif

#ifndef WINVER                // Allow use of features specific
                              // to Windows XP or later.
#define WINVER 0x0501         // Change this to the appropriate value
                              // to target other versions of Windows.
#endif

#ifndef _WIN32_WINNT          // Allow use of features specific to
                              // Windows XP or later.
#define _WIN32_WINNT 0x0501   // Change this to the appropriate value
                              // to target other versions of Windows.
#endif

#ifndef _WIN32_WINDOWS        // Allow use of features specific to
                              // Windows 98 or later.
#define _WIN32_WINDOWS 0x0501 // Change this to the appropriate
                              // value to target Windows Me or later.
#endif

#define WIN32_LEAN_AND_MEAN   // Exclude rarely-used stuff
                              // from Windows headers
#include <stdio.h>
#include <tchar.h>
#include "windows.h"
#include "stdlib.h"
#include "limits.h"
#include "time.h"
#include "string.h"
#include "winsock2.h" // for byte order functions
#include "limits.h"
#include "assert.h"
#include "sal.h"
#include "wchar.h"

#include "rand.h"
```

```
#include "iostream"
#include <iomanip>

#include "vector"
#include "string"
#include <algorithm>
```

2. rand.h

```
/****************************************************************
Filename:  rand.h
Original author:  Michael Howard

This code (along with rand.cpp) serves as a wrapper around
rand() that produces 32-bit numbers rather than 16-bit numbers,
so we can cover the entire 32-bit spectrum.
****************************************************************/

#define USE_SECURE_RANDOM_NUMBERS 0
unsigned int GetRand();
```

3. Log.h

```
/*************************************************************
Filename:  log.h
Original author:  Michael Howard

This code (along with "log.cpp") maintains all the logging
and tracing code for MiniFuzz.
*************************************************************/

using namespace std;

bool OpenLogFile(string sDir);
void CloseLogFile();

void ReportFuzzError(__in_z const char *szFilename,
                               string sExc, CONTEXT ctx);

void Log(__in_z const char *szMessage, __in_z const char *szArg);
void Log(__in_z const char *szMessage, __in_z const char *szArg,
            DWORD dwErr);
void Log(__in_z const char *szMessage, DWORD dwErr);
void Log(__in_z const char *szMessage);

void Trace(__in_z const char *szMessage, bool fCRLF);
void Trace(__in_z const char *szMessage);
```

4. Fuzz.h

```
/************************************************************
Filename:  fuzz.h
Original author:  Michael Howard
Modified by:  Clifford Juan

Header file for mutation-based fuzzing functions and
generation-based fuzzing functions used by fuzzing toolset.
************************************************************/

#define VERSION "0.25 ALPHA"

// Different kinds of mutation-based "dumb fuzzing"
const enum MALFORM_TYPES
{
   MALFORM_SNIP_FILE,               // Truncate the file at random
   MALFORM_FILL_RANDOM,             // Fill random portions of the
                                    // file with random data
   MALFORM_NUKE_SZ,                 // Remove null terminator from
                                    // embedded ASCII strings
   MALFORM_FLIP_HIGH_BITS,          // Flip random hi-bits
   MALFORM_XOR_BITS,                // Similar to FILL_RANDOM, but
                                    // using the "xor" operation
   MALFORM_INTERESTING_DATA,        // Sections of the file marked
                                    // for "smart fuzzing"
   MALFORM_FLIP_ADJACENT_BYTES,     // Randomly flip adjacent bytes
   MALFORM_LAST                     // This is just a sentinel
};

// Some of the possible interesting parts of a file
typedef const enum _INTERESTING_BIT_TYPES {
            IB_END      =     0,    // End of data marker
            IB_SIZE     =     1,
            IB_OFFSET   =     2,
            IB_COUNT    =     4};

typedef const enum _ENDIAN {ENDIAN_NONE,
                            ENDIAN_BIG,
                            ENDIAN_LITTLE};

// Interesting parts of a file worth looking at for
// mutation-based "smart fuzzing"
typedef struct _INTERESTING_BITS {
      DWORD cbOffset;   // Byte offset of this data
      DWORD cbSize;     // The size of the data in bytes
                        // (DWORD is 4 bytes)
      _INTERESTING_BIT_TYPES  eType;      // The type of data
      _ENDIAN     eEndian;   // The byte order or "endian-ness"
                             // (always little-endian for HD Photo)
      char *szComment;  // Documentation, in the form of a comment
} INTERESTING_BITS;

// Used to mark the end of an interesting data section
const INTERESTING_BITS gEndMarker = {0,0,IB_END,ENDIAN_NONE,NULL};
```

```
typedef struct _DATATYPES {
    const char *szExtension; // file extension, including
                             // the trailing dot "."
      INTERESTING_BITS *pBits;
} DATATYPES;

// Maximum number of malform operations on a file, chosen at random
const size_t MAX_MALFORMS = 7;

// Maximum file size that can be handled by this fuzzer,
// (2^32) - 1 bytes
const unsigned __int32 MAX_FILESIZE = (_UI32_MAX / 8);

// Mutation-based fuzzing functions, to carry out
// "dumb fuzzing" and "smart fuzzing"
bool MalformSnipFile(HANDLE hFile,__in DWORD *pdwSize);
bool MalformFillRandom(HANDLE hFile,DWORD dwSize);
bool MalformNukeString(HANDLE hFile,DWORD dwSize);
bool MalformHiBits(HANDLE hFile,DWORD dwSize);
bool MalformXorBits(HANDLE hFile,DWORD dwSize);
bool MalformInterestingData(HANDLE hFile,DWORD dwSize,
                              __in INTERESTING_BITS *pBits);
bool MalformFlipAdjacentBytes(HANDLE hFile,DWORD dwSize);

// Generation-based fuzzing functions, to generate
// semi-valid HD Photo files
bool MalformGenBasedResizeFile(HANDLE hFile,__in DWORD *pdwSize);
bool MalformGenBasedGenerateFile(HANDLE hFile,DWORD dwSize);

// "Spawn the target application" function
bool LaunchFile(__in_z const char *, __in_z const char *,
                        __in_z const char *,bool *);
```

B. FUZZING TOOLSET C++ SOURCE FILES

1. stdafx.cpp

```
/************************************************************
Filename:  stdafx.cpp
Original author:  Michael Howard

Auxiliary C++ file used by MiniFuzz
************************************************************/

#include "stdafx.h"
```

2. rand.cpp

```
/*****************************************************************
Filename:  rand.cpp
Original author:  Michael Howard

This code (along with rand.h) serves as a wrapper around rand()
that produces 32-bit numbers rather than 16-bit numbers, so we
can cover the entire 32-bit spectrum.
******************************************************************/

#include "stdafx.h"

#if USE_SECURE_RANDOM_NUMBERS == 0

// rand() only produces values from 0-32767
// and we need a full 32-bit spectrum
unsigned int GetRand() {
      unsigned int r[4];

      r[0] = rand() & 0xFF;
      r[1] = rand() & 0xFF;
      r[2] = rand() & 0xFF;
      r[3] = rand() & 0xFF;

      return      (r[0] << 24) +
                  (r[1] << 16) +
                  (r[2] << 8 ) +
                   r[3];
}

#else

unsigned int GetRand() {
      return rand_s();
}

#endif
```

3. LaunchExe.cpp

```
/************************************************************
Filename:  LaunchExe.cpp
Original author:  Michael Howard
Modified by:  Clifford Juan

This C++ source code file runs the target application with
the malformed file, and awaits debug events.
If a debug event occurs, the event is recorded to the
"minifuzz" log file.
************************************************************/

#include "stdafx.h"
#include "fuzz.h"
#include "log.h"

using namespace std;

// The length of time an app should run for us to consider
// the test a success (2 seconds)
const DWORD MAX_EXE_RUNTIME_MS  = 2000;

// Log interesting information if we get a debug event back
// from the debugged application
bool ReportFailure(DEBUG_EVENT *pdbg, const char *szFilename) {
      HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE,
                                                pdbg->dwThreadId);
      if (!hThread) {
            Log("OpenThread failed");
            return false;
      }

      string sExc;

      // Get information on what type of exception has occurred
      switch(pdbg->u.Exception.ExceptionRecord.ExceptionCode) {
            case EXCEPTION_ACCESS_VIOLATION :
                  sExc = "Access Violation";       break;
            case EXCEPTION_STACK_OVERFLOW:
                  sExc = "Stack Overflow";         break;
            case EXCEPTION_DATATYPE_MISALIGNMENT:
                  sExc = "Datatype misalignment";  break;
            case EXCEPTION_ARRAY_BOUNDS_EXCEEDED:
                  sExc = "Array Bounds Exceeded";  break;
            case EXCEPTION_FLT_DIVIDE_BY_ZERO:
                  sExc = "Float Div/0";            break;
            case EXCEPTION_INT_DIVIDE_BY_ZERO:
                  sExc = "Int Div/0";              break;
            case EXCEPTION_ILLEGAL_INSTRUCTION:
                  sExc = "Illegal Instruction";    break;
            case EXCEPTION_IN_PAGE_ERROR:
                  sExc = "In-page error";          break;
            case EXCEPTION_PRIV_INSTRUCTION:
                  sExc = "Privileged Instruction"; break;
            default :
                  sExc = "Unknown";                break;
      }

      CONTEXT ctx;
      memset(&ctx,0,sizeof CONTEXT);
      ctx.ContextFlags = CONTEXT_ALL;

      if (!GetThreadContext(hThread, &ctx)) {
            Log("GetThreadContext failed");
            return false;
      }

      // Call function to record event in log file
      ReportFuzzError(szFilename,sExc,ctx);

      return true;
}

// Spawn the executable to fuzz
bool LaunchFile(const char *szDir, const char *szExe,
                        const char *szFilename, bool *pfDeleteTempFile)
{

      if (pfDeleteTempFile == NULL && szDir == NULL || szExe == NULL
            || szFilename == NULL) {
            Log("Invalid args to LaunchFile");
            return false;
      }

      *pfDeleteTempFile = false;
      bool fError = false;

      PROCESS_INFORMATION pi;
      STARTUPINFO         si;
      memset(&pi,0,sizeof PROCESS_INFORMATION);
      memset(&si,0,sizeof STARTUPINFO);
      si.cb = sizeof STARTUPINFO;

      // Build up the cmd-line
      string sExe(szExe);
      sExe.append(" ");
      sExe.append(szFilename);

      BOOL fRet = CreateProcess(NULL,
            const_cast<LPSTR>(sExe.c_str()),
            NULL, NULL,
            FALSE,
            DEBUG_ONLY_THIS_PROCESS,
            NULL, NULL,
            &si, &pi);

      if (!fRet) {
            Log("Unable to launch process", szExe, GetLastError());
            return false;
      }

      DWORD dwStart = GetTickCount();
```

```
// Now wait for debug events from the new process
do {
   DEBUG_EVENT  dbg;
   if (WaitForDebugEvent(&dbg, 200)) {

