Inspection and Sanitization Guidance for Waveform Audio File Format

Version 1.0

1 March 2012

National Security Agency
9800 Savage Rd, Suite 6721
Ft. George G. Meade. MD 20755

Authored/Released by:
Unified Cross Domain Capabilities Office
cds_tech@nsa.gov

DOCUMENT REVISION HISTORY

Date	Version	Description
3/1/2012	1.0	final
12/13/2017	1.0	Updated Contact information, IAC Logo, Cited Trademarks and Copyrights, Expanded Acronyms, and added Legal Disclaimer

EXECUTIVE SUMMARY

This *Inspection and Sanitization Guidance for Waveform Audio File Format* document provides guidelines and specifications for developing file inspection and sanitization software for the Waveform Audio File Format, hereafter referred to as WAVE.

The WAVE file format is Microsoft Windows' native file format for the storage of digitized sound. The format was developed in the early 1990s and its use has been growing ever since. Given the increasing use of digital audio, WAVE file formats are likely to see even wider applications.

The WAVE format is a type of the more general Resource Interchange File Format (RIFF) multimedia data standard. WAVE files are composed of *chunks* that are typically referred to as *WAVE chunks*. A RIFF file is composed of a single RIFF chunk; however, the RIFF chunk's data field may contain one or more *subchunks*.

WAVE file vulnerabilities can typically be categorized into three main groups:

- Format errors in a WAVE file
- Use of nonstandard chunk types and data formats
- Embedding atypical data in correctly formatted WAVE chunks or modifying existing data

The first group consists of vulnerabilities due to format errors in a WAVE file. Format errors may cause exceptions in a WAVE application (e.g., media player) when a malformed WAVE file is processed, resulting in buffer overflow and/or denial-of-service (DoS) conditions.

The second group of WAVE file vulnerabilities is associated with the use of nonstandard chunk types and data formats. Given the WAVE specifications' requirement for WAVE applications to ignore unrecognized chunks, malicious users could insert data (potentially sensitive) or executable code (potentially malicious) into a WAVE file as part of a nonstandard unrecognized chunk. If a WAVE application simply skips over unknown types of chunks, large sections of the file could remain undetected or unchecked.

The third group of WAVE format vulnerabilities involves embedding atypical data in correctly formatted WAVE chunks or modifying existing data. Although the WAVE format does not define or support embedded executable objects, nothing prevents a malicious user or WAVE application from storing executable code within a WAVE chunk and subsequently executing it. A malicious user could also embed sensitive information within a WAVE chunk to transfer the information illegally from a high-side network to a low-side network.

This document proposes functional requirements for a WAVE analysis software that could be capable of inspecting and sanitizing WAVE files and filtering these files as appropriate. Using the information provided in this report, reviewers can identify and analyze potential locations where embedded hidden or malicious data may reside in WAVE files. The proposed WAVE file inspection techniques include comprehensive analysis and validation of a WAVE file layout, including analysis and validation of each chunk's format and fields, along with signal processing-based pattern recognition of the expected audio data contained within the file.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SCOPE

1.1 Purpose of this Document

The purpose of this document is to provide guidance for the development of a sanitization and analysis software tool for the WAVE file format. It provides inspection and analysis guidance on the various constituents that are contained within the WAVE file structure and describes how they can be a cause for concern for either hiding sensitive data or attempts to exploit a system.

This document provides an analysis of numerous features in WAVE files and provides recommendations to mitigate these threats to provide a safer file. Although this report does not address vulnerabilities related to specific WAVE reader software applications, several applications were utilized in the analysis of the WAVE standard.

The intended audience of this document includes system engineers, designers, software developers, and testers who work on file inspection and sanitization applications that involve processing WAVE documents.

1.2 Introduction

The WAVE file format is Microsoft®[1] Windows' native file format for the storage of digitized sound. The format was developed in the early 1990s and its use has been growing ever since. The WAVE format has expanded and grown, and with the move to digital audio, WAVE file formats are likely to see even wider applications.

1.3 Background

In August 1991, IBM®[2] and Microsoft Corporation published *Multimedia Programming Interface and Data Specifications, Version 1.0* [1] in which they defined the Resource Interchange File Format (RIFF) multimedia data standard and the Waveform Audio File Format. The Waveform Audio File Format was developed as a standard format for storing digitized audio data in computer files.

---

[1] Microsoft is a registered trademark of Microsoft Corp.

[2] IBM is a registered trademark of IBM Corp.

Subsequently, supplements to [1] were published by Microsoft ([2], [3]). Files containing audio data stored in the Waveform Audio File Format are referred to as WAV or WAVE files (e.g., *mysong.wav*), depending upon vendor and application preference. The WAVE format is a subset of the more general RIFF; these formats are specified in references [1], [2], and [3].

WAVE files are composed of *chunks* that are typically referred to as *WAVE chunks*. For consistency, this document refers to Waveform Audio File Format files as *WAVE files*, the format as *WAVE format*, and the chunks as *WAVE chunks*. However, when researching the topic, the reader should search on both the *WAVE* and *WAV* strings.

1.4 Document Organization

This document describes the objects and syntax of a WAVE file. It refers to these elements as constructs. In addition to describing each relevant object, this document also describes its potential flaws or ways in which data can be hidden, disclosed, or embedded maliciously.

Sanitizing WAVE files poses a unique and complex challenge given that WAVE files contain both structured data (e.g., chunk headers follow a specific format and use standard chunk identifiers) as well as unstructured data (i.e., audio data) of arbitrary length.

The following table summarizes the organization of this document.

Table 1-1. Document Organization

Section	Description
Section 1: Scope	This section describes the scope, organization, and limitations of this document.
Section 2: Construct and Taxonomy	This section describes a definition of how the constructs are represented as well as the terms defined in this document.
Section 3: WAVE File Overview	This section describes the general overview and description of the WAVE file format.
Section 4: WAVE File – Constructs and Metadata	This section describes numerous objects and dictionary information defined in the WAVE file that provide rich media content, interactive features, and embedded content.

1.5 Recommendations

The following subsections summarize the categories of recommendation actions that appear in this document, and associated options.

1.5.1 Actions

Each construct description lists recommended actions for handling the construct when processing a document. Generally, inspection and sanitization programs will perform an action on a construct: *Validate*, *Remove*, *Replace*, *External Filtering Required*, *Review, or Reject*.

The Recommendation section in each construct lists each of these actions and corresponding applicable explanations of the action to take. It notes if a particular action does not apply, indicates actions that are not part of the standard set of actions (listed in the previous paragraph). For example, a program may choose to reject a file if it is encrypted. Additionally, for some constructs, an action may further break down to specific elements of a construct (e.g., for hidden data in a dictionary's syntax) to give administrators the flexibility to handle specific elements differently.

Recommendations such as Remove and Replace alter the contents of the document. WAVE documents are structured such that each object can be identified by a byte offset into the file. There are other fields within various objects that also use byte offset information to identify the location of other objects. If one of these recommendations is taken, the byte offset information throughout the entire WAVE file will require recalculation to correct the structure of the file.

NOTE

The recommendations in this document are brief explanations rather than a How-To Guide. Readers should refer to the construct description or WAVE Specification official documentation for additional details.

Table 1-2 summarizes the recommendation actions.

Table 1-2. Recommendation Actions

Recommendation Action	Comments
Validate	Verify the data structure's integrity, which may include integrity checks on other components in the file. (This should almost always be a recommended action.)
Replace	Replace the data structure, or one or more of its elements, with values that alleviate the risk (e.g., replacing a user name with a non-identifying, harmless value, or substituting a common name for all authors).
Remove	Remove the data structure or one or more of its elements and any other affected areas.
External Filtering Required	Note the data type and pass the data to an external action for handling that data type (e.g., extract text and pass it to a dirty word search).
Review	Present the data structure or its constructs for a human to review. (This should almost always be recommended if the object being inspected can be revised by a human.)
Reject	Reject the WAVE file.

NOTE

No recommendations for logging all actions and found data are included here because all activity logging in a file inspection application should occur "at an appropriate level" and presented in a form that a human can analyze further (e.g., the audit information may be stored in any format but must be parsable and provide enough information to address the issue when presented to a human.)

1.5.2 Action Options

The companion to this document, *Data Transfer Guidance for WAVE Documents*, specifies four options for each recommended action: *Mandatory*, *Recommended*, *Optional*, or *Ignore*. Depending on the circumstances (e.g., a low to high data transfer versus a classified to unclassified transfer), programs can be configured to handle constructs differently.

Table 1-3 summarizes the recommendation action options.

Table 1-3. Recommendation Action Options

Action Options	Comments
Mandatory	For the given direction (e.g., secure private network to unsecure Internet), the file inspection and sanitization program must perform this recommended action.
Recommended	Programs should implement this action if technically feasible.
Optional	Programs may choose to perform or ignore this recommended action.
Ignore	Programs can ignore this construct or data structure entirely.

1.5.3 Naming Convention for Recommendations

Recommendations in this document are numbered sequentially, where applicable, and adhere to a standard naming convention identified by a single number *x*, where *x* is a sequential number following by the recommendation keyword defined in Table 1-2. There may be multiple recommendations of the same type, which remain uniquely identified by its number. There is only one file content under review in this document (WAVE).

1.6 Data Transfer Guidance

Each format that is documented for inspection and sanitization analysis has a companion document (i.e., the aforementioned DTG document). The DTG serves as a checklist for administrators and others to describe expected behaviors for inspection and sanitization programs. For example, administrators may only remove certain values in a metadata dictionary. Or, the administrator may decide to remove all hidden data if the document is being transferred to a lower security domain.

The DTG gives the administrator the flexibility to specify behaviors for inspection and sanitization programs. The workbook contains a worksheet for each security domain (i.e., the originating domain). Each worksheet lists the numbered constructs from this document and enumerated recommendations in a row. After the recommendations, the worksheet displays a cell for each possible destination domain. This enables an administrator to select the action option for data transfer from the originating domain to the particular destination domain. Each construct row also contains two comment cells: one for low to high transfers and another for high to low transfers.

The recommended actions address two broad risk types: data hiding and data execution. Most data structures are vulnerable to one risk type, while others are susceptible to both risk types. Each construct row in the DTG worksheet contains a cell for designating the risk type (i.e., data execution, data hiding, or both) and another cell for assessing the risk level for that construct (i.e., high, medium and low). This enables administrators to assign the risk type and risk level to each specific construct.

1.7 Document Limitations

This document covers information from the IBM and Microsoft standard *Multimedia Programming Interface and Data Specification, Version 1.0*, August, 1991 (ref [1]). At the time of this writing, the latest update was *Microsoft Multimedia Standards Update, Revision 1.0.97*, November, 2006.