      // Get copies of all loaded DLL file handles
      // - we don't need them, so close them
      if (dbg.dwDebugEventCode == LOAD_DLL_DEBUG_EVENT) {
         CloseHandle(dbg.u.LoadDll.hFile);
         ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId,
            DBG_CONTINUE);
         continue;
      }

      // Get a copy of the loaded exe file handle
      // - we don't need it, so close it
      if (dbg.dwDebugEventCode == CREATE_PROCESS_DEBUG_EVENT) {
         CloseHandle(dbg.u.CreateProcessInfo.hFile);
         ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId,
            DBG_CONTINUE);
         continue;
      }

      // At this point we only care about real debug events
      if (dbg.dwDebugEventCode != EXCEPTION_DEBUG_EVENT) {
         ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId,
            DBG_CONTINUE);
         continue;
      }

      // An exception has occurred
      // NOTE: This code can catch *ALL* exceptions, including
      // first-chance exceptions.
      // Catching and logging a first-chance exception does
      // *NOT* mean there is a security bug in your code.
      // Don't go filing bugs on first-chance exceptions
      // unless it's a real bug!
      switch (dbg.u.Exception.ExceptionRecord.ExceptionCode) {
         case EXCEPTION_ACCESS_VIOLATION :
         case EXCEPTION_STACK_OVERFLOW:
         case EXCEPTION_DATATYPE_MISALIGNMENT:
         case EXCEPTION_ARRAY_BOUNDS_EXCEEDED :
         case EXCEPTION_FLT_DIVIDE_BY_ZERO:
         case EXCEPTION_INT_DIVIDE_BY_ZERO:
         case EXCEPTION_ILLEGAL_INSTRUCTION:
         case EXCEPTION_IN_PAGE_ERROR:
         case EXCEPTION_PRIV_INSTRUCTION:

            // DO NOT HANDLE FIRST CHANCE EXCEPTIONS!
            // If you want to handle them,
            // then change this code to:
            //if (dbg.u.Exception.dwFirstChance) {

            //if (dbg.u.Exception.dwFirstChance) {
            if (!dbg.u.Exception.dwFirstChance) {
                  ReportFailure(&dbg, szFilename);
```

```
                        fError = true;
                  }
                  break;


               default:
                  ContinueDebugEvent(dbg.dwProcessId,
                     dbg.dwThreadId, DBG_CONTINUE);
                  break;
               }
            } else {
                 break;
            }

      } while((GetTickCount() - dwStart) < MAX_EXE_RUNTIME_MS);

      DebugActiveProcessStop(pi.dwProcessId);
      if (!TerminateProcess(pi.hProcess,1))
            Log("Unable to kill process", szExe, GetLastError());

      if (pi.hThread)
            CloseHandle(pi.hThread);

      if (pi.hProcess)
            CloseHandle(pi.hProcess);

      if (pfDeleteTempFile && !fError)
            *pfDeleteTempFile = true;

      return fError;
}
```

4. Log.cpp

```
/*************************************************************
Filename:  log.cpp
Original author:  Michael Howard
Modified by:  Clifford Juan

This code maintains all the logging and tracing code for
MiniFuzz.  It writes all events to a log file (each malformed
file consumed by the target application, along with the
timestamp when it was created, and interesting debug events
caused by malformed files) named "minifuzz.log".
*************************************************************/

#include "stdafx.h"
#include "fuzz.h"
#include "log.h"
#include <fstream>

using namespace std;

static ofstream *g_pOut = NULL;
```

```
// Get the system time for the purpose of logging timestamps
static void GetLogTime(string *psTime) {
      SYSTEMTIME tm;
      char t[64];

      GetSystemTime(&tm);
      sprintf_s(t,_countof(t),
                        "%d-%d-%d %d:%02d:%02d",
                        tm.wYear,
                        tm.wMonth,
                        tm.wDay,
                        tm.wHour,
                        tm.wMinute,
                        tm.wSecond);
      psTime->assign(t);
}

// Make sure we can write to log file
bool OpenLogFile(string sDir) {
      assert(g_pOut == NULL);

      if (g_pOut != NULL)
            return true;

      sDir.append("\\");
      sDir.append("minifuzz.log");

      g_pOut = new ofstream(sDir.c_str(),ios_base::app);
      if (!g_pOut)
            return false;

      string sTime;
      GetLogTime(&sTime);

      *g_pOut << "------------------------------" << endl;
      *g_pOut << "Log created: " << sTime << endl;
      *g_pOut << "------------------------------" << endl;

      return true;
}

// Close log file
void CloseLogFile() {
      assert(g_pOut != NULL);

      if (g_pOut == NULL)
            return;

      g_pOut->flush();
      g_pOut->close();
      g_pOut = NULL;
}

// Record an interesting debug event and capture state of the
// registers if malformed file causes application to deviate
// from its specified behavior
```

```
void ReportFuzzError(__in_z const char* szFilename, string sExc,
                               CONTEXT ctx) {
      Trace("\nFuzz error in ",false);
      Trace(szFilename,true);

      assert(g_pOut != NULL);

      if (g_pOut == NULL)
            return;

      string sTime;
      GetLogTime(&sTime);

      g_pOut->setf( ios_base::hex );

      *g_pOut << endl;
      *g_pOut << "******************************************" << endl;
      *g_pOut << "Fuzz error parsing " << szFilename << " at "
                  << sTime << endl;
      *g_pOut << sExc << endl;

      *g_pOut << setbase(16);
      *g_pOut << "EIP: [" << setw(12) << ctx.Eip << "]\t";
      *g_pOut << "EBP: [" << setw(12) << ctx.Ebp << "]" << endl;
      *g_pOut << "EDI: [" << setw(12) << ctx.Edi << "]\t";
      *g_pOut << "ESI: [" << setw(12) << ctx.Esi << "]" << endl;
      *g_pOut << "EAX: [" << setw(12) << ctx.Eax << "]\t";
      *g_pOut << "EBX: [" << setw(12) << ctx.Ebx << "]" << endl;
      *g_pOut << "ECX: [" << setw(12) << ctx.Ecx << "]\t";
      *g_pOut << "EDX: [" << setw(12) << ctx.Edx << "]" << endl;
      *g_pOut << "******************************************" << endl;
}

/* Log and trace functions */

void Log(__in_z const char *szMessage, __in_z const char *szArg,
             DWORD dwErr) {
      string sTime;
      GetLogTime(&sTime);
      *g_pOut << "[" << sTime << "] " << szMessage << " " << szArg
              << " err=" << dwErr << endl;
}

void Log(__in_z const char *szMessage, DWORD dwErr) {
      string sTime;
      GetLogTime(&sTime);
      *g_pOut << "[" << sTime << "] " << szMessage
              << " err=" << dwErr << endl;
}

void Log(__in_z const char *szMessage) {
      string sTime;
      GetLogTime(&sTime);
      *g_pOut << "[" << sTime << "] " << szMessage << endl;
}

void Trace(__in_z const char *szMessage, bool fCRLF) {
      clog << szMessage;
      if (fCRLF) cout << "\n";
}

void Trace(__in_z const char *szMessage) {
      Trace(szMessage, false);
}
```

5. Fuzz.cpp

```
/************************************************************
Filename:  fuzz.cpp
Original author:  Michael Howard
Modified by:  Clifford Juan

Source file for mutation-based fuzzing functions and
generation-based fuzzing functions used by fuzzing toolset.
************************************************************/

#include "stdafx.h"
#include "fuzz.h"
#include "log.h"

using namespace std;

/////////////////////////////////////////
// Is the CPU big or little-endian?
/////////////////////////////////////////
_ENDIAN CpuEndianNess() {
      return (htonl(1) == 1) ? ENDIAN_BIG : ENDIAN_LITTLE;
}

////////////////////////////////////////////////////////////////
// Takes an incoming file size, and picks a random range
// within that file size.
////////////////////////////////////////////////////////////////
void GetSizes(__in DWORD dwFileSize, __inout DWORD *pdwStart,
                    __inout DWORD *pdwSize) {

      // Make sure we have pointers!
      assert(pdwStart && pdwSize);
      if (!pdwStart || !pdwSize)
            return;

      // Max filesize is 2Gb
      assert(MAX_FILESIZE < _UI32_MAX / 2) ;
      if (dwFileSize >= MAX_FILESIZE)
            dwFileSize = MAX_FILESIZE;

      *pdwStart = GetRand() % dwFileSize;
      *pdwStart = (*pdwStart + 4 -1) & ~(4-1);

      *pdwSize = 1 + GetRand() % 1024;
      *pdwSize = min(dwFileSize, *pdwSize);
      if (*pdwSize + *pdwStart >= dwFileSize)
            *pdwStart = dwFileSize - *pdwSize;
}

//////////////////////////////////////////////////////////////////
// Seek to a file location, and allocate a blob of memory and
// load from the file into the blob
//////////////////////////////////////////////////////////////////
BYTE *SeekAllocAndRead(HANDLE hFile,DWORD dwStart, DWORD dwSize) {
      DWORD ptr = SetFilePointer(hFile,dwStart,NULL,FILE_BEGIN);
      if (ptr == INVALID_SET_FILE_POINTER) {
            Log("Unable to seek to file");
            return NULL;
      }

      // Caller must call delete[] on this blob
      BYTE *p = new BYTE[dwSize];
      if (!p) {
            Log("SeeUnable to alloc memory");
            return NULL;
      }

      DWORD dwBytesRead = 0;
      if (!ReadFile(hFile,p,dwSize,&dwBytesRead,NULL) ||
            dwSize != dwBytesRead) {
            Log("Unable to read from file");
            delete [] p;
            return NULL;
      }

      return p;
}

////////////////////////////////////////////////////
// Truncate the file, and return the new filesize
////////////////////////////////////////////////////
bool MalformSnipFile(HANDLE hFile, __inout DWORD *pdwSize) {
      Trace("S");

      if (!pdwSize)
            return false;

      *pdwSize -= GetRand() % (*pdwSize / 2);
      DWORD ptr = SetFilePointer(hFile,*pdwSize,NULL,FILE_BEGIN);
      if (ptr == INVALID_SET_FILE_POINTER) {
            Log("Unable to seek to file");
            return false;
      }

      return SetEndOfFile(hFile) ? true : false;
}

/////////////////////////////////////////
// Flip some random bytes in the file
/////////////////////////////////////////
bool MalformFlipAdjacentBytes(HANDLE hFile,DWORD dwSize) {
      Trace("F");

      const DWORD MAX_FLIPS = 10;
      const DWORD MAX_BYTES_TO_READ = 2;

      DWORD cNumberOfFlips = 1 + (GetRand() % MAX_FLIPS);

      for (DWORD i = 0; i < cNumberOfFlips; i++) {
         DWORD offFlip = GetRand() % (dwSize - MAX_BYTES_TO_READ);

         if (SetFilePointer(hFile,offFlip,NULL,FILE_BEGIN) ==
            INVALID_SET_FILE_POINTER) {
            continue;
         } else {
              DWORD dwBytesRead = 0;
              BYTE buff[MAX_BYTES_TO_READ];
              if (!ReadFile(hFile,&buff,MAX_BYTES_TO_READ,
                  &dwBytesRead,NULL)
                  || MAX_BYTES_TO_READ != dwBytesRead) {
                  Log("Unable to read from file");
                  continue;
              } else {
                   // flip the two bytes
                   BYTE t = buff[0];
                   buff[0] = buff[1];
                   buff[1] = t;

                   if (SetFilePointer(hFile,offFlip,NULL,FILE_BEGIN)
                      != INVALID_SET_FILE_POINTER) {
                      DWORD dwBytesWritten = 0;
                      WriteFile(hFile,buff,MAX_BYTES_TO_READ,
                         &dwBytesWritten,NULL);
                   }
              }
          }
      }

      return true;
}

///////////////////////////////////////////////////////
// Fill a random range in the file with random data
// Fills with one random char 50% of the time
// Fills with all random chars 50% of the time
///////////////////////////////////////////////////////
bool MalformFillRandom(HANDLE hFile,DWORD dwFileSize) {
      Trace("R");

      DWORD dwStart, dwSize;
      GetSizes(dwFileSize, &dwStart, &dwSize);
      BYTE *pWhereToWrite = SeekAllocAndRead(hFile,dwStart,dwSize);
      BYTE *pStart = pWhereToWrite;
      if (!pStart)
            return false;

      // 50% chance of pure random junk
      bool fRandomData = (GetRand() % 2) == 1;
      BYTE cChar = static_cast<BYTE>(GetRand() % 255);
      for (unsigned int i=0; i < dwSize; i++)
            *pWhereToWrite++ = (fRandomData) ?
                              (BYTE)GetRand() % 255 : cChar;

      if (SetFilePointer(hFile,dwStart,NULL,FILE_BEGIN)
            != INVALID_SET_FILE_POINTER) {
            DWORD dwBytesWritten = 0;
            WriteFile(hFile,pStart,dwSize,&dwBytesWritten,NULL);
      }

      delete [] pStart;

      return true;
}

/////////////////////////////////////////////////
// Searches for an ASCII or Unicode string,
// and then removes the NULL
/////////////////////////////////////////////////
bool MalformNukeString(__in HANDLE hFile,__in DWORD dwFileSize) {
      Trace("Z");

      // Pick a random spot in the file, and search for printable
      // ASCII or Unicode data. If a trailing NULL is found, then
      // replace it with a random non-NULL value.
      DWORD dwStart = 0;
      if (dwFileSize >= 10)
            dwStart = GetRand() % (dwFileSize-(dwFileSize/10));

      // must be even
      if (dwStart & 1)
            ++dwStart;

      if (SetFilePointer(hFile,dwStart,NULL,FILE_BEGIN)
                                    == INVALID_SET_FILE_POINTER)
            return false;

      bool fNullFound = false;
      bool fAsciiFound = false;
      bool fUnicodeFound = false;
      const size_t cchMinLen = 5;
      size_t len = 0;

      for (DWORD i=dwStart; i < dwFileSize-2; i+=2) {
         typedef union {
            BYTE                  b[2];
            unsigned short        w;
         } _BUF;

         assert(sizeof(_BUF) == 2);

         _BUF b;
         DWORD dwRead = 0;

         if (ReadFile(hFile,b.b,2,&dwRead,NULL) && 2 == dwRead) {
            int         i0 = static_cast<int>(b.b[0]);
            int         i1 = static_cast<int>(b.b[1]);
            unsigned short    w0 = static_cast<unsigned short>(b.w);

            if (isprint(i0) && isprint(i1)) {   // ASCII
               fAsciiFound = true;
               ++len;
            }  else if (!iswprint(w0)) {     // Unicode
                  fUnicodeFound = true;
                  ++len;
            }  else if (i0 == 0 || w0 == 0) {      // NULL value
                  if ((fUnicodeFound || fAsciiFound)
                     && len >= cchMinLen) {
                     // We found a string to malform
                     // (terminate the trailing NULL)
                     if (SetFilePointer(hFile,-2,NULL,FILE_CURRENT)
                         != INVALID_SET_FILE_POINTER) {
                         // Generate a random non-NULL value
                         BYTE r[2];
                         r[0] = 1
                            + static_cast<BYTE>(GetRand() % 255);
                         r[1] = 1
                            + static_cast<BYTE>(GetRand() % 255);
                         DWORD cbToWrite = fUnicodeFound ?
                              sizeof(WCHAR) : sizeof(BYTE);
                         DWORD cbWritten = 0;

                         WriteFile(hFile,r,cbToWrite,&cbWritten,NULL);
                     }

                     // Done
                     break;
                  }

                  } else {          // Nothing else to do
                       fUnicodeFound =
                       fAsciiFound =
                       fNullFound = false;

                       len = 0;
                  }
            }
      }

      return true;
}

/////////////////////////////////////////
// XORs a random series of bytes
/////////////////////////////////////////
bool MalformXorBits(HANDLE hFile,DWORD dwFileSize) {
      Trace("X");

      DWORD dwStart, dwSize;
      GetSizes(dwFileSize, &dwStart, &dwSize);
      BYTE *pWhereToWrite = SeekAllocAndRead(hFile,dwStart,dwSize);
      BYTE *pStart = pWhereToWrite;
      if (!pStart)
```

```
            return false;

      BYTE xor = (BYTE)(GetRand() % 256);

      // Apply XOR operation every n-bytes
      unsigned int cJump = 1 << GetRand() % 3;
      for (unsigned int i=0; i < dwSize; i += cJump) {
            *pWhereToWrite ^= xor;
            pWhereToWrite += cJump;
      }

      if (SetFilePointer(hFile,dwStart,NULL,FILE_BEGIN)
            != INVALID_SET_FILE_POINTER) {
            DWORD cbWritten = 0;
            WriteFile(hFile,pStart,dwSize,&cbWritten,NULL);
      }

      delete [] pStart;

      return true;
}

/////////////////////////////////////////
// Sets high-bits in a series of bytes
/////////////////////////////////////////
bool MalformHiBits(HANDLE hFile,DWORD dwFileSize) {
      Trace("H");

      DWORD dwStart, dwSize;
      GetSizes(dwFileSize, &dwStart, &dwSize);
      BYTE *pWhereToWrite = SeekAllocAndRead(hFile,dwStart,dwSize);
      BYTE *pStart = pWhereToWrite;
      if (!pStart)
            return false;

      // Set high-bit every n-bits
      unsigned int cJump = 1 << GetRand() % 4;
      for (unsigned int i=0; i < dwSize; i += cJump) {
            *pWhereToWrite |= 0x80;
            pWhereToWrite += cJump;
      }

      if (SetFilePointer(hFile,dwStart,NULL,FILE_BEGIN)
            != INVALID_SET_FILE_POINTER) {
            DWORD cbWritten = 0;
            WriteFile(hFile,pStart,dwSize,&cbWritten,NULL);
      }

      delete [] pStart;

      return true;
}
//////////////////////////////////////////////////////////////////////
// Find the number of "interesting bits" for this element
//////////////////////////////////////////////////////////////////////
extern const INTERESTING_BITS gEndMarker;
static size_t InterestingDataCount(__in INTERESTING_BITS *pBits) {
      if (!pBits)
            return 0;

      size_t i = 0;

      for (i = 0;
             memcmp((void*)&pBits[i], (void*)&gEndMarker,
                                    sizeof INTERESTING_BITS);
             i++)
            ;

      return i;
}

// Disable the compiler, because signed data and unsigned data
// are mixed purposely for fuzzing.
#pragma warning(push)
#pragma warning(disable:4245)

///////////////////////////////////////////////////////////
// Pick an interesting portion of the file to
// perform mutation-based "smart fuzzing"
///////////////////////////////////////////////////////////
bool MalformInterestingData(HANDLE hFile,DWORD dwFileSize,
                                    __in INTERESTING_BITS *pBits) {
      Trace("I");

      if (!pBits)
            return false;

      // Choose an "interesting" element to corrupt
      INTERESTING_BITS pPicked =
                        pBits[GetRand() % InterestingDataCount(pBits)];
      if (pPicked.cbSize >= dwFileSize)
            return false;

      typedef union {
            DWORD             dwAttackData;
            unsigned short    shAttackData[2];
            BYTE              bAttackData[4];
      } ATTACK_DATA;

      ATTACK_DATA att;
      memset(&att,0,sizeof ATTACK_DATA);

      // Tweak number with 'known bad numbers' 50% of the time
      if ((GetRand() % 10) >= 5) {

            DWORD dwBadNumbers[] = {
                        0xffffffff,
                        0xfffffffe,
                        0x80000001,
                        0x80000000,
                        0x7fffffff,
                        0x7ffffffe,
                        0x3fffffff,
                        0x3ffffffe,
                        0x100001,
                        0x100000,
                        0xfffff,
                        0xffffe,
                        0x10000,
                        0x10001,
                        0xfffe,
                        0xffff,
                        0x1001,
                        0x1000,
                        0xfff,
                        0xffe,
                        0x101,
                        0x100,
                        0xff,
                        0xfe,
                        0x1,
                        0,
                        -1};

            att.dwAttackData =
               dwBadNumbers[GetRand() % _countof(dwBadNumbers)];

      } else {
            // Create random value
            att.shAttackData[0] = (unsigned short)GetRand();
            att.shAttackData[1] = (unsigned short)GetRand();
      }

      // Change number to appropriate endian-ness
      if (pPicked.eEndian != CpuEndianNess()) {
            if (pPicked.eEndian == ENDIAN_BIG) {
                  // Convert to big; never used for HD Photo
                  att.dwAttackData = htonl(att.dwAttackData);
            }
            else {
                  // Convert to small (on Intel)
                  att.dwAttackData = ntohl(att.dwAttackData);
            }
      }

      BYTE *pWhereToWrite =
            SeekAllocAndRead(hFile,pPicked.cbOffset,pPicked.cbSize);
      BYTE *pStart = pWhereToWrite;
      if (!pStart)
            return false;

      switch(pPicked.cbSize) {
            case 1 : memcpy(pWhereToWrite,&att.bAttackData[0], 1);
                  break;
            case 2 : memcpy(pWhereToWrite,&att.shAttackData[0],2);
                  break;
            case 4 : memcpy(pWhereToWrite,&att.dwAttackData,   4);
                  break;
            default: memcpy(pWhereToWrite,&att.bAttackData[0], 1);
                  break;
      }
```