1.7.1 Covert Channel Analysis

This document addresses data that can be hidden from normal human review but can be read by a human with a hex viewer or by a machine; this type of method will be called a **Type 1** covert channel. This document will also describe some of the methods for hiding data in a way that is not easily read by a human with a hex viewer or by a machine; methods in this class will be called **Type 2** covert channels. Note that a Type 1 channel is not strictly inferior to a Type 2 channel; for instance, a Type 2 channel using the least significant bits in the image codestream will require special processing to detect the hidden data, but it may introduce visual distortion that is noticeable by normal human review.

This document will provide guidance for the removal or mitigation of Type 1 covert channels in WAV files. It will describe the Type 2 covert channels that are most likely to appear in WAV files and will provide guidance for the mitigation of these channels.

1.7.2 Character Encoding

The WAVE format is a subclass of the RIFF file format: a WAVE file is a RIFF file of type WAVE. RIFF files and WAVE files use the following data formats:

- **Byte Order.** All non-string data values are stored in *little-endian* order[3]. That is, the *least significant* byte occurs first. This is the byte order used on Intel processors.
- **Strings.** Strings of text are used to specify various data parameters (e.g., cue points, notes, and copyright information). Strings are stored in *big-endian* order (i.e., characters are stored left to right). The WAVE format uses various methods of storing strings in WAVE chunk fields. Some strings (aka character strings) are represented as a list of American Standard Code for Information Interchange (ASCII) character bytes where the *first byte* specifies the number of characters in the string. For example, the five-character string **"hello"** is stored as: **5 'h' 'e' 'l' 'l' 'o'**, where the 5 is an integer, not an ASCII character**.** This is the same string format used by the Pascal programming language and is referred to as *byte prefix string* format. Other string values are stored as *null-terminated strings*-that is, a list of ASCII character bytes where the *last byte* is the NULL character (i.e., all zeroes). Some strings are stored using *word prefix string* format—the first word (two bytes) of the string contains the length of the string. Finally, some strings are stored using *byte prefix with null termination* (i.e., the string length is contained in the first byte and the last byte is the NULL character) or *word prefix with null termination* (i.e., the string length is contained in the first word and the last byte is the NULL character). It is important to determine which string format is used for each type of data parameter field.
- **Byte Size.** A byte consists of 8 bits of data.
- **Word Size.** A word consists of 16 bits of data.
- **Double-Word Size.** A double-word consists of 32 bits of data.

[3] Compared to *Big-Endian* order where the *most significant* byte is stored first.

2. CONSTRUCTS AND TAXONOMY

2.1 Constructs

Although this document delves into many low level constructs in the WAVE file format, it does not serve as a complete reference to all of the different constructs in the standard. The document covers the overall file format, including the header and audio data. We have identified the following as particular areas of concern for developers of file inspection and sanitization programs; however there is complete detail in the standard that should be examined alongside this documentation.

- **Description**: a high level explanation of the data structure or element
- **Concern**: an explanation of potential problems posed by the element. For example, some metadata elements can cause inadvertent data leakage and others can be used for data exfiltration.
- **Location**: provides a textual description of where to find the element in the document. This can vary by client (or product) or it may apply across the entire product line.
- **Examples**: if applicable, the definition will contain an example of the construct.
- **Recommendations**: as described in Section 2.2.

Recommendations appear within each of the WAVE constructs. For the purposes of this document, these recommendations are "alternatives." Some recommendations may seem better than others and some recommendations may be more difficult to implement. Certain recommendations complement each other and can be grouped together (e.g., "Remove action object" and "Remove reference to action object"). Other recommendations may seem contradictory (e.g., "Remove Metadata" and "Replace Metadata").

2.2 Taxonomy

The following table describes the terms that appear in this document.

Table 2-1. Document Taxonomy

Term	Definition
Consistency	A construct state in which object information is set to correct values, and that required objects are implemented as defined in the WAVE specification.
Construct	An object in WAVE terminology that represents some form of information or data in the hierarchy of the WAVE document structure.
DTG	A list of all ISG constructs and their associated recommendations. DTGs are used to define policies for handling every ISG construct when performing inspection and sanitization.
Grammar	A precise, formal, and rigorous syntax for defining constructs.
Inspection and Sanitization	Activities for processing files to prevent inadvertent data leakage, data exfiltration, and malicious data or code transmission.
ISG	A document (such as this) that details a file format or protocol and inspection and sanitization activities for constructs within that file format.
Recommendations	A series of actions for handling a construct when performing inspection and sanitization activities.
Referential Integrity	The construct state in which all associated objects are properly referenced in the construct and that construct entries reference existing objects. References may also point to byte offset location into the WAVE file as opposed to object number. If this isn't the case the document may not open or may break.

3. WAVE OVERVIEW

In August 1991, IBM and Microsoft Corporations published *Multimedia Programming Interface and Data Specifications, Version 1.0* ([1]) in which the Resource Interchange File Format (RIFF) multimedia data standard and the Waveform Audio File Format were defined. The Waveform Audio File Format was developed as a standard format for storing digitized audio data in computer files. Subsequently, supplements to [1] were published by Microsoft ([2], [3]). Files containing audio data stored in the Waveform Audio File Format are referred to as WAV or WAVE files (e.g., *mysong.wav*), depending upon vendor and application preference. The WAVE format is a subset of the more general RIFF; these formats are specified in references [1], [2], and [3]. WAVE files are composed of *chunks* that are typically referred to as *WAVE chunks*. For consistency, this document refers to Waveform Audio File Format files as *WAVE files*, the format as *WAVE format*, and the chunks as *WAVE chunks*.

Before describing the WAVE format, it is helpful to understand RIFF. The RIFF specification ([1]) uses building blocks called *chunks* to structure a file. A single chunk is a logical unit of multimedia data that cannot exceed four gigabytes (4 GB) because of its use of a 32-bit unsigned integer to record the file size header Each RIFF chunk is composed of a *header* (consisting of a *chunk identifier* and the *size* of the chunk data) followed by the *chunk data*. The first four bytes of the chunk data specify the *form type* identifier, which describes the type of data contained in the chunk. A RIFF file is composed of a single RIFF chunk; however, the RIFF chunk's data field may contain one or more subchunks as shown in Figure 3-1. When a RIFF chunk has a form type of **WAVE**, the RIFF chunk is expected to contain Waveform audio data. Thus, a WAVE file is actually a RIFF file where the RIFF data's form type is **WAVE** as shown in Figure 3-2.

Table 3-1 lists the various RIFF form types that are currently registered. RIFF files are usually referred to by the format of the data that they contain. For example, a RIFF file containing audio samples has a form type of WAVE and is called a WAVE file (*.wav*), while a RIFF file that contains Audio Visual Interleaved (AVI) form data is called an AVI file (*.avi*).

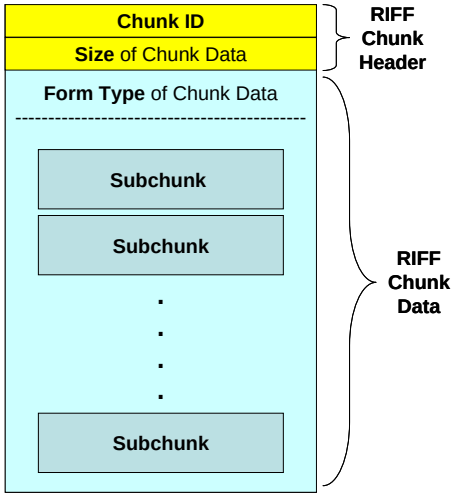

Figure 3-1. RIFF Chunk Layout

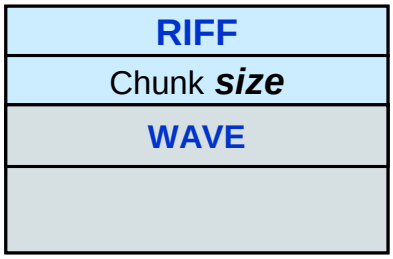

Figure 3-2. RIFF WAVE Chunk Layout

Table 3-1. RIFF Data Form Types

RIFF Form Type	Data Description
ACON	Windows®[4] NT Animated Cursor Format
AVI_[5]	Audio Visual Interleaved Format
PAL_	Palette File Format
CPPO	APPS Software International C++ Object Format
RDIB	Device Independent Bitmap Format
RMID	MIDI Audio Format
RMMP	Multimedia Movie File Format
WAVE	Waveform Audio Format

The RIFF chunk is the only mandatory chunk type—it must be present in a RIFF file. All other RIFF chunk types are optional. It is common for a RIFF file to contain multiple nested subchunks. In general, an outer chunk is referred to as a *parent chunk* and the nested inner chunks are referred to as *child chunks* or *subchunks*. The first chunk in a

[4] Windows is a registered trademark of Microsoft Corp.

[5] The actual RIFF Form type is "**AVI** "; the space is replaced with an underscore (_) in this document for clarity.

RIFF file must be a RIFF chunk; all other chunks in the file must be subchunks of the RIFF chunk[6]. Only RIFF chunks and LIST chunks may contain subchunks.

A valid WAVE file is composed of a single RIFF chunk that contains two or more WAVE chunks in the RIFF chunk data field. (The WAVE chunks are child subchunks of the RIFF chunk.) A WAVE chunk uses the same format as a RIFF chunk: each WAVE chunk begins with a *header* (a WAVE *chunk identifier* and the *size* of the WAVE chunk data) followed by the WAVE *chunk data* (and, as stated above, the first 4-byte field of the WAVE chunk data is a *form type* identifier that indicates the type of WAVE data).

Several types of WAVE chunks have been defined. Some WAVE chunks contain the actual audio data samples, while other WAVE chunks convey descriptive as well as vital information (e.g., audio configuration data and parameters) about the contents and playback of the audio data in the file. WAVE chunks are discussed further in Section 4.

Table 3-2 lists some of the types of RIFF chunks that may be present in a WAVE file. For example, a LIST chunk typically appears as a subchunk of a RIFF chunk; however, LIST chunks may also contain subchunks (see Figure 3-3). There are many different types of LIST chunks; those that are relevant to WAVE files will be discussed in Section 4.

Table 3-2. RIFF Chunk Types That May Appear in WAVE Files

RIFF Chunks	Chunk ID	Description
RIFF	**RIFF**	Basic building block of a RIFF file
LIST	**LIST**	Contains a list or ordered sequence of subchunks
Character Set	**CSET**	Defines character-set and language information
Junk Chunk	**JUNK**	Contains chunk padding, filler or outdated information.
Display Chunk	**DISP**	Contains easily rendered and displayable objects.
Pad Chunk	**PAD_**	Contains padding with no relevant data. Similar to a Junk chunk.