```
      if (SetFilePointer(hFile, pPicked.cbOffset, NULL, FILE_BEGIN)
            != INVALID_SET_FILE_POINTER) {
         DWORD dwBytesWritten = 0;
         WriteFile(hFile,pStart,pPicked.cbSize,&dwBytesWritten,NULL);
         assert(pPicked.cbSize == dwBytesWritten);
      }

      delete [] pStart;

      return true;
}


//////////////////////////////////////////////////////////////
//Generation fuzzer function
// - Part 1: Set the size to the 198-byte default blueprint
//////////////////////////////////////////////////////////////
bool MalformGenBasedResizeFile(HANDLE hFile, __inout DWORD *pdwSize) {
      Trace("Generation-Based Fuzzing");

      if (!pdwSize)
            return false;

      // Shape the file using the template's normal size of 198 bytes
      *pdwSize = *pdwSize - *pdwSize + 198;

      // Set the file size according to the type of test case selected
      DWORD ptr = SetFilePointer(hFile,*pdwSize,NULL,FILE_BEGIN);
      if (ptr == INVALID_SET_FILE_POINTER) {
            Log("Unable to seek to file");
            cout<<"Gen-based fuzzing: Unable to seek to file"<<endl;
            return false;
      }

      return SetEndOfFile(hFile) ? true : false;
}

//////////////////////////////////////////////////////////////
//Generation fuzzer function
// - Part 2: Generate the fuzzed HD Photo file using
//           test cases described in Chapter 4
//////////////////////////////////////////////////////////////
bool MalformGenBasedGenerateFile(HANDLE hFile,DWORD dwFileSize)
{
    //////////////////////////////////////////////////////////////
    // Implementation of test cases for generation-based fuzzing.
    //
    // These test cases are built using the 198-byte HD Photo
    // sample image described in Chapter 2 as the foundation.
    // Thus, the 198-byte image serves as the template
    // for fuzzing.
    //
    // This code fuzzes the 198-byte image with at least
    // one test case. The test cases are chosen at random.
    //////////////////////////////////////////////////////////////
```

```
DWORD dwStart, dwSize;
GetSizes(dwFileSize, &dwStart, &dwSize);
BYTE *pWhereToWrite = SeekAllocAndRead(hFile,dwStart,dwSize);
BYTE *pStart = pWhereToWrite;
if (!pStart)
      return false;

// Booleans to check which test cases to use
// in the generation-based fuzzing of HD Photo

// Fuzz the HD Photo version number
bool testCase1 = false;

// Malform the file with one of the mandatory IFD tags
// missing (such as the PixelFormat tag & the ImageWidth tag)
bool testCase2 = false;
unsigned int genSubCase2 = (GetRand() % 7) + 1;

// Fuzz the field value of all IFD entries
bool testCase3 = false;

// Fuzz the HD Photo Bitstream's GDI signature
bool testCase4 = false;

// Malform the file by leaving out all or parts of the
// HD Photo Bitstream
bool testCase5 = false;
unsigned int genSubCase5 = (GetRand() % 3) + 1;

// Fuzz the total number of directory entries advertised
// in the IFD
bool testCase6 = false;

// Malform the file by switching the order of the
// ImageHeight and ImageWidth tags in the IFD
bool testCase7 = false;

// Fuzz the value of the WidthResolution and HeightResolution
// IFD tags (which use IEEE float as the data type)
bool testCase8 = false;

// Fuzz the field Type for each of the IFD tags in the file
bool testCase9 = false;

// Fuzz the offset to the next IFD,
// instead of having the default value of
// four trailing nulls (indicating only one IFD)
bool testCase10 = false;
// Random fuzzing of IFD offset
bool test10_sub1 = (GetRand() % 2) == 1;
// Fuzz with value 0x00000008, attempt to force an
// infinite loop in parser
bool test10_sub2 = !(test10_sub1);

// Generate a HD Photo file with no Image File Directory
bool testCase11 = false;
```

```
// Generate original image, but with the BITSTREAM_FORMAT
// syntax element set
bool testCase12 = false;

// Generate original image, but with the alpha channel flag set
bool testCase13 = false; //(Part of testCase10)

// Fuzz the entire elementary HD Photo bitstream
// with random values
bool testCase14 = false;

// Fuzz the PixelFormat IFD tag's value
// (fuzzes image's pixel format)
bool testCase15 = false;

// Fuzz the value of the ImageWidth, ImageHeight,
// and ImageByteCount IFD tags
bool testCase16 = false;

// Malform file by generating a 90-degree clockwise rotation
// using the Transformation IFD tag, but keep the
// ImageWidth/ImageHeight tags the same.
bool testCase17 = false;

// Fuzz WIDTH_MINUS1 and HEIGHT_MINUS1 bitstream syntax elements
bool testCase18 = false;

// Generate original image, but with the tiling flag set
bool testCase19 = false;

// Generate original image, but with CLR_FMT syntax element
// set to RESERVED
bool testCase20 = false;

// Select a generation-based fuzzing test case at random
unsigned int chooseTestCase = (GetRand() % 20) + 1;

// Set the Boolean variable corresponding to the
// selected generation-based test case
if (dwSize == 198)
{
      switch(chooseTestCase) {
            case 1: testCase1 = true;
                  break;
            case 2: testCase2 = true;
                  break;
            case 3: testCase3 = true;
                  break;
            case 4: testCase4 = true;
                  break;
            case 5: testCase5 = true;
                  break;
            case 6: testCase6 = true;
                  break;
            case 7: testCase7 = true;
                  break;
            case 8: testCase8 = true;
```

```
                  break;
            case 9: testCase9 = true;
                  break;
            case 10: testCase10 = true;
                  break;
            case 11: testCase11 = true;
                  break;
            case 12: testCase12 = true;
                  break;
            case 13: testCase13 = true;
                  break;
            case 14: testCase14 = true;
                  break;
            case 15: testCase15 = true;
                  break;
            case 16: testCase16 = true;
                  break;
            case 17: testCase17 = true;
                  break;
            case 18: testCase18 = true;
                  break;
            case 19: testCase19 = true;
                  break;
            case 20: testCase20 = true;
                  break;
            default: testCase1 = true;
                  break;
      }
}

///////////////////////////////////////////////////////////////
// Generate a HD Photo file, based on the test case selected.
///////////////////////////////////////////////////////////////
for (unsigned int i=0; i < dwSize; i++)
{
      ///////////////////////////////////////////
      // Implement the selected test case
      ///////////////////////////////////////////
      if (i==0 || i==1) {
            *pWhereToWrite++ = (BYTE)73;
      }
      if (i==2) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==3) {
            // Fuzz the version number if test case selected,
            // otherwise set version number to 1
            if (testCase1) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)1;
            }
      }
      if (i==4) {
            *pWhereToWrite++ = (BYTE)8;
      }
```

```
if (i==5 || i==6 || i==7) {
      *pWhereToWrite++ = (BYTE)0;
}

if (testCase11 == true) {
      // Omit entire IFD from the HD Photo file
}
else { // Generate the IFD for the HD Photo file

// Number of entries in IFD, for our blueprint
// the total number is nine (9)
if (i==8) {
if (testCase6 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)9;
      }
}
if (i==9) {
      if (testCase6 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}

/////////////////////////////////////////////////////////
// Generate first IFD entry (PixelFormat tag, required)
/////////////////////////////////////////////////////////
if ((genSubCase2==1) && (testCase2==true)) {
      // Leave out first IFD entry as part of test case
}
else {
      if (i==10) {
            *pWhereToWrite++ = (BYTE)1;
      }
      if (i==11) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==12) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)1;
            }
      }
      if (i==13) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
```

```
    if (i==14) {
        *pWhereToWrite++ = (BYTE)16;
    }
    if (i==15) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==16 || i==17)     {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==18) {
        if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)122;
        }
    }
    if (i==19 || i==20 || i==21) {
        if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
}

//////////////////////////////////////
// Generate second IFD entry
// (Transformation tag, optional)
//////////////////////////////////////
if (i==22) {
    *pWhereToWrite++ = (BYTE)2;
}
if (i==23) {
    *pWhereToWrite++ = (BYTE)188;
}
if (i==24) {
    if (testCase9 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)4;
    }
}
if (i==25) {
    if (testCase9 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)0;
    }
}
if (i==26) {
    *pWhereToWrite++ = (BYTE)1;
}
if (i==27) {
      *pWhereToWrite++ = (BYTE)0;
}
if (i==28) {
      *pWhereToWrite++ = (BYTE)0;
}
if (i==29) {
      *pWhereToWrite++ = (BYTE)0;
}
if (i==30) {
      if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else if (testCase17 == true) {
            // Fuzz test case: Generate 90-degree rotation
            *pWhereToWrite++ = (BYTE)7;
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}
if (i==31) {
      if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}
if (i==32) {
      if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}
if (i==33) {
      if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}

///////////////////////////////////////////////////////
// Generate third IFD entry (ImageType tag, optional)
///////////////////////////////////////////////////////
if (i==34) {
      *pWhereToWrite++ = (BYTE)4;
}
if (i==35) {
      *pWhereToWrite++ = (BYTE)188;
}
if (i==36) {
      if (testCase9 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
```

```
    }
    else {
        *pWhereToWrite++ = (BYTE)4;
    }
}
if (i==37) {
    if (testCase9 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)0;
    }
}
if (i==38) {
    *pWhereToWrite++ = (BYTE)1;
}
if (i==39) {
    *pWhereToWrite++ = (BYTE)0;
}
if (i==40) {
    *pWhereToWrite++ = (BYTE)0;
}
if (i==41) {
    *pWhereToWrite++ = (BYTE)0;
}
if (i==42) {
    if (testCase3 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)0;
    }
}
if (i==43) {
    if (testCase3 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)0;
    }
}
if (i==44) {
    if (testCase3 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)0;
    }
}
if (i==45) {
    if (testCase3 == true) {
        *pWhereToWrite++ = (BYTE)GetRand() % 255;
    }
    else {
        *pWhereToWrite++ = (BYTE)0;
    }
}
```

```
/////////////////////////////////////////////////////////
// Generate fourth IFD entry (ImageWidth tag, required)
/////////////////////////////////////////////////////////
if ((genSubCase2==2) && (testCase2==true)) {
      // Leave out fourth IFD entry as part of test case
}
else {
      if (i==46) {
            if (testCase7 == true) {
                  *pWhereToWrite++ = (BYTE)129;
            }
            else {
                  *pWhereToWrite++ = (BYTE)128;
            }
      }
      if (i==47) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==48) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)4;
            }
      }
      if (i==49) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==50) {
            *pWhereToWrite++ = (BYTE)1;
      }
      if (i==51) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==52) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==53) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==54) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else if (testCase17 == true){
                  // Set image width to a number
                  // much greater than 1
                  *pWhereToWrite++ = (BYTE)255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)1;
            }
      }
      if (i==55) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else if (testCase17 == true){
                  // Set image width to a number
                  // much greater than 1
                  *pWhereToWrite++ = (BYTE)7;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==56) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==57) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
}