[6] See http://msdn2.microsoft.com/en-us/library/ms713231(printer).aspx.

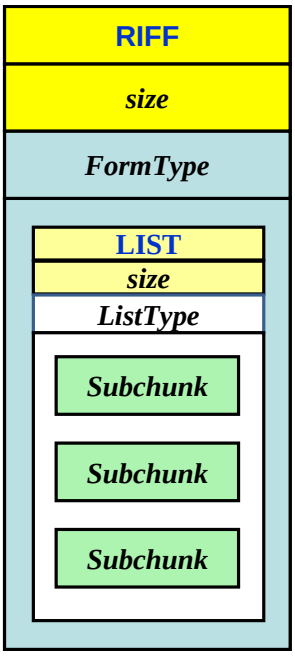

Figure 3-3. LIST Chunk Embedded Within a RIFF Chunk[7]

One of the features developers like about WAVE files is their extensibility. The WAVE file format is written such that additional custom chunk data fields and/or additional chunks can be added with minimal impact on existing data. In accordance with the initial WAVE format specification [1], WAVE applications must anticipate and skip any unknown chunk types; this can be easily accomplished due to the simplicity of the WAVE chunk format. Unfortunately, this capability enables the abuse and misuse of the WAVE format, which can result in potential security vulnerabilities (e.g., the insertion of non-audio-related data or malware, audio data format errors that can result in buffer overflows). Developers can also submit different or custom waveform encoding schemes. To date, over 200 encoding schemes have been included in the WAVE file format.

3.1 RIFF File Format Overview

As noted in Section 3 (see Figure 3-1), a RIFF file is composed of one RIFF chunk. All other chunks in the file are subchunks within the RIFF chunk (see Figure 3-4). Chunk

[7] See Resource Interchange File Format Services, available at http://msdn.microsoft.com/en-us/library/ms713231.aspx.

IDs that are defined in the RIFF specification are always capitalized (e.g., **RIFF, LIST**). Chunk IDs that are not RIFF-specific are always lowercase (e.g., **data**, **note**). Except for **LIST** chunks, a subchunk has a lowercase chunk ID because the subchunk's significance and use are only defined within the context of its specific parent chunk.

A **LIST** chunk is a type of RIFF subchunk that can also contain subchunks; recall that only **RIFF** chunks and **LIST** chunks may contain subchunks.
A RIFF chunk consists of the following fields:

- A 4-byte ASCII-character chunk identifier (denoted generically as the **<Chunk ID>** or ***ChunkID***)
- A 32-bit (i.e., 4-byte) unsigned integer value[8] specifying the number of bytes in the data field(s) of the chunk (denoted generically as the **<size>** or ***size***)
- One or more RIFF data fields (the data fields together are denoted generically as the **<chunk data>** or ***Data***), where:
    - The RIFF chunk data must begin with a 4-byte ASCII-character form type (denoted generically as the ***FormType***); the form type specifies the format of the data stored in the RIFF chunk.
    - The remaining RIFF **<chunk data>** consists of multimedia data formatted according to the ***FormType***.
- If the RIFF **<chunk data>** does not end at the end of a 16-bit word boundary (i.e., the ***size*** is an odd number), then a single NULL character byte must be added at the end of the **<chunk data>** to pad the data to the end of a 16-bit word boundary.

The RIFF format is hierarchical. That is, the **<chunk data>** may contain one or more chunks, often referred to as *subchunks* of the *parent* RIFF chunk. A RIFF subchunk is also composed of a ***ChunkID***, ***size***, and **<chunk data>** (see Figure 3-4). The ***ChunkID*** and the ***size*** fields together are often referred to as the *chunk header*. The chunk header field must always appear prior to the **<chunk data>** field.
It is important to understand that the ***size*** value does not include the 8 bytes in the chunk header (namely the 4-byte ***ChunkID*** and the 4-byte ***size*** fields) nor does the ***size*** value include the optional single NULL character byte that may be padded at the end of the **<chunk data>** field to round it to the end of a 16-bit word boundary.

---

8 The ***size*** value is either zero or a positive number.

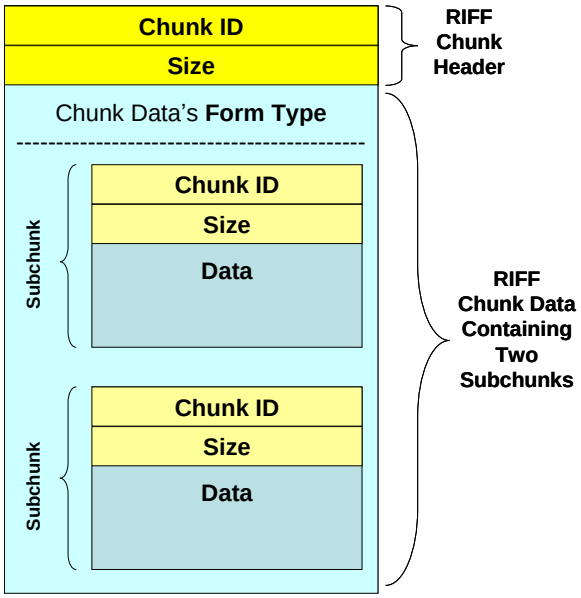

Figure 3-4. A Generic RIFF Chunk Containing Two Subchunks

Thus the true actual length of a RIFF chunk equals the size of the header (8 bytes) plus the ***size*** of the **<chunk data>**, rounded up the next even number (to account for the single NULL character pad when it is present). That is:

- IF **size** is even, THEN **Actual_chunk_length** = 8 + **size**.
- ELSE IF ***size*** is odd, THEN **Actual_chunk_length** = 8 + **size** +1.

Furthermore, following the end of the RIFF chunk (the End-of-File [EOF] marker), there may be optional unused space (aka file padding) to fill out the file to the End-of-Cluster [EOC] marker (see Figure 3-5).

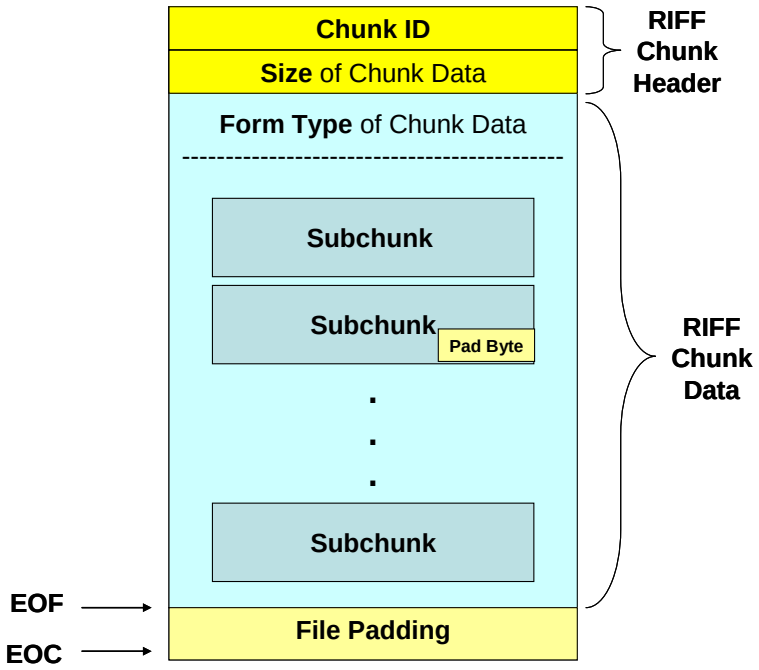

Figure 3-5. A Generic RIFF File with a Subchunk Pad Byte and EOF Padding

Every RIFF chunk MUST be aligned to a 16-bit word boundary; however, the chunk layout figures in this report are depicted along 32-bit double-word boundaries, as is typical in the RIFF specification, for ease in understanding. It is important to recognize that real RIFF chunks may not be aligned to double-word boundaries by happenstance and therefore they will appear differently in a memory dump. This report is focused on WAVE files and WAVE chunks. Section 3.2 presents an overview of the WAVE file format; Section 4 discusses WAVE constructs in more detail.

3.2 WAVE File Format Overview

A WAVE file is a type of RIFF file where the RIFF chunk's form type is **WAVE** and the RIFF chunk's data area contains WAVE subchunks. Thus, as discussed in Section 3.1, a WAVE file begins with a **RIFF** chunk:

- The 4-byte ***ChunkID* = RIFF**
- A 4-byte ***size*** of the RIFF chunk data field (denoting the number of bytes)
- Followed by the RIFF **<chunk data>,** where

- The 4-byte ***FormType*** = **WAVE**
- The remaining RIFF chunk data consists of two or more WAVE chunks (that are also RIFF subchunks)

This high-level view of a WAVE file was depicted in Figure 3-1. Since a WAVE chunk is a subchunk of a RIFF chunk, the format of a WAVE chunk is the same as the format of a RIFF chunk. Each WAVE chunk is composed of a ***ChunkID***, the ***size***, and the **<chunk data>**. The ***ChunkID*** is represented as a 4-byte character array which identifies the type of WAVE chunk. The chunk ***size*** is represented as a 4-byte unsigned 32-bit integer which specifies the length (i.e., number of bytes) of the WAVE **<chunk data>**.

As in the RIFF format, if the WAVE chunk does not naturally terminate at a 16-bit word boundary, a single NULL pad byte is added to the end of the chunk data to align it to a 16-bit word boundary. As in the RIFF format, the WAVE ***size*** does not include the length of the 8-byte WAVE chunk header (i.e., 4-byte ***ChunkID*** and 4-byte ***size*** fields) nor does it include the pad byte when it is present. Thus, the true actual length of a WAVE chunk equals the length of the header (8 bytes) plus the ***size*** of the WAVE **<chunk data>**, rounded up an even number (to include the 1-byte NULL character pad when it is present). That is:

- IF ***size*** is even, THEN ***Actual_chunk_length*** = 8 + ***size***.
- ELSE IF ***size*** is odd, THEN ***Actual_chunk_length*** = 8 + ***size*** +1.

Note that the operating system divides files into clusters and there may be additional bytes of unused space between the EOF marker and the EOC marker in the actual file.

Audio data in WAVE files can be stored in numerous different formats; the format and parameters are specified using various types of WAVE chunks. As a minimum, a WAVE file MUST contain the following two WAVE chunks:

- A format (i.e., **fmt_)** chunk that contains information used to play the digital audio data in the WAVE data chunk. (**Note:** The format chunk ID consists of the three lowercase letters **fmt** followed by a space [“**fmt** ”]; for clarity, this document will depict the format chunk ID as **fmt**_, using the underscore to depict the space.)
- A **data** chunk that contains the actual digital audio data.

Figure 3-6 depicts a WAVE file that consists of only the two required subchunks, where the audio data is encoded in the PCM format.

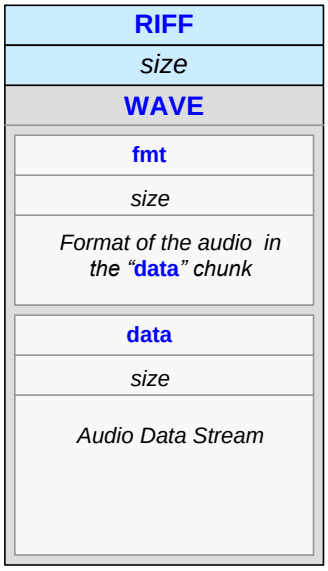

Figure 3-6. A WAVE File Containing Only the Two Required WAVE Chunks

A third chunk, the **fact** chunk, is also required under the following conditions:

- If the audio data is NOT encoded using PCM, then a **fact** chunk is required[9].
- If the audio data is stored within a *wave-LIST* chunk instead of as a primary **data** chunk, then a **fact** chunk is required. A wave-LIST chunk is a **LIST** chunk that has a form type of **wavl**, denoted as **LIST(wavl)**[10]. (**Note:** A wave-LIST chunk increases the complexity of the WAVE file and is not supported by many applications. While it is recommended that wave-LIST chunks should not be used when creating WAVE files, analysis software must be capable of recognizing and processing wave-LIST chunks.)

9 If the audio data uses PCM encoding, then the **fact** chunk is not needed.

10 WAVE-list chunks are discussed later in the document.

The **fmt_** chunk and **fact** chunk (when present) MUST appear before the **data** chunk. In addition to the required WAVE chunks, a WAVE file may contain one or more optional chunks which provide additional information (e.g., cue points, a playlist, copyright, license, and author). The optional WAVE chunks may appear either before or after the **data** chunk. Figure 3-7 depicts a WAVE file that consists of three required WAVE chunks plus additional optional WAVE chunks. Figure 3-8 depicts a WAVE file that has padding between the EOF and EOC.

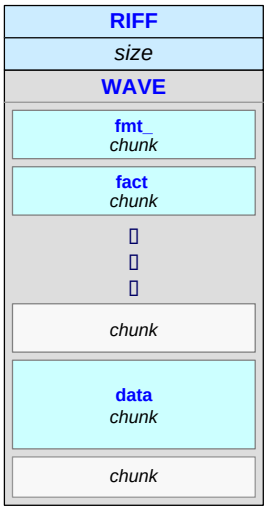

Figure 3-7. A WAVE File That Includes a Fact Chunk and Optional Chunks

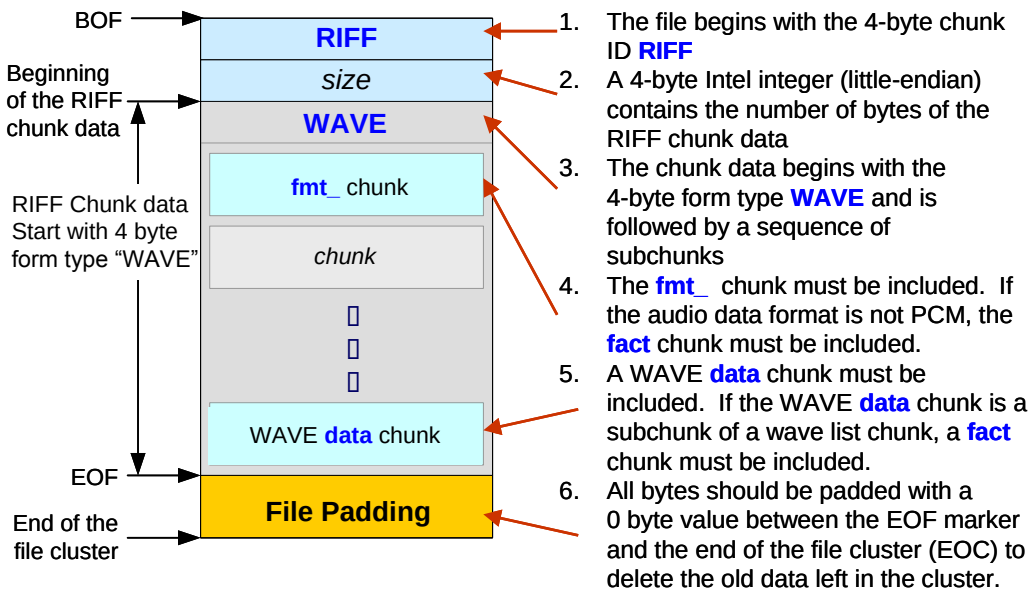

Figure 3-8. Layout of WAVE Chunk with End-of-File Padding

The WAVE format defines nine (9) primary chunk types; in addition, two of the primary chunk types have one or more subchunks contained within them. There are five (5) subchunk types defined. Each WAVE chunk and subchunk type is specified by a unique 4-byte ***ChunkID***. Table 3-3 contains a list of the WAVE chunk types in the first column of the table: the *primary* WAVE chunk types are left-aligned and the *subchunk* types are asterisked (*) and right-aligned below their associated primary chunk type.

Recall that upper case ***ChunkID*** tags represent global RIFF-defined chunk types (e.g., **RIFF** and **LIST**). All of the WAVE format-defined ***ChunkID*** tags consist of lower case characters (e.g., **fact**, **data**).

Table 3-3. WAVE Chunk Types

WAVE Chunks	Chunk ID	Required?	Description
Format Chunk	**fmt_**	Yes. (**Note:** The chunk ID ends with a space character 0x20.)	Contains information to interpret the audio data in the chunk data.
Fact Chunk	**fact**	Required for all compressed WAVE data. Required if the	Contains compression codec-dependent information.

WAVE Chunks	Chunk ID	Required?	Description
		WAVE data is contained in a wave-LIST chunk.	
Data Chunk	**data**	Yes. (May exist as a wave-LIST subchunk, but not recommended.)	Contains the audio data stream. Must be present either as a primary chunk (safer) or as a subchunk in a wave-LIST chunk.
Wave-LIST Chunk	**LIST** (with form type **wavl**)	Optional - A **LIST** chunk with a list type **wavl**. Either a **data** chunk or a wave-LIST chunk MUST be present. Also contains a **data** subchunk.	Contains several alternating **slnt** and **data** chunks. Helps reduce WAVE file size. Increases complexity of the file structure. Many programs do not recognize this chunk and ignore it. Can be easily abused by programmers. Recommend use of other compression formats to reduce file size.
*** Data Subchunk**	**data**	See **data** chunk above	Contains the audio data stream.
*** Silent Subchunk**	**slnt**	Optional - a subchunk contained within a wave-LIST chunk.	Specifies the duration of a silent segment. The silent sample value is the last sample value in the preceding data chunk within the wave-LIST chunk not the 0 value).
Cue Points Chunk	**cue_**	Optional. (**Note:** The chunk ID ends with a space character 0x20.)	Specifies one or more sample offsets which are used to mark sections of the audio. If it is included, a single cue chunk should be used to specify all cue points.
Playlist Chunk	**plst**	Optional.	Specifies the play order of a series of cue points.
Associated-Data- List Chunk	**LIST** (with form type **adtl**)	Optional - a **LIST** chunk with a list type **adtl**.	Defines text labels and names which are associated with each cue point. Contains a list of labels, notes, and/or labeled text chunks.

WAVE Chunks	Chunk ID	Required?	Description
*** Label Subchunk**	**labl**	Optional - a subchunk contained within an associated-data-list chunk.	Associates a text label with a cue point.
*** Note Subchunk**	**note**	Optional - a subchunk contained within an associated-data-list chunk.	Associates a text comment with a cue point.
*** Text with Data Length Subchunk**	**ltxt**	Optional - a subchunk contained within an associated-data-list chunk.	Associates a text label with a section of waveform data.
*** Embedded File Information Subchunk**	**file**	Optional - - a subchunk contained within an associated-data-list chunk.	Contains information described in other file formats.
Instrument Chunk	**inst**	Optional.	Contains some of the same type of information as the sampler chunk. Describes how the waveform data should be played as an instrument sound.
Sample Chunk	**smpl**	Optional.	Contains parameters that a player (e.g., a Musical Instrument Digital Interface [MIDI] sampler) can use to play a WAVE file as an instrument.

As shown in Table 3-3, a wave-LIST chunk (i.e., **LIST** with form type **wavl**) may contain silent subchunks (i.e., **slnt**) and data subchunks (i.e., **data**); an associated-data-list chunk (i.e., **LIST** with form type **adtl**) may contain label (i.e., **labl**) subchunks, note subchunks (i.e., **note**), and labeled text subchunks (i.e., **ltxt**). The WAVE chunks and subchunks are discussed in more detail in Section 4.

4. CONSTRUCTS AND METADATA

4.1 WAVE Header

WAVE 4.1: WAVE FILE HEADER

OVERVIEW:

WAVE files are RIFF files containing a single RIFF chunk with a form type of WAVE. Using this information, the WAVE file header can be identified by analyzing the first 12 bytes of the file.

CONCERNS:

Data attack - Format errors in the WAVE file header can cause buffer overflow and/or denial-of-service (DoS) attacks in WAVE applications. Attackers can create WAVE files containing incorrect formats and embed malicious executable code to exploit vulnerabilities in audio file applications.

PRODUCT: WAVE

LOCATION:

WAVE header in the first 12 bytes of the file.

RECOMMENDATIONS:

1 **Validate**: Check that ***ChunkID*** is **RIFF** in bytes 1-4.

2 **Validate**: Check that ***Size*** is equal to 8 less than the byte-length of the file in bytes 5-8.

3 **Validate:** Check that ***FormType*** is **WAVE** in bytes 9-12.

4 **Remove**: N/A

5 **Replace**: If ***Size*** is incorrect, replace it with 8 less than the byte-length of the file.

6 **External Filtering Required**: N/A

7 **Review**: N/A

8 **Reject**: N/A

WAVE 4.1: END

4.2 Format Chunk

WAVE 4.2: FORMAT CHUNK COMPRESSION CODE

OVERVIEW:
The format chunk data field is composed of a fixed-length section followed by a variable-length section containing format parameters specific to the compression code given in the fixed area. The format chunk compression code indicates the type of compression used for the audio data. The most common is PCM (***CompressionCode*** = 1). There are over 200 ***CompressionCode*** values representing different compression types.