////////////////////////////////////////////////////////
// Generate fifth IFD entry (ImageHeight tag, required)
////////////////////////////////////////////////////////
if ((genSubCase2==3) && (testCase2==true)) {
      // Leave out fifth IFD entry as part of test case
}
else {
      if (i==58) {
            if (testCase7 == true) {
                  *pWhereToWrite++ = (BYTE)128;
            }
            else {
                  *pWhereToWrite++ = (BYTE)129;
            }
      }
      if (i==59) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==60) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
```

```
            *pWhereToWrite++ = (BYTE)4;
        }
    }
    if (i==61) {
        if (testCase9 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
    if (i==62) {
        *pWhereToWrite++ = (BYTE)1;
    }
    if (i==63) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==64) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==65) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==66) {
        if (testCase3 == true || testCase16 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)1;
        }
    }
    if (i==67) {
        if (testCase3 == true || testCase16 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
    if (i==68) {
        if (testCase3 == true || testCase16 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
    if (i==69) {
        if (testCase3 == true || testCase16 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
}
```

```
////////////////////////////////////////////
// Generate sixth IFD entry
// (WidthResolution tag, required)
////////////////////////////////////////////
if ((genSubCase2==4) && (testCase2==true)) {
      // Leave out sixth IFD entry as part of test case
}
else {
      if (i==70) {
            *pWhereToWrite++ = (BYTE)130;
      }
      if (i==71) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==72) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)11;
            }
      }
      if (i==73) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==74) {
            *pWhereToWrite++ = (BYTE)1;
      }
      if (i==75) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==76) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==77) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==78) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==79) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==80) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)192;
            }
      }
      if (i==81) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)66;
            }
      }
}

/////////////////////////////////////
// Generate seventh IFD entry
// (HeightResolution tag, required)
/////////////////////////////////////
if ((genSubCase2==5) && (testCase2==true)) {
      // Leave out seventh IFD entry as part of test case
}
else {
      if (i==82) {
            *pWhereToWrite++ = (BYTE)131;
      }
      if (i==83) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==84) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)11;
            }
      }
      if (i==85) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==86) {
            *pWhereToWrite++ = (BYTE)1;
      }
      if (i==87) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==88) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==89) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==90) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==91) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==92) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)192;
            }
      }
      if (i==93) {
            if (testCase3 == true || testCase8 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)66;
            }
      }
}
//////////////////////////////////////////////////////////
// Generate eighth IFD entry (ImageOffset tag, required)
//////////////////////////////////////////////////////////
if ((genSubCase2==6) && (testCase2==true)) {
      // Leave out eighth IFD entry as part of test case
}
else {
      if (i==94) {
            *pWhereToWrite++ = (BYTE)192;
      }
      if (i==95) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==96) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)4;
            }
```

```
    }
    if (i==97) {
        if (testCase9 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
    if (i==98) {
        *pWhereToWrite++ = (BYTE)1;
    }
    if (i==99) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==100) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==101) {
        *pWhereToWrite++ = (BYTE)0;
    }
    if (i==102) {
        if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)138;
        }
    }
    if (i==103) {
        if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
    if (i==104) {
        if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
    if (i==105) {
        if (testCase3 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
            *pWhereToWrite++ = (BYTE)0;
        }
    }
}

///////////////////////////////////////////////////
// Generate ninth IFD entry
// (ImageByteCount tag, required)
///////////////////////////////////////////////////
if ((genSubCase2==7) && (testCase2==true)) {
      // Leave out ninth IFD entry as part of test case
}
else {
      if (i==106) {
            *pWhereToWrite++ = (BYTE)193;
      }
      if (i==107) {
            *pWhereToWrite++ = (BYTE)188;
      }
      if (i==108) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)4;
            }
      }
      if (i==109) {
            if (testCase9 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==110) {
            *pWhereToWrite++ = (BYTE)1;
      }
      if (i==111) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==112) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==113) {
            *pWhereToWrite++ = (BYTE)0;
      }
      if (i==114) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)60;
            }
      }
      if (i==115) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==116) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==117) {
            if (testCase3 == true || testCase16 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
}

/////////////////////////////////////////////////////
// Generate offset to next IFD (if it exists)
/////////////////////////////////////////////////////
if (i==118) {
      if (testCase10 == true) {
            if (test10_sub1 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)8;
            }
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}
if (i==119) {
      if (testCase10 == true) {
            if (test10_sub1 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}
if (i==120) {
      if (testCase10 == true) {
            if (test10_sub1 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}
if (i==121) {
      if (testCase10 == true) {
            if (test10_sub1 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      else {
            *pWhereToWrite++ = (BYTE)0;
      }
}

} // End of IFD (and test case #11)

//////////////////////////////////////////
// Generate GUID of PixelFormat IFD tag
/////////////////////////////////////////
if (i==122) {
      *pWhereToWrite++ = (BYTE)36;
}
if (i==123) {
      *pWhereToWrite++ = (BYTE)195;
}
if (i==124) {
      *pWhereToWrite++ = (BYTE)221;
}
if (i==125) {
      *pWhereToWrite++ = (BYTE)111;
}
if (i==126) {
      *pWhereToWrite++ = (BYTE)3;
}
if (i==127) {
      *pWhereToWrite++ = (BYTE)78;
}
if (i==128) {
      *pWhereToWrite++ = (BYTE)254;
}
if (i==129) {
      *pWhereToWrite++ = (BYTE)75;
}
if (i==130) {
      *pWhereToWrite++ = (BYTE)177;
}
if (i==131) {
      *pWhereToWrite++ = (BYTE)133;
}
if (i==132) {
      *pWhereToWrite++ = (BYTE)61;
}
if (i==133) {
      *pWhereToWrite++ = (BYTE)119;
}
```

```
if (i==134) {
      *pWhereToWrite++ = (BYTE)118;
}
if (i==135) {
      *pWhereToWrite++ = (BYTE)141;
}
if (i==136) {
      *pWhereToWrite++ = (BYTE)201;
}
if (i==137) {
      if (testCase15 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)12;
      }
}

///////////////////////////////////////////////////
// Generate GDI Signature of HD Photo Bitstream
///////////////////////////////////////////////////
if (i==138) {
      if (testCase4 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else if (testCase14 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)87;
      }
}
if (i==139) {
      if (testCase4 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else if (testCase14 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)77;
      }
}
if (i==140) {
      if (testCase4 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else if (testCase14 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)80;
      }
}
if (i==141) {
      if (testCase4 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
```

```
        }
        else if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
              *pWhereToWrite++ = (BYTE)72;
        }
  }
  if (i==142) {
        if (testCase4 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
              *pWhereToWrite++ = (BYTE)79;
        }
  }
  if (i==143) {
        if (testCase4 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
              *pWhereToWrite++ = (BYTE)84;
        }
  }
  if (i==144) {
        if (testCase4 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
              *pWhereToWrite++ = (BYTE)79;
        }
  }
  if (i==145) {
        if (testCase4 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
        }
        else {
              *pWhereToWrite++ = (BYTE)0;
        }
  }

  //////////////////////////////////////////////
  // Generate rest of HD Photo Image Header
  //////////////////////////////////////////////
  if ((testCase5 == true) && (genSubCase5 == 2)) {
        // Execute test case to leave out Image Header
        // and Image Plane Header
  }
  else {
        if (i==146) {
              if (testCase14 == true) {
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
              }
              else {
                    *pWhereToWrite++ = (BYTE)16;
              }
        }
        if (i==147) {
              if (testCase14 == true) {
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
              }
              else if (testCase12 == true) {
                    // Set BITSTREAM_FORMAT syntax element
                    *pWhereToWrite++ = (BYTE)64;
              }
              else if (testCase19 == true) {
                    // Set the tiling flag for this template,
                    // even though original image is untiled
                    *pWhereToWrite++ = (BYTE)128;
              }
              else {
                    *pWhereToWrite++ = (BYTE)0;
              }
        }
        if (i==148) {
              if (testCase13 == true) {
                    // Set alpha channel flag for
                    // this template, even though
                    // original image has no such channel
                    *pWhereToWrite++ = (BYTE)193;
              }
              else if (testCase14 == true) {
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
              }
              else {
                    *pWhereToWrite++ = (BYTE)192;
              }
        }
        if (i==149) {
              if (testCase14 == true) {
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
              }
              else {
                    *pWhereToWrite++ = (BYTE)113;
              }
        }
        if (i==150) {
              if (testCase14 == true) {
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
              }
              else if (testCase17 == true){
                    // Test case 17: Set image width
                    // to a number much greater than 1
                    *pWhereToWrite++ = (BYTE)7;
              }
              else if (testCase18 == true) {
                 // Fuzz WIDTH_MINUS1
                 if ((GetRand() % 2) == 1) {
                    // Set byte value to interesting
                    // values 50% of the time
                    switch((GetRand() % 6) + 1) {
                       case 1: *pWhereToWrite++
                                    = (BYTE)191; //0xBF
                                break;
                       case 2: *pWhereToWrite++
                                    = (BYTE)207; //0xCF
                                break;
                       case 3: *pWhereToWrite++
                                    = (BYTE)223; //0xDF
                                break;
                       case 4: *pWhereToWrite++
                                    = (BYTE)239; //0xEF
                                break;
                       case 5: *pWhereToWrite++
                                    = (BYTE)254; //0xFE
                                break;
                       case 6: *pWhereToWrite++
                                    = (BYTE)255; //0xFF
                                break;
                       default: *pWhereToWrite++
                                    = (BYTE)255; //0xFF
                                break;
                    }
                 }
                 else {
                    // Set byte value to random value
                    // 50% of the time
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
                 }
              }
              else {
                    *pWhereToWrite++ = (BYTE)0;
              }
        }
        if (i==151) {
           if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
           }
           else if (testCase17 == true){
              // Test case 17: Set image width to
              // a number much greater than 1
              *pWhereToWrite++ = (BYTE)254;
           }
           else if (testCase18 == true) {
              // Fuzz WIDTH_MINUS1
              if ((GetRand() % 2) == 1) {
                 // Set byte value to interesting
                 // values 50% of the time
                 switch((GetRand() % 6) + 1) {
                    case 1: *pWhereToWrite++
                                 = (BYTE)191; //0xBF
                             break;
                    case 2: *pWhereToWrite++
                                 = (BYTE)207; //0xCF
                             break;
                    case 3: *pWhereToWrite++
                                 = (BYTE)223; //0xDF
                             break;
                    case 4: *pWhereToWrite++
                                 = (BYTE)239; //0xEF
                             break;
                    case 5: *pWhereToWrite++
                                 = (BYTE)254; //0xFE
                             break;
                    case 6: *pWhereToWrite++
                                 = (BYTE)255; //0xFF
                             break;
                    default: *pWhereToWrite++
                                 = (BYTE)255; //0xFF
                             break;
                          }
                    }
                    else {
                       // Set byte value to random value
                       // 50% of the time
                       *pWhereToWrite++ =
                                (BYTE)GetRand() % 255;
                    }
              }
              else {
                    *pWhereToWrite++ = (BYTE)0;
              }
        }
        if (i==152) {
           if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
           }
           else if (testCase18 == true) {
              // Fuzz HEIGHT_MINUS1
              if ((GetRand() % 2) == 1) {
                 // Set byte value to interesting
                 // values 50% of the time
                 switch((GetRand() % 6) + 1) {
                    case 1: *pWhereToWrite++
                                = (BYTE)191; //0xBF
                            break;
                    case 2: *pWhereToWrite++
                                = (BYTE)207; //0xCF
                            break;
                    case 3: *pWhereToWrite++
                                = (BYTE)223; //0xDF
                            break;
                    case 4: *pWhereToWrite++
                                = (BYTE)239; //0xEF
                            break;
                    case 5: *pWhereToWrite++
                                = (BYTE)254; //0xFE
                            break;
                    case 6: *pWhereToWrite++
                                = (BYTE)255; //0xFF
                            break;
                    default: *pWhereToWrite++
                                = (BYTE)255; //0xFF
                            break;
                    }
                 }
                 else {
                    // Set byte value to random
                    // value 50% of the time
                    *pWhereToWrite++ = (BYTE)GetRand() % 255;
                 }
              }
              else {
                    *pWhereToWrite++ = (BYTE)0;
              }
        }
        if (i==153) {
           if (testCase14 == true) {
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
           }
        else if (testCase18 == true) {
           // Fuzz HEIGHT_MINUS1
           if ((GetRand() % 2) == 1) {
              // Set byte value to interesting
              // values 50% of the time
              switch((GetRand() % 6) + 1) {
                 case 1: *pWhereToWrite++ = (BYTE)191; //0xBF
                          break;
                 case 2: *pWhereToWrite++ = (BYTE)207; //0xCF
                          break;
                 case 3: *pWhereToWrite++ = (BYTE)223; //0xDF
                          break;
                 case 4: *pWhereToWrite++ = (BYTE)239; //0xEF
                          break;
                 case 5: *pWhereToWrite++ = (BYTE)254; //0xFE
                          break;
                 case 6: *pWhereToWrite++ = (BYTE)255; //0xFF
                          break;
                default: *pWhereToWrite++ = (BYTE)255; //0xFF
                          break;
              }
           }
           else {
              // Set byte value to random value
              // 50% of the time
              *pWhereToWrite++ = (BYTE)GetRand() % 255;
           }
        }
        else {
           *pWhereToWrite++ = (BYTE)0;
        }
     }
  }
```