CONCERNS:
Data attack - Setting the *CompressionCode* to an invalid type can cause certain WAVE players to fail, enabling buffer overflow and/or denial-of-service attacks in WAVE applications.

PRODUCT: WAVE

LOCATION:
Format chunk

EXAMPLE:
Figure 4-2 shows the layout of the fixed-length portion of the format chunk.

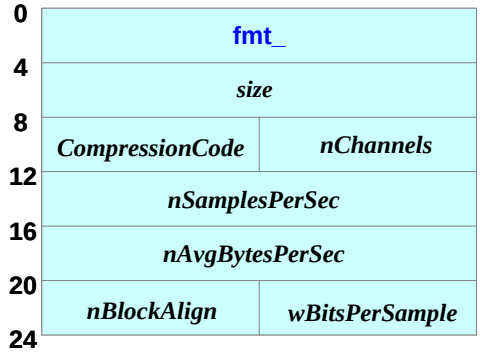

Figure 4-1. Layout of Fixed-Length Portion of Format Chunk

RECOMMENDATIONS:
1 **Validate**: Validate the two-byte ***CompressionCode*** against the list of allowable entries (e.g., http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html)

2 **Validate: If the compression format is not PCM, verify that a Fact chunk exists**

3 **Remove: N/A**

4 **Replace**: N/A

5 **External Filtering Required: N/A**

6 **Review: N/A**

7 **Reject: N/A**

WAVE 4.2: END

WAVE 4.3: FORMAT CHUNK NUMBER OF CHANNELS

OVERVIEW:

The two-byte format chunk parameter *nChannels* specifies the number of audio channels contained in the WAVE file. Since *nChannels* is a two-byte parameter, as many as 65536 channels are permitted in a WAVE file.

CONCERNS:

Data Hiding - *nChannels* is typically equal to 1 for mono, or 2 for stereo. Many WAVE applications will not accept a WAVE file with more than two channels. A value of *nChannels* greater than 2 suggests the possibility of hiding data in one or more channels.

PRODUCT: WAVE

LOCATION:

Format chunk

RECOMMENDATIONS:

1 **Validate**: Verify that the value of ***nChannels*** is between 1-65536

2 **Validate:** Verify that the value of *nChannels* is less than some predefined value or threshold.

3 **Remove**: Remove all channels above predefined value or threshold.

4 **Replace**: N/A

5 **External Filtering Required**: N/A

6 **Review**: N/A

WAVE 4.3: END

WAVE 4.4: FORMAT CHUNK PARAMETER CONSISTENCY

OVERVIEW:

Parameters specified in the format chunk include:

- ***nChannels***: number of channels
- ***nSamplesPerSec***: Sampling rate
- ***BitsPerSample***: number of bits per sample
- ***nAvgBytesPerSec***: Byte rate
- ***nBlockAlign***: number of bytes per sample including all channels

These parameters, along with the ***CompressionCode***, are used by applications to specify how to read the audio data in the data chunk. For PCM data,

nAvgBytesPerSec = nSamplesPerSec * nChannels * (BitsPerSample/8)

nBlockAlign = nChannels * (BitsPerSample/8)

CONCERNS:

Data attack - The relations among these five parameters should be checked for consistency. Inconsistent values for format chunk parameters can cause certain WAVE applications to fail, enabling buffer overflow or DoS attacks.

PRODUCT: WAVE

LOCATION:

Format chunk

RECOMMENDATIONS:

1 **Validate**: Validate consistency of format chunk parameters.

2 **Remove**: N/A

3 **Replace**: N/A

4 **External Filtering Required**: N/A

5 **Review**: N/A

6 **Reject**: N/A

WAVE 4.4: END

WAVE 4.5: FORMAT CHUNK QUANTIZATION

OVERVIEW:

The format chunk parameter ***BitsPerSample*** specifies the quantization of the audio data samples. ***BitsPerSample*** values are rounded up to the nearest multiple of 8. For example,

A value of 9 bits per sample would be rounded to 16 bits, leaving 7 unused bits per audio sample.

CONCERNS:

Data Hiding - ***BitsPerSample*** values are rounded up to the nearest multiple of 8. If the value of ***BitsPerSample*** is not a multiple of 8, a WAVE file could contain embedded data in the unused bits.

PRODUCT: WAVE

LOCATION:

Format chunk

RECOMMENDATIONS:

1 **Validate**: Validate consistency of format chunk parametersIf ***BitsPerSample*** is not a multiple of 8, verify the unused bits in the audio samples.

2 **Remove**: N/A

3 **Replace**: If ***BitsPerSample*** is not a multiple of 8 and an audio sample contains non-zero bit values beyond ***BitsPerSample***zero the bits up to the next multiple of 8.

4 **External Filtering Required**: N/A

5 **Review**: N/A

6 **Reject**: If the number of unused bits in the audio samples is above a preset threshold, reject the file.

WAVE 4.5: END

4.3 Fact Chunk

WAVE 4.6: FACT CHUNK PARAMETERS

OVERVIEW:
When the Fact chunk is present, bytes 9-12 contain the length of the audio data in bytes (***dwSampleLength***)

CONCERNS:
Data attack - Setting the number of samples to an incorrect value can result in buffer overflow and/or denial-of-service attacks in WAVE applications.

PRODUCT: WAVE

LOCATION:
Fact chunk

EXAMPLE:
Figure 4-3 shows the layout of the fact chunk.

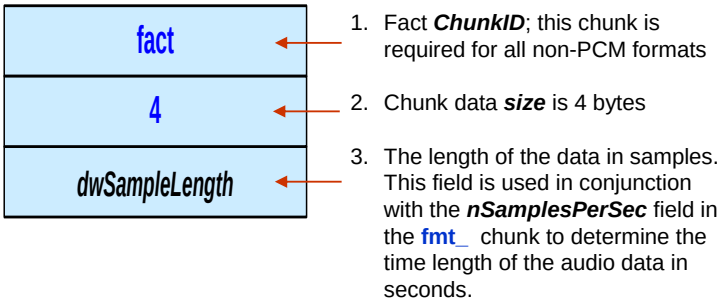

Figure 4-2. Layout of the Fact Chunk

RECOMMENDATIONS:

1 **Validate**: Verify that ***dwSampleLength*** is the same as the sample-length of the data chunk.

2 **Remove**: N/A

3 **Replace**: If the sample-length of the data chunk is not equal to ***dwSampleLength***, replace ***dwSampleLength*** with this value.

4 **External Filtering Required**: N/A

5 **Review**: N/A

6 **Reject**: N/A

WAVE 4.6: END

4.4 Data Chunk

WAVE 4.7: DATA CHUNK PARAMETER CONSISTENCY

OVERVIEW:
The **data** chunk may be present as a primary chunk (the most common occurrence) or as a subchunk to a wave-LIST chunk (which is less common). The **data** chunk's complexity arises regarding how an application reads the data along with its dependency on the format chunk. The data section is governed by several of the format chunk fields (e.g., compression code, number of channels, and bits per sample). Thus a **data** chunk cannot exist without a preceding **fmt**_ chunk. Each sample point of data is stored in little-endian, time-sequenced order, starting with the oldest time and progressing forward. An instance in time is composed of N samples, where N is the number of channels. A sample is composed of P bits, where P is the bits per sample. The P bits are the result of the given compression scheme.

CONCERNS:
Data attack - Format errors in the data chunk can cause buffer overflow and/or denial-of-service (DoS) attacks in WAVE applications. Attackers can create WAVE files containing incorrect formats and embed malicious executable code to exploit vulnerabilities in audio file processing applications.

PRODUCT: WAVE

LOCATION:
Data chunk

EXAMPLE:

Figure 4-4 shows a graphical representation of the data chunk layout for mono and stereo channels. The data chunk on the left depicts a mono channel containing ***n*** samples of audio data. The data chunk on the right depicts a stereo channel containing ***n*** stereo samples, where each stereo sample consists of two mono samples (i.e., a left-channel audio sample and a right-channel audio sample). If the number of significant bits per sample is not a multiple of 8, then each sample point must be rounded up to a size that IS a multiple of 8 and the data bits must be left justified and padded with 0 bits. WAVE applications interpret samples sizes of less than 8 bits as unsigned integers (which range from 0—255) and sample sizes of 9 or greater as signed integers (e.g., 16 bit samples range from -32768 through +32767).

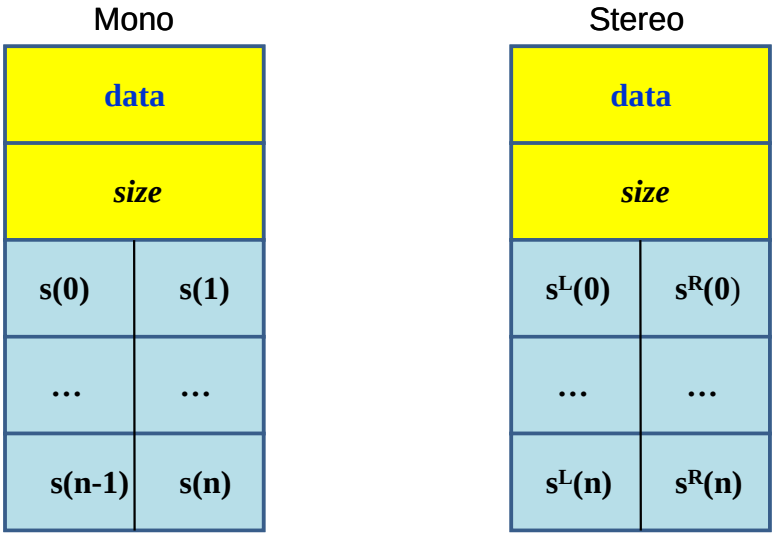

Figure 4-3 Data Chunk Layouts for Mono and Stereo Channels

RECOMMENDATIONS:

1 **Validate**: Verify that ***ChunkID*** is **data**.

2 **Validate**: Verify that ***Size*** is equal to the byte-length of the audio data samples.

3 **Validate**: Verify that the Data chunk is preceded by the Format chunk.

4 **Remove**: N/A

5 **Replace:** N/A

6 **External Filtering Required**: N/A

7 **Review**: N/A

8 **Reject**: If ***ChunkID*** is not **data**, reject the WAVE file.

WAVE 4.7: END

WAVE 4.8: DATA CHUNK AUDIO SAMPLES

OVERVIEW:

The ***data*** field in the data chunk contains the actual audio samples.

CONCERNS:

Data attack – It is possible for attackers to insert executable code in the data chunk.

Data hiding - The data field may contain steganographic material: hidden data such as text, images and other media. Although no evaluation tool can detect every possible instance of steganography, a cursory analysis of bit statistics can reveal the presence of hidden data [10, 11].

PRODUCT: WAVE

LOCATION:

Data Chunk

EXAMPLE:

Figure 4-5 shows computed bit statistics for 16-bit PCM data including samples of speech, music, and white noise. Note that the statistics for several most significant bits of speech and music exhibit significant deviation from 0.5.

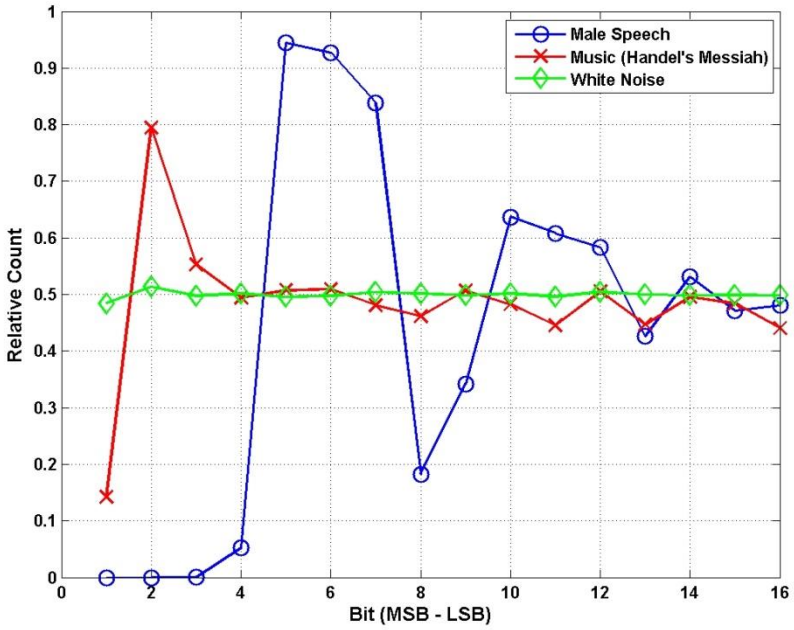

Figure 4-4 Bit Statistics for Speech, Music, and White Noise

RECOMMENDATIONS:

1 **Validate**: Use software filter to flag non-audio by comparing computed bit statistics with typical audio bit statistics.

2 **Remove**: N/A

3 **Replace**: Replace the least significant bit with a predefined value.

4 **Replace**: Replace the audio data with a version that has had a bitmask applied to the least significant bits in a manner that can't be reversed easily.

5 **External Filtering Required**: N/A

6 **Review**: N/A

7 **Reject**: N/A

WAVE 4.8: END

4.5 Optional Content

WAVE 4.9: WAVE LIST CHUNK

OVERVIEW:

The wave-LIST chunk is a type of **LIST** chunk with form type of **wavl**. Recall that **LIST** chunks can contain subchunks. The wave-LIST chunk is used to help reduce the WAVE file size where there are several periods of silence (or long periods of silence). However, this chunk type is not popular and many applications do not process (i.e., ignore or skip over) wave-LIST chunks[11]. When a WAVE file does not contain a primary **data** chunk, a wave-LIST chunk that contains one or more **data** subchunks MUST be present.

The wave-LIST chunk contains a list of alternating **data** and silent (**slnt**) subchunks. Either the **data** subchunk or **slnt** subchunk may appear first. There may be more than one pair of **data** and **slnt** subchunks. For example, a wave-LIST chunk that has four pairs of **data** and **slnt** subchunks will have the subchunks appear in order as either:

- **data, slnt, data, slnt, data, slnt, data, slnt,** or
- **slnt, data, slnt, data, slnt, data, slnt, data**

Figure 4-6 illustrates the layout of the wave-list chunk.

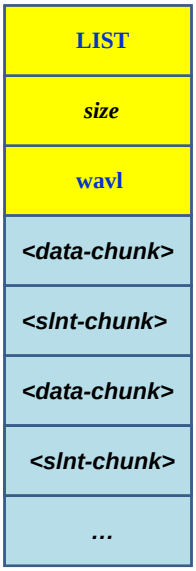

Figure 4-5 Wave-LIST Chunk Layout

[11] See http://www.sonicspot.com/guide/wavefiles.html#list.

CONCERNS:
Data Attack - Setting the duration in the **slnt** chunk to a bogus number (e.g., extremely large) could cause playback of the file to take an extremely long time and result in a DoS attack.

PRODUCT: WAVE

LOCATION:
wave-LIST Chunk

RECOMMENDATIONS:

1 **Validate**: Verify that ***ChunkID*** is **LIST**.

2 **Validate**: Verify that ***FormType*** is **wavl**.

3 **Validate**: If a primary data chunk is not present, verify that at least one data chunk is present in the wave-LIST chunk.

4 **Validate**: If at least one silent chunk is present, verify that the data and silent chunks alternate.

5 **Remove**: N/A

6 **Replace**: N/A

7 **External Filtering Required**: N/A

8 **Review**: N/A

9 **Reject**: If neither a primary data chunk nor a wave-LIST data chunk is present, reject the file.

10 **Reject**: If the data and silent chunks do not alternate, reject the file.

WAVE 4.9: END

WAVE 4.10: SILENT CHUNK

OVERVIEW:
The main function of the **slnt** subchunk type is to specify periods of silence measured in samples. **Slnt** chunks are only contained in **wave-LIST** chunks where they are typically used to reduce file sizes. To reduce the file size, **slnt** chunks are used to represent periods of silence. During this silence period, the last sample's audio value is held. The last sample's value could be either zero

or nonzero. In cases where no **data** subchunk precedes the **slnt** chunk, a sample value of 127 is used for 8-bit samples and a sample value of 0 is used for 16-bit samples[12] and this sample value (either 0 or 127) is held during the silence period. (**Note:** The Waveform specifications do not identify values for 24-bit samples.) Figure 4-7 illustrates the layout of the silent chunk.

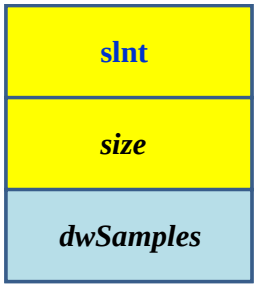

Figure 4-6. Silent Subchunk Layout

CONCERNS:
Data Attack - Setting of the silent duration to a bogus number (e.g., extremely large) could cause playback of the file to take an extremely long time and result in a DoS attack.

PRODUCT: WAVE

LOCATION:
Silent chunk

RECOMMENDATIONS:

1. **Validate**: Verify that the Chunk ID is **slnt.**

2. **Validate**: Check the silence length ***dwSamples*** parameter. If a **fact** chunk is present, verify that ***dwSamples*** is less than the **fact** chunk's ***dwSampleLength***. If no **fact** chunk is present, verify that ***dwSamples*** is less than ***size****8/(***nSamplesPerSec****wBitsperSample***), where:

   - ***size*** is the size of the **data** chunk in bytes.
   - ***nSamplesPerSec*** is the sampling rate in the **fmt**_ chunk
   - ***wBitsperSample*** is the sample size from the **fmt**_ chunk

3. **Remove**: N/A

[12] The **slnt** value uses the mid-point value. Eight-bit samples have values ranging from 0-255; 127 is the midpoint value. Sixteen-bit samples are treated as signed numbers; 0 is the midpoint value.

4. **Replace**: Replace ***dwSamples*** with a value less than ***dwSampleLength***.

5. **External Filtering Required**: N/A

6. **Review**: N/A

7. **Reject**: N/A

WAVE 4.10: END

WAVE 4.11: CUE CHUNK

OVERVIEW:

A Cue chunk specifies one or more sample offsets which are often used to mark noteworthy sections of audio. For example, the beginning and end of a verse in a song may have cue points to make them easier to find. In conjunction with the playlist chunk, the cue chunk can be used to store looping information. The cue chunk is optional and if included, a single cue chunk should specify all cue points for the "WAVE" chunk. No more than one cue chunk is allowed in a "WAVE" chunk.

Figure 4-8 illustrates the layout of the cue points chunk and Figure 4-9 depicts an example of a cue point's fields for PCM-formatted data in a single standalone **data** chunk. The cue points chunk is composed of the following fields:

- ***ChunkID*****:** The chunk identifier value must be **cue**_.
- ***size*****:** The chunk size is a 32-bit positive integer containing the number of bytes in the cue chunk not counting the 8 bytes for the ID and size fields.
- ***dwCuePoints*****:** A 32-bit field containing the number of cue points in the chunk
- ***List-of-Cue-Points*****:** A list of one or more cue points. The number of cue points in the list is ***dwCuePoints***. Each cue point in the list contains the following fields:
  - ***dwName*****:** A 32-bit field containing the unique name or identifier for the cue point. May consist of four characters (i.e., FOURCC format) or an integer value.
  - ***dwPosition*****:** Specifies the sequential sample number within the play order in the playlist chunk[13]. (Recall that the **cue**_ chunk requires the presence of a playlist chunk.)

[13] There is ambiguity regarding the ***dwPosition*** field. Reference [2] states that ***dwPosition*** is the sample position within the play order. However, both Sonic Spot and a University of Trieste author (http://enteos2.area.trieste.it/russo/LabInfoMM2005-2006/ProgrammaEMaterialeDidattico/consultazione/016-waveFilesFormat2 htm) imply that ***dwPosition*** specifies the segment number in the **plst** chunk and is used to determine the play order of the play segments in the **plst** chunk.