```
/////////////////////////////////////////////
// Generate the HD Photo Image Plane Header
/////////////////////////////////////////////
if ((testCase5 == true) && (genSubCase5 == 2
      || genSubCase5 == 3)) {
      // Execute test case to leave out Image Plane Header
}
else {
   if (i==154) {
      if (testCase14 == true) {
         *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else if (testCase20 == true) {
         // Set CLR_FMT value to 7 (Reserved)
         *pWhereToWrite++ = (BYTE)112;
      }
            else {
                  *pWhereToWrite++ = (BYTE)96;
            }
      }
      if (i==155) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)1;
            }
      }
      if (i==156) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)160;
            }
      }
      if (i==157) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==158) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)10;
            }
      }
      if (i==159) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==160) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==161) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)160;
            }
      }
      if (i==162) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==163) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
}

//////////////////////////////////////////
// Generate default HD Photo Index Table
//////////////////////////////////////////
if (i==164) {
      if (testCase14 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      else {
            *pWhereToWrite++ = (BYTE)255;
      }
}

//////////////////////////////////////
// Generate start of Tile Layer
//////////////////////////////////////
if (i==165) {
      if (testCase14 == true) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
```

```
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==166) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)0;
            }
      }
      if (i==167) {
            if (testCase14 == true) {
                  *pWhereToWrite++ = (BYTE)GetRand() % 255;
            }
            else {
                  *pWhereToWrite++ = (BYTE)1;
            }
      }

      ////////////////////////////////////////////////////////
      // Fill Macroblock and Block Layers with random data,
      // since it represents just the actual image data
      ////////////////////////////////////////////////////////
      if (i>=168 && i<=198) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
      if (i>198)  {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
}

if (testCase2 == true){
      // Fill rest of space left by omitted IFD entry with junk
      for (int i=0; i<12; i++) {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
}

if (testCase11 == true){
      // Image file directory is missing, so fill up
      // rest of the file with junk
      for (int i=0; i<114; i++)
      {
            *pWhereToWrite++ = (BYTE)GetRand() % 255;
      }
}

// Write the newly generated HD Photo semi-valid
// file template to file
if (SetFilePointer(hFile,dwStart,NULL,FILE_BEGIN)
                              != INVALID_SET_FILE_POINTER) {
      DWORD dwBytesWritten = 0;
      WriteFile(hFile,pStart,dwSize,&dwBytesWritten,NULL);
}
```

```
      // Free up the memory used to write our generated HD Photo file
      delete [] pStart;

      return true;
}

#pragma warning(pop)
```

6. MiniFuzz.cpp

```
/**************************************************************
Filename:  MiniFuzz.cpp
Original author:  Michael Howard
Modified by:  Clifford Juan

This is the main driver code for the MiniFuzz toolset
modified for fuzz testing HD Photo image viewer applications.
**************************************************************/

// Header files
#include "stdafx.h"
#include "fuzz.h"
#include "log.h"

using namespace std;

/* Interesting parts of an HD Photo file to fuzz */
static INTERESTING_BITS gInterestingHDPhoto[] = {

      ////////////////////////////////////////////////////////////////
      // "Interesting" locations to fuzz for specific HD Photo files.
      // Template: {file-offset, # bytes, byte-order
      // (always ENDIAN_LITTLE), comment}
      //
      // Note:  These locations may vary if another toolset other
      //        than the Geekpedia C++ source code listed in
      //        Appendix A is used to create the HD Photo image
      //        files used in fuzz testing.
      //        For the Geekpedia HD Photo conversion source code,
      //        which generated the sample HD Photo files used,
      //        these locations are fixed at the offsets
      //        provided below.
      //
      //        If another toolset is used to generate the HD Photo
      //        files you want to use with this fuzzer, be sure to
      //        adjust the offset locations to ensure that you are
      //        correctly fuzzing the exact bytes for each of the
      //        interesting elements in HD Photo described below.
      /////////////////////////////////////////////////////////////////

      /* 1st "smart fuzzing" test case */
      {3, 1, IB_OFFSET, ENDIAN_LITTLE, "version number"},
      /* 2nd "smart fuzzing" test case */
      {8, 2, IB_COUNT, ENDIAN_LITTLE, "# entries in 1st IFD"},
      /* 3rd "smart fuzzing" test case */
      {18, 4, IB_SIZE, ENDIAN_LITTLE, "offset to PixelFormat field"},
      /* 4th "smart fuzzing" test case */
      {30, 4, IB_SIZE, ENDIAN_LITTLE, "IFD Transformation field"},
      /* 5th "smart fuzzing" test case */
      {42, 4, IB_SIZE, ENDIAN_LITTLE, "IFD ImageType field"},

      /* 6th "smart fuzzing" test case */
      {54, 4, IB_SIZE, ENDIAN_LITTLE, "IFD image width"},
      /* 7th "smart fuzzing" test case */
      {66, 4, IB_SIZE, ENDIAN_LITTLE, "IFD image height"},
      /* 8th "smart fuzzing" test case */
      {78, 4, IB_SIZE, ENDIAN_LITTLE, "width resolution"},
      /* 9th "smart fuzzing" test case */
      {90, 4, IB_SIZE, ENDIAN_LITTLE, "height resolution"},
      /* 10th "smart fuzzing" test case */
      {102, 4, IB_OFFSET, ENDIAN_LITTLE,
            "IFD ImageOffset - image data offset"},

      /* 11th "smart fuzzing" test case */
      {114, 4, IB_COUNT, ENDIAN_LITTLE,
            "IFD ImageByteCount - image data in bytes"},
      /* 12th "smart fuzzing" test case */
      {118, 4, IB_OFFSET, ENDIAN_LITTLE, "IFD offset to next IFD"},
      /* 13th "smart fuzzing" test case */
      {150, 2, IB_SIZE,  ENDIAN_LITTLE,
            "HD Photo Bitstream - Image Width"},
      /* 14th "smart fuzzing" test case */
      {152, 2, IB_SIZE,  ENDIAN_LITTLE,
            "HD Photo Bitstream - Image Height"},

      gEndMarker
};


// Fuzzing Windows Media Photo / HD Photo files only!
// (Only files with .wdp extension)
const DATATYPES gDataTypes[] = {
      /* As of July 2007, Windows Photo Gallery
      only accepts .wdp extension for HD Photo */
                  {".wdp", gInterestingHDPhoto},
      };

// Global list of valid HD Photo files to corrupt
vector<string>    gFilenames;

// Store fuzzed HD Photo files successful in
// crashing an app in "badfiles" dir
const string TEMPDIR("badfiles");

void Usage() {
      cout << endl;
      cout << "Usage: MiniFuzz [dir] [app]" << endl;
      cout << endl;
      cout << "Where <dir> is the directory containing"
           << " the valid HD Photo files for testing, " << endl;
      cout << "and <exe> is the executable to test"
```

```
           << " for HD Photo fuzzing."
           << endl << endl;
      cout << "MiniFuzz will create a dir named '"
           << TEMPDIR << "' under <dir> " << endl;
      cout << "to temporarily store malformed HD Photo files." << endl;
}

// Given a filename, find out if it is an HD Photo file extension
// If it is, return true.  If it is not, return false.
bool IsFileICareAbout(__in_z const char *szFilename) {

      if (szFilename == NULL)
            return false;

      if (*szFilename == '.')
            return false;

      // Make sure it's a HD Photo file type.
      // If it's not, skip the filename
      string filename = szFilename;
      transform(filename.begin(),
            filename.end(),filename.begin(),tolower);
      for (int i = 0;
            i < sizeof(gDataTypes) / sizeof(gDataTypes[0]); i++)
         if (filename.find(gDataTypes[i].szExtension,0)
                                                != string::npos)
            return true;

      cout << "Skipping '" << filename << "'" << endl;
      return false;
}

// Read all the files from the template directory
// (first argument to the application)
bool GetAllFiles(__in_z const char *szDir) {

      if (szDir == NULL)
            return false;

      string dir = szDir;
      dir.append("\\*");

      gFilenames.empty();