- ***fccChunk*:** Specifies the name of the chunk that the cue point references; it is a FOURCC value. This field typically equals **data**; however**,** if a wave-LIST chunk is present in the file, then the value could be either **data** or **slnt.**
- ***dwChunkStart*:** A 32-bit integer that specifies the beginning of the **data** or **slnt** chunk that the cue point references. If there is only one **data** chunk present in the WAVE file, then this value must equal zero. However, if a WAVE-List chunk is present that contains multiple **data** or **slnt** chunks, then the ***dwChunkStart*** value should be the byte offset to the beginning of the **data** or **slnt** chunk relative to the ***Data*** section of the wave-LIST chunk. For example, if the cue point references the first chunk in the ***Data*** section of the wave-LIST chunk, then the ***dwChunkStart*** value will be zero because the offset from the beginning of the wave-LIST chunk's ***Data*** section is zero. Similarly, the maximum possible ***dwChunkStart*** value should be the offset in bytes from the beginning of the ***Data*** section of the wave-LIST chunk to the start of the **last data** or **slnt** chunk in the **wave-LIST** chunk.
- ***dwBlockStart*:** A 32-bit integer that specifies the beginning of the block of compressed data within a **data** or **slnt** chunk that the cue point references. The value depends on whether the **fmt**_ chunk's ***CompressionCode*** specifies that the audio data is to be compressed[14] and whether only a single **data** chunk is present versus a **wave-LIST** chunk containing one or more pairs of **slnt** and **data** chunks. (**Note:** Blocks are only used for compressed data.)
- ***dwSampleOffset*:** A 32-bit integer that specifies the sample offset of the cue point relative to the start of the block[15]. (**Note:** This is measured as the number of SAMPLES to the beginning of the cue point, NOT the number of bytes.) The value depends on whether the **fmt**_ chunk's ***CompressionCode*** specifies that the audio data is to be compressed and whether only a single **data** chunk is present versus a wave-LIST chunk containing one or more pairs of **slnt** and **data** chunks. (**Note:** Blocks are only used for compressed data.)

[14] See Section 4.1 for the discussion on blocks and the format chunk.

[15] There are ambiguities and possible errors in the specifications regarding cue points in **slnt** chunks. Examples given in [1] state that for a cue point within a **slnt** chunk, the ***dwBlockStart*** is the position of the ***Data*** section of the **slnt** chunk relative to the start of the ***Data*** section of the **wave-LIST** chunk and the ***dwSampleOffset*** is the sample position of the cue point relative to the start of the **slnt** chunk. The authors think the latter is an error and the ***dwSampleOffset*** is the sample position of the cue point relative to the start of the ***Data*** section of the **slnt** chunk.

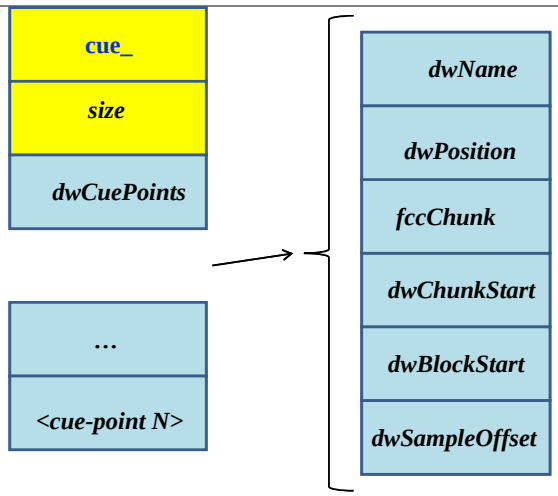

Figure 4-7. Cue Points Chunk Layout

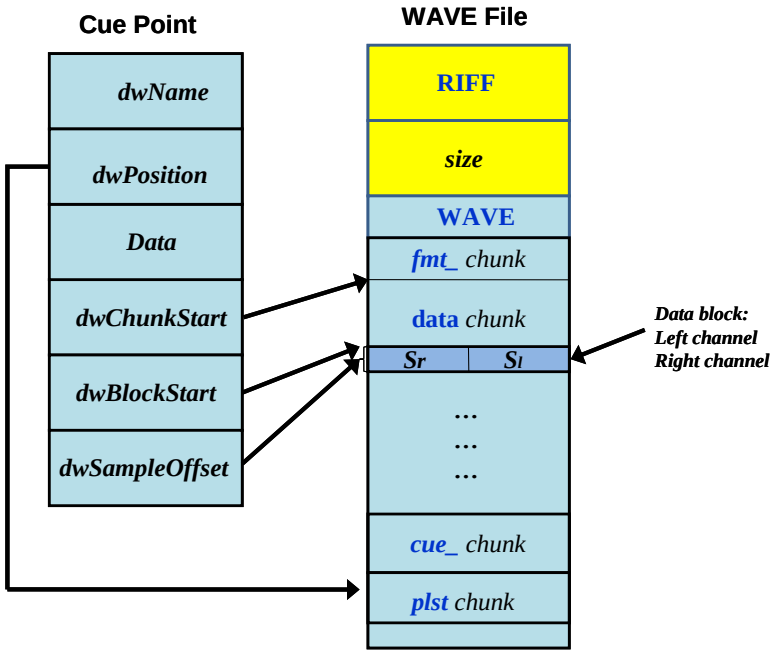

Figure 4-9. Cue Point in WAVE File Having PCM Data in a Single Data Chunk

CONCERNS:

Data Hiding - Use of the cue chunk could result in the audio data not being played back. Alternatively, sensitive information and/or malicious code could be embedded in the **data** chunk and the cue points could be used to skip over this data.

PRODUCT: WAVE

LOCATION:
Cue chunk

RECOMMENDATIONS:

1 **Validate**: Verify that the Chunk ID is **cue**_.

2 **Validate**: Verify that there are ***dwCuePoints*** items in the cue point list.

3 **Validate**: For each item in the cue points list, verify that ***dwName*** is distinct from all other entries of ***dwName*** in the cue points list.

4 **Validate**: For each item in the cue points list, verify that ***dwPosition*** points to a valid segment number in the Playlist chunk.

5 **Validate**: For each item in the cue points list, verify that ***fccChunk*** is either or **data** or **slnt**.

6 **Validate**: For each item in the cue points list, verify that the cue point refers to a valid section within the **data** or **slnt** chunk specified in the ***fccChunk*** field.

7 **Validate**: Check that *dwChunkStart* is the beginning of the **data** or **slnt** chunk that the cue point references. If there is only one data chunk, then ensure this value is zero.

8 **Validate**: If a WAVE-List chunk is present that contains multiple **data** or **slnt** chunks, then check that the ***dwChunkStart*** value is the byte offset to the beginning of the **data** or **slnt** chunk relative to the ***Data*** section of the wave-LIST chunk.

9 **Validate**: If compressed, check that ***dwBlockStart*** points to the beginning of the block of compressed data within a **data** or **slnt** chunk that the cue point references.

10 **Validate:** Check that ***dwSampleOffset*** is pointing to the correct location.

11 **Remove**: Remove the cue points chunk and associated playlist chunk to avoid skipping around the audio data during playback.

12 **Replace:** N/A

13 **External Filtering Required**: N/A

14 **Review**: N/A

15 **Reject**: N/A

WAVE 4.11: END

WAVE 4.12: PLAYLIST CHUNK

OVERVIEW:

The playlist chunk specifies the play order of a series of cue points. The cue points are defined in the cue chunk, elsewhere in the file. A playlist consists of an array of segments, each containing information about what sample the segment should start playing from, how long the segment is (in samples) and how many times to repeat the segment before moving on to the next segment in the play order. Figure 4-9 illustrates the playist chunk layout. The playlist chunk contains the following fields:

- ***ChunkID*:** The chunk ID value must be **plst**.
- ***size*:** The chunk size is a 32-bit positive integer containing the number of bytes in the playlist chunk not counting the 8 bytes for the ID and size fields.
- ***dwSegments*:** Number of play segments
- ***<play-segments>*:** Table of play segments. Each segment contains the following fields:
  - ***dwName*:** Contains the cue point name. The value must match one of the names listed in the cue-point table.
  - ***dwLength*:** Specifies the length of the section in samples.
  - ***dwLoops*:** Specifies the number of times to play the section during playback.

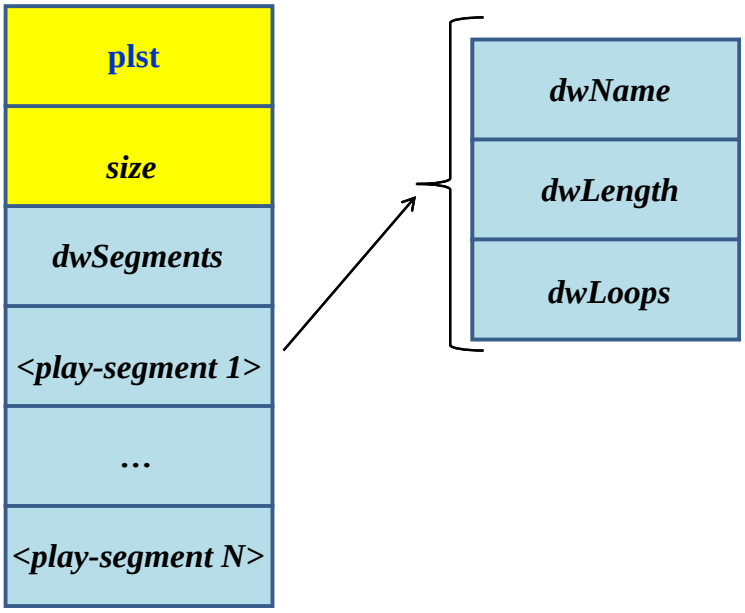

Figure 4-10. Playlist Chunk Layout

CONCERNS:

Data Attack - The ***dwLength*** and ***dwLoops*** values could be set to specially crafted values, resulting in play back of infinite loops of audio data.

Data Hiding - A playlist chunk can be crafted using specific cue points to skip over significant audio data.

PRODUCT: WAVE

LOCATION:
Playlist chunk

RECOMMENDATIONS:

1 **Validate**: Verify that the Chunk ID is **plst**

2 **Validate**: Check the cue length and loop time values to ensure that they are reasonable values (i.e., cue length should be no greater than the length of the **data** or **slnt** chunk data portion; the loop time should be no greater than a predetermined time value).

3 **Validate**: Ensure that every cue point name in the playlist chunk is contained in the cue chunk.

4 **Remove**: Remove the playlist chunk and associated cue points chunk to avoid skipping around the audio data during playback.

5 **Replace**: N/A

6 **External Filtering Required**: N/A

7 **Review**: N/A

8 **Reject**: N/A

WAVE 4.12: END

WAVE 4.13: ASSOCIATED DATA LIST CHUNK

OVERVIEW:
An associated-data-list chunk is a LIST chunk that has a form type of **adtl**. An associated-data-list chunk provides a method of connecting tags to sections of waveform data. Example tags include labels and descriptive text. This chunk associates a text label, title, and comment text with

a cue point. The associated-data-list chunk may have one or more of the following types of subchunks: a *label* chunk, *note* chunk, and/or *labeled text* chunk. Figure 4-11 illustrates the layout of the associated data list chunk.