      WIN32_FIND_DATA fd;
      memset(&fd,0,sizeof WIN32_FIND_DATA);
      HANDLE hFile = FindFirstFile(dir.c_str(),&fd);
      if (hFile == INVALID_HANDLE_VALUE)
            return false;

      if (IsFileICareAbout(fd.cFileName))
            gFilenames.push_back(fd.cFileName);

      while(FindNextFile(hFile,&fd) != 0)
            if (IsFileICareAbout(fd.cFileName))
                  gFilenames.push_back(fd.cFileName);
      FindClose(hFile);

      return true;
}

// Given a filename, gets the "interesting bits" about that filetype
// Use the INTERESTING_BITS construct to aid with
// mutation-based "smart fuzzing"
// Returns NULL if the filetype is not found
INTERESTING_BITS *GetInterestingBitsFromFilename(string sFilename) {
   INTERESTING_BITS *pIb = NULL;

   for (size_t i = 0; i < _countof(gDataTypes); i++) {
     string sExt(gDataTypes[i].szExtension);

     transform(sExt.begin(), sExt.end(),sExt.begin(),tolower);
     transform(sFilename.begin(),
         sFilename.end(),sFilename.begin(),tolower);

     size_t cExtLen = sExt.length();
     size_t cFileNameLen = sFilename.length();
     if (cFileNameLen > cExtLen) {
       if (0 == sFilename.compare(cFileNameLen - cExtLen,cExtLen,sExt))
       {
          pIb = gDataTypes[i].pBits;
          break;
       }
     }
   }

   return pIb;
}

bool FuzzAFile(__in_z const char *szDir, __in_z const char *szExe) {
      if (szDir == NULL || szExe == NULL || gFilenames.size() == 0)
            return false;

      ////////////////////////////////////////////////////////////
      // HD Photo Fuzzing steps
      // (1)Select a HD Photo filename at random from
      //    the gFilename vector
      // (2)Save a copy of the valid HD Photo file
      //    to a temporary file
      // (3)Malform the temporary file using either
      //    generation-based fuzzing or mutation-based fuzzing
      // (4)Launch the target application with the
      //    fuzzed HD Photo file
      ////////////////////////////////////////////////////////////

      ////////////////////////////////////////////////////////////
      // (1)Select a HD Photo filename at random from
      //    the gFilename vector
      ////////////////////////////////////////////////////////////
      static unsigned __int64 cCount;
      char szNum[32];
      _ui64toa_s(++cCount,szNum,_countof(szNum),16);
```

```
size_t whichFile = GetRand() % gFilenames.size();
string filename = szNum;
filename.append("-");
filename.append(gFilenames[whichFile].c_str());

///////////////////////////////////////////////////////////
// (2)Save a copy of the valid HD Photo file
//    to a temporary file
///////////////////////////////////////////////////////////
string temp = szDir;
temp.append("\\");
temp.append(TEMPDIR);
temp.append("\\");
temp.append(filename);

string from = szDir;
from.append("\\");
from.append(gFilenames[whichFile].c_str());

// Copy the correct file to temp dir
BOOL f = CopyFile(from.c_str(), temp.c_str(),TRUE);
if (!f && GetLastError() != ERROR_FILE_EXISTS) {
      Log("Unable to copy ", from.c_str(), GetLastError());
      return false;
}

// Make sure we can write to the file, and we don't want
// the indexer reading this
SetFileAttributes(temp.c_str(),

FILE_ATTRIBUTE_NORMAL|FILE_ATTRIBUTE_NOT_CONTENT_INDEXED);

// Open the file - this is the handle we'll use for
// all corruption
HANDLE hFile = CreateFile(temp.c_str(),
      GENERIC_WRITE | GENERIC_READ,
      0,
      NULL,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL | FILE_FLAG_RANDOM_ACCESS |
      FILE_FLAG_WRITE_THROUGH,
      NULL);
if (hFile == INVALID_HANDLE_VALUE) {
      Log("Unable to open ", temp.c_str(), GetLastError());
      return false;
}

LARGE_INTEGER size;
if (!GetFileSizeEx(hFile,&size) || size.QuadPart >= MAX_FILESIZE)
{
      Log("Unable to open file, or it's too large",
            temp.c_str(), GetLastError());
      return false;
}

Trace(filename.c_str());
Trace("\t");
```

```
Log(filename.c_str());

///////////////////////////////////////////////////////////
// (3)Malform the temporary file using either
//    generation-based fuzzing or mutation-based fuzzing
///////////////////////////////////////////////////////////
bool fFileIsSnipped = false;
size_t iNumberOfMalforms = GetRand() % MAX_MALFORMS;
DWORD dwFileSize = static_cast<DWORD>(size.QuadPart);

/* Boolean flag indicates whether to do
   smart fuzzing or dumb fuzzing */
bool selectMutationBasedFuzzing = ((GetRand() %2) == 0);

if (selectMutationBasedFuzzing == true){
   cout<<endl
       <<"Mutation-based fuzzing chosen to malform file...\n";
}
else{
   cout<<endl
       <<"Generation-based fuzzing chosen to malform file...\n";
}

bool smartFuzzing = false;

/* If we're doing mutation-based fuzzing, randomly pick
   whether to conduct "dumb fuzzing" or "smart fuzzing" */
if (selectMutationBasedFuzzing == true){
   if ( (GetRand() % 2) == 0){
      cout<<"[*] Selected smart fuzzing...\n";
      smartFuzzing = true;
   }
   else {
      cout<<"[*] Selected dumb fuzzing...\n";
      smartFuzzing = false;
   }
}

/*******************************************************/
/* Start of code for conducting mutation-based fuzzing */
/*******************************************************/
if (selectMutationBasedFuzzing == true){

      // Conduct dumb fuzzing 50% of the time
      if (smartFuzzing == false)
      {
         cout<<"Conducting dumb fuzzing\n";

         // Apply the test cases "n" number of times,
         // where 0 <= n <= iNumberofMalforms
         // Note: maximum number of malforms (MAX_MALFORMS)
         //       is contained in "Fuzz.h" and can be modified
         //       to suit the needs of the tester.
         for (unsigned int i=0; i < iNumberOfMalforms; i++)
         {
            switch(GetRand() % MALFORM_LAST)
            {
               /* "Dumb fuzzing" test case -
                  truncating the HD Photo file */
               case MALFORM_SNIP_FILE:
                  // Can only truncate a file one time
                  if (fFileIsSnipped)
                     MalformFillRandom(hFile,dwFileSize);
                  else {
                     DWORD dwSize = dwFileSize;
                     if ((fFileIsSnipped =
                        MalformSnipFile(hFile, &dwSize)) == true)
                        size.QuadPart = dwFileSize = dwSize;

                  }
                  break;

               /* "Dumb fuzzing" test case -
                  find an ASCII string inside
                  the HD Photo file and remove
                  the null terminator */
               case MALFORM_NUKE_SZ :
                  MalformNukeString(hFile, dwFileSize);
                  break;

               /* "Dumb fuzzing" test case -
                  toggle on or off the most significant
                  bits of a series of bytes inside
                  a valid HD Photo file */
               case MALFORM_FLIP_HIGH_BITS :
                  MalformHiBits(hFile, dwFileSize);
                  break;

               /* "Dumb fuzzing" test case -
                  perform an exclusive OR (XOR)
                  operation on a range of bytes
                  (selected at random) inside a
                  valid HD Photo file with
                  random values */
               case MALFORM_XOR_BITS:
                  MalformXorBits(hFile, dwFileSize);
                  break;

               /* Use "smart fuzzing" test cases defined
                  previously in this source code and mix
                  it in with "dumb fuzzing" */
               case MALFORM_INTERESTING_DATA : {
                  INTERESTING_BITS *pBits =
      GetInterestingBitsFromFilename(gFilenames[whichFile]);

                  if (pBits)
                     MalformInterestingData(hFile,dwFileSize,
                                                      pBits);
                  else
                     MalformFillRandom(hFile, dwFileSize);
                  }
                  break;
```

```
               /* "Dumb fuzzing" test case -
                  switch adjacent bytes
                  inside a valid HD Photo file */
               case MALFORM_FLIP_ADJACENT_BYTES :
                  MalformFlipAdjacentBytes(hFile, dwFileSize);
                  break;

               /* "Dumb fuzzing" test case -
                  replace a randomly-selected range
                  of bytes inside a valid HD Photo
                  file with random values */
               case MALFORM_FILL_RANDOM :
               default:
                  MalformFillRandom(hFile, dwFileSize);
                  break;
            } //end switch-statement
         } //end for-loop
      }//end if-statement

      // Conduct smart fuzzing 50% of the time
      else
      {
         cout<<"Conducting smart fuzzing\n";
         INTERESTING_BITS *pBits =
           GetInterestingBitsFromFilename(gFilenames[whichFile]);

         // If there is interesting bits to fuzz,
         // then fuzz that data.
         // Otherwise, fill the file with random data
         if (pBits)
            MalformInterestingData(hFile,dwFileSize,pBits);
         else
            MalformFillRandom(hFile, dwFileSize);
      }

}
/*******************************************************/
/* End of code for mutation-based fuzzing              */
/*******************************************************/


/*******************************************************/
/* Start of code for generation-based fuzzing          */
/*******************************************************/
if (selectMutationBasedFuzzing == false)
{
   // Resize current file to our generation template for HD Photo
   bool fFileIsGenerated = false;
   DWORD dwFileGenSize = static_cast<DWORD>(size.QuadPart);
   DWORD dwGenSize = dwFileGenSize;

   if ((fFileIsGenerated =
      MalformGenBasedResizeFile(hFile, &dwGenSize)) == true)
      size.QuadPart = dwFileGenSize = dwGenSize;

   // Call function to generate malformed HD Photo file
   MalformGenBasedGenerateFile(hFile,dwGenSize);
```

```
      }
      /*******************************************************/
      /* End of code for generation-based fuzzing            */
      /*******************************************************/


      if (hFile != INVALID_HANDLE_VALUE)
            CloseHandle(hFile);

      //////////////////////////////////////////////////
      // (4)Launch the exe
      //////////////////////////////////////////////////
      bool fDeleteTempFile = false;
      LaunchFile(szDir, szExe, temp.c_str(), &fDeleteTempFile);

      if (fDeleteTempFile)
            if (!DeleteFile(temp.c_str())) {
                  // small delay in case the file is still locked
                  Sleep(700);
                  if (!DeleteFile(temp.c_str()))
                        Log("DeleteFile failed", temp.c_str(),
                                                GetLastError());
            }

      Trace("\n");

      return true;
}

/****************************************************************
// This Boolean function enforces that this toolset can
// only be used by admin users.
// This is done by ensuring that the user's token is verified.
****************************************************************/
__checkReturn bool IsUserAdmin() {
      SID_IDENTIFIER_AUTHORITY NtAuthority = SECURITY_NT_AUTHORITY;
      PSID AdministratorsGroup;
      BOOL b = AllocateAndInitializeSid(
            &NtAuthority,
            2,
            SECURITY_BUILTIN_DOMAIN_RID,
            DOMAIN_ALIAS_RID_ADMINS,
            0, 0, 0, 0, 0, 0,
            &AdministratorsGroup);

      if(b) {
            if (!CheckTokenMembership(NULL, AdministratorsGroup, &b))
                   b = FALSE;

            FreeSid(AdministratorsGroup);
      }

      return b ? true : false;
}
// Self-explanatory ... set heap options, doesn't matter if this fails
void SetHeapOptions() {
      ULONG  HeapFragValue = 2;
```

```
    HeapSetInformation(GetProcessHeap(),
                       HeapCompatibilityInformation,
                       &HeapFragValue,
                       sizeof(HeapFragValue));
// Windows Vista only
#ifdef HeapEnableTerminationOnCorruption
      (void)HeapSetInformation(NULL, HeapEnableTerminationOnCorruption,
NULL, 0
#endif
}

///////////////////////////////////////////////
// Main program for fuzzing toolset
///////////////////////////////////////////////
int main(int argc, char* argv[]) {

      // Exit if user isn't admin or number of
      // command-line arguments isn't correct
      if (!IsUserAdmin() || argc != 3) {
            Usage();
            return 1;
      }

      SetHeapOptions();

      // Exit if we can't read the HD Photo files in
      // the target directory (1st cmd-line argument)
      if (!GetAllFiles(argv[1])) {
            cerr << "Unable to read template files err="
                   << GetLastError() << endl;
            return 1;
      }

      string tempdir = argv[1];
      tempdir.append("\\");
      tempdir.append(TEMPDIR);
      if (!CreateDirectory(tempdir.c_str(),NULL) && GetLastError()
                                          != ERROR_ALREADY_EXISTS) {
            cerr << "Unable to create temp dir err="
                 << GetLastError() << endl;
            return 1;
      }

      if (!OpenLogFile(tempdir)) {
            cerr << "Unable to create log file err="
                 << GetLastError() << endl;
            return 1;
      }

      srand((unsigned int)time(0));

      // Fuzz away at selected HD Photo application until
      // user kills the fuzzing toolset
      // (*) 1st arg is the target directory
      //     holding the valid HD Photo files
      // (*) 2nd arg is the target application
      //     to fuzz using HD Photo
      for (;;) {
            if (!FuzzAFile(argv[1], argv[2]))
                  break;
      }

      // Write to log file and close it
      CloseLogFile();

      // End of program
      return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[DeMott 2006] DeMott, J. (2006, June). "The Evolving Art of Fuzzing." Retrieved March 2007 from the World Wide Web: http://www.vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf

[Evers 2006] Evers, J. CNET News.com. (2006, May 25) "Microsoft shows off JPEG rival." Retrieved December 2006 from the World Wide Web: http://news.com.com/2100-1025_3-6076650.html

[Geekpedia 2007] Geekpedia Website. (2007, May) "Programming tutorial: Converting Graphics to HD Photo (WDP/ HDP)". Retrieved May 2007 from the World Wide Web: http://www.geekpedia.com/tutorial208_Converting-Graphics-to-HD-Photo-(WDP-or-HDP).html

[Gruener 2007] Gruener, Wolfgang. (2007, April) TGDaily.com. "TG Daily – Love at First Sight? Another Look at Microsoft's HD Photo format." Retrieved April 2007 from the World Wide Web: http://www.tgdaily.com/content/view/31492/113/

[Harris 2005] Harris, S., Harper, A., Eagle, C., Ness, J., and Lester, M. (2005) *Gray Hat Hacking: The Ethical Hacker's Handbook.* California: McGraw-Hill/Osborne.

[HDPhotoFeatureSpec 2006] Microsoft Corporation. (2006, November) "HD Photo Feature Specification." Retrieved January 2007 from the World Wide Web: http://www.microsoft.com/downloads/info.aspx?na=90&p=&SrcDisplayLang=en&SrcCategoryId=&SrcFamilyId=6fe1ebac-c7b3-4768-90bc-13d330d5ec02&u=http%3a%2f%2fdownload.microsoft.com%2fdownload%2f7%2f4%2f3%2f74394b57-2028-4859-a668-c4c0af0726e2%2fHDPhoto_Feature_Spec_1.0.doc

[HDPhotoBitstreamSpec 2006] Microsoft Corporation. (2006, November) "HD Photo Bitstream Specification." Retrieved February 2007 from the World Wide Web: http://www.microsoft.com/downloads/info.aspx?na=90&p=&SrcDisplayLang=en&SrcCategoryId=&SrcFamilyId=285eeffd-d86c-48c3-ab93-3abd5ee7f1ce&u=http%3a%2f%2fdownload.microsoft.com%2fdownload%2f1%

2f7%2f4%2f1743e193-c99b-4ffa-9365-b3bb4c2a736a%2fhdphoto_1.0_dpk_setup.exe

(Note: This document is included as part of the provided documentation in the "HD Photo Device Porting Kit" package. It can be found under the ***/doc*** directory, with the filename "***HDPhoto_Bitstream_Spec_1.0.doc***".)

[Howard 2006] Howard, M. and Lipner, S. (2006) *The Security Development Lifecycle*. Washington: Microsoft Press.

[Jurrien 2007] Jurrien, R. (2007, August) "JPEG XR format from Microsoft." Retrieved August 2007 from the World Wide Web: http://www.letsgodigital.org/en/16123/jpeg-xr-format/

[Lipner 2005] Lipner, S. and Howard, M. (2005, March) "The Trustworthy Computing Security Development Lifecycle." Retrieved June 2007 from the World Wide Web: http://msdn2.microsoft.com/en-us/library/ms995349.aspx

[MacNN 2007] MacNN.com. (2007, March) "MS Unveils HD Photo Photo File Format." Retrieved April 2007 from the World Wide Web: http://macnn.com/print/42443

[Miller 1990] Miller, B., Fredriksen, L., and So, B. (1990, December) "An Empirical Study of the Reliability of UNIX Utilities." Communications of the ACM, Volume 33, Issue 12, pp. 32-44. December 1990.

[MKH 2007] MKH Computer Services Website. (2007) "MKH Computer Services – Glossary." Retrieved April 2007 from the World Wide Web: http://www.mkh-computer-services.co.uk/glossary/glossary_all.php

[MS-HDPhoto 2007] Microsoft.com. (2006) "HD Photo Main Page." Retrieved February 2007 from the World Wide Web: http://www.microsoft.com/windows/windowsmedia/forpros/wmphoto/default.aspx

[Oehlert 2005] Oehlert, P. (2005, April) "Violating Assumptions With Fuzzing." IEEE Security & Privacy, March/April 2005, pp. 58-62.

[Padilla 2005] Padilla, O. (2005, December) "Analyzing Common Binary Parser Mistakes." Retrieved July 2007 from the World Wide Web: http://www.uninformed.org/?v=3&a=1&t=txt

[Paintdotnet 2007] Paint.NET Forum. (2007, April) "Paint.NET – HD Photo plugin –

BETA." Retrieved June 2007 from the World Wide Web: http://paintdotnet.forumer.com/viewtopic.php?t=4124

[QuickOnlineTips 2007] QuickOnlineTips.com (2007, March) "Microsoft HD Photo: New Imaging File Format." Retrieved April 2007 from the World Wide Web: http://www.quickonlinetips.com/archives/2007/03/microsoft-hd-photo-new-imaging-file-format/

[SANS 2007] SANS Institute Website. (2006). "SANS Institute – SANS Top-20 Internet Security Attack Targets (2006 Annual Update)." Retrieved April 2007 from the World Wide Web: http://www.sans.org/top20/

[Seltzer 2006] Seltzer, L. (2006, February). "The Future of Security Gets Fuzzy." Retrieved June 2007 from the World Wide Web: http://www.eweek.com/article2/0,1759,1914332,00.asp

[Shankland 2007] Shankland, S. (2007, March). ZDNews.com. "Microsoft: Make our HD Photo format a standard." Retrieved March 2007 from the World Wide Web: http://news.zdnet.com/2100-3513_22-6165004.html

[Sutton 2007] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. New Jersey: Addison Wesley.

[TIFF1 2007] FileFormat.info. (2007) "TIFF File Format Summary." Retrieved April 2007 from the World Wide Web: http://www.fileformat.info/format/tiff

[TIFF2 1997] EE458 Course Page at Cooper University. (1997) "The TIFF Image File Format." Retrieved April 2007 from the World Wide Web: http://www.ee.cooper.edu/courses/course_pages/past_courses/EE458/TIFF/

[TIFF3 1992] Adobe Developers Association. (1992, June) "TIFF Revision 6.0." Retrieved April 2007 from the World Wide Web: http://partners.adobe.com/asn/developer/PDFS/TN/TIFF6.pdf

[Warnock 2007] Warnock, M. (2007). "Look out! It's the fuzz!" Ianewsletter: The Newsletter for Information Assurance Technology Professionals, Vol. 10, No.1, Spring 2007, pp. 4-9.

[Weinstein 2007] Weinstein, D. (2007, July). "Fuzzing Fundamentals." Redmond Developers News. Retrieved July 2007 from the World Wide Web: http://reddevnews.com/techbriefs/article.aspx?editorialsid=261

[Wikipedia-BitmapImage 2007] Wikipedia Site. (2007) “Bitmap image.” Retrieved June 2007 from the World Wide Web: http://en.wikipedia.org/wiki/Windows_and_OS/2_bitmap

[Wikipedia-HDPhoto 2007] Wikipedia.org. (2007) “HD Photo.” Retrieved March 2007 from the World Wide Web: http://en.wikipedia.org/wiki/HD_Photo

[Wikipedia-IEEE754 2007] Wikipedia.org. (2007) “IEEE 754.” Retrieved May 2007 from the World Wide Web: http://en.wikipedia.org/wiki/IEEE_754

[Wikipedia-IntegerOverflow 2007] Wikipedia Site. (2007) “Integer overflow.” Retrieved June 2007 from the World Wide Web: http://en.wikipedia.org/wiki/Integer_overflow

[Wikipedia-WPG 2007] Wikipedia Site. (2007) “Windows Photo Gallery.” Retrieved June 2007 from the World Wide Web: http://en.wikipedia.org/wiki/Windows_Photo_Gallery

[Wysopal 2007] Wysopal, C., Nelson, L., Zovi, D., and Dustin, E. (2007) *The Art of Software Security Testing.* New Jersey: Addison-Wesley.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia

2. Dudley Knox Library
Naval Postgraduate School
Monterey, California

3. Dr. Bret Michael
Naval Postgraduate School
Monterey, California

4. Chris Eagle
Naval Postgraduate School
Monterey, California

5. Dr. Cynthia Irvine
Naval Postgraduate School
Monterey, California

6. Dr. Ron Ritchey
Information Assurance Technology Analysis Center
Herndon, VA

7. Mr. Gene Tyler
Information Assurance Technology Analysis Center
Herndon, VA

8. Mr. Richard Hale
Defense Information Systems Agency
Falls Church, VA

9. Dr. Steven King
Office of the Deputy Under Secretary of Defense for S&T
Rosslyn, VA

10. Clifford Juan
Naval Postgraduate School
Monterey, California