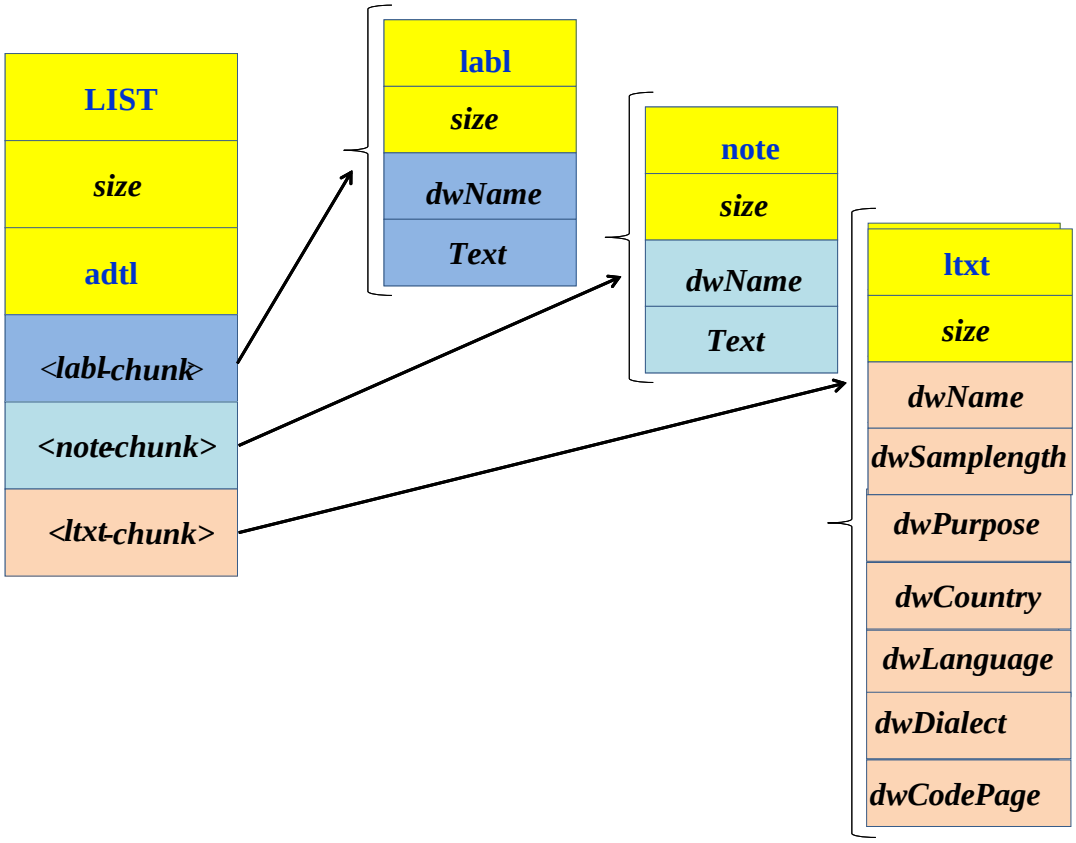

Figure 4-11. Associated Data List Chunk Layout

CONCERNS:
Data Hiding - If no **cue**_ chunk exists, then the text fields are not meaningful. When an application processes the file, it would likely skip over these chunks because there are no cue points to link with the associated text. This could be used for data-hiding purposes, with sensitive data contained in the text fields.

PRODUCT: WAVE

LOCATION:
Associated-Data-List Chunk

RECOMMENDATIONS:

1 **Validate**: Verify that the Chunk ID is **adtl**.

2 **Validate**: Verify that the file contains a **cue_** chunk.

3 **Validate**: Verify that *dwCountry* is a valid entry.

4 **Validate**: Verify that *dwLanguage* is a valid entry.

5 **Validate**: Verify that *dwDialect* is a valid entry.

6 **Remove**: Remove the associated data list chunk and associated label, note, and labeled text subchunks to avoid skipping around the audio data during playback.

7 **Replace:** N/A

8 **External Filtering Required**: N/A

9 **Review**: N/A

10 **Reject**: N/A

WAVE 4.13: END

WAVE 4.14: LABEL SUBCHUNK

OVERVIEW:

The label subchunk associates a text label (e.g., title) with a cue point. This information is often displayed next to markers or flags in digital audio editors. It must be a subchunk of the associated-data-list chunk. Figure 4-12 illustrates the layout of the label subchunk. The label subchunk consists of the following fields:

- ***ChunkID*:** The chunk identifier value must be **labl**.
- ***size*:** The chunk size is a 32-bit positive integer containing the number of bytes in the label subchunk not counting the 8 bytes for the ID and size fields.
- ***dwName*:** Specifies the cue point name
- ***Text*:** Contains a variable-length, null-terminated string which is the label associated with the cue point.

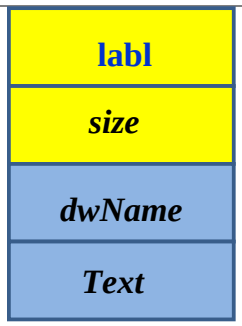

Figure 4-12. Label Subchunk Layout

CONCERNS:
Data Hiding, Data Disclosure – the text included in the ***Text*** field could contain sensitive information or hidden data.

PRODUCT: WAVE

LOCATION:
Label Subchunk

RECOMMENDATIONS:

1 **Validate**: Verify that the Chunk ID is **labl**.

2 **Validate**: Verify that the cue point ID name ***dwName*** corresponds to a valid cue point.

3 **Remove**: Remove the associated data list chunk and associated label, note, and labeled text subchunks to avoid skipping around the audio data during playback.

4 **Replace**: Replace the ***Text*** field with a standard template or null characters.

5 **External Filtering Required**: Pass the contents of the Text fields to an external filter.

6 **Review**: N/A

7 **Reject**: N/A

WAVE 4.14: END

WAVE 4.15: NOTE SUBCHUNK

OVERVIEW:

The note subchunk associates text (e.g., a comment) with a cue point. This information is stored in an identical fashion to the labels in the label chunk. It must be a subchunk of the associated-data-list chunk. Figure 4-13 illustrates the layout of the note subchunk. The note subchunk consists of the following fields:

- ***ChunkID*:** The chunk identifier value must be **note**.
- ***size*:** The chunk size is a 32-bit positive integer containing the number of bytes in the note subchunk not counting the 8 bytes for the ID and size fields.
- ***dwName*:** Specifies the cue point name. It must match one of the cue point names in the **cue_** chunk.
- ***Text*:** A variable-length, null-terminated string consisting of the textual comment associated with the cue point.

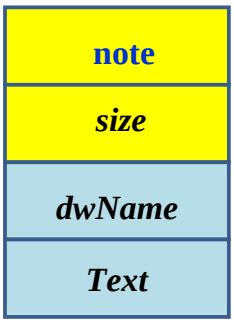

Figure 4-13. Note Subchunk Layout

CONCERNS:

Data Hiding, Data Disclosure – the text included in the ***Text*** field could contain sensitive information or hidden data.

PRODUCT: WAVE

LOCATION:

Note Subchunk

RECOMMENDATIONS:

1 **Validate**: Verify that the Chunk ID is **note**.

2 **Validate**: Verify that the cue point ID name ***dwName*** corresponds to a valid cue point.

3 **Remove**: Remove the associated data list chunk and associated label, note, and labeled text subchunks to avoid skipping around the audio data during playback.

4 **Replace**: Replace the ***Text*** field with a standard template or null characters.

5 **External Filtering Required**: Pass the Text field to an external filter.

6 **Review**: N/A

7 **Reject**: N/A

WAVE 4.15: END

WAVE 4.16: LABELED TEXT SUBCHUNK

OVERVIEW:

The labeled text subchunk associates some descriptive text with a data segment of a specified length (e.g., number of audio samples) that is associated with a cue point. This information is often displayed in marked regions of a waveform in digital audio editors. It must be a subchunk of the associated-data-list chunk. Figure 4-14 illustrates the layout of the labeled text subchunk. The labeled text subchunk consists of the following fields:

- ***ChunkID*:** The chunk ID value must be **ltxt**.
- ***size*:** The chunk size is a 32-bit positive integer containing the number of bytes in the labeled text subchunk not counting the 8 bytes for the ID and size fields.
- ***dwName*:** Specifies a cue point ID; must be one of the cue point names in the **cue**_ chunk.
- ***dwSampleLength*:** Specifies the number of samples in the segment of waveform data to be associated with the descriptive text. (**Note:** This is the "length" of data associated with the text.)
- ***dwPurpose*:** Specifies the purpose of the text. Any FOURCC code is usable. Common codes include **scrp** for scripts and **capt** for close captions.
- ***dwCountry*:** A 16-bit integer that specifies the country code for the text. The country codes are defined in reference [2].
- ***dwLanguage*:** A 16-bit integer that specifies the language code of the text. The language and dialect codes are defined in reference [2].
- ***dwDialect*:** A 16-bit integer that specifies the dialect code of the text. The language and dialect codes are defined in reference [2].
- ***dwCodepage*:** Specifies the code page for the text. The *code page* is the character-encoding table used in mapping a sequence of bits (usually 8) to a character in a language. Each language generally has its own mapping. For example, English, Greek, Russian, and Arabic each have their own code pages. The code pages are defined in reference [8].
- ***Comment*:** A sequence of bytes consisting of the textual comment to be associated with the data segment. (**Note:** Reference [1] defines a byte as an unsigned character.)

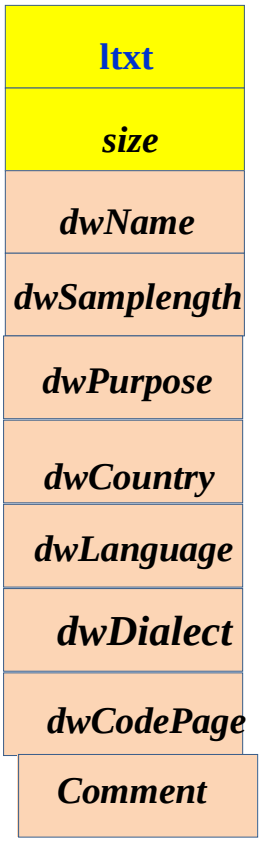

Figure 4-14. Labeled Text Subchunk Layout

CONCERNS:
Data Hiding, Data Disclosure – the text included in the ***Text*** field could contain sensitive information or hidden data.

PRODUCT: WAVE

LOCATION:
Labeled Text Subchunk

RECOMMENDATIONS:

1 **Validate**: Verify that the ChunkID is **ltxt**.

2 **Validate**: Verify that the cue point ID name ***dwName*** corresponds to a valid cue point.

3 **Validate**: Verify that *dwCountry* is a valid entry.

4 **Validate**: Verify that *dwLanguage* is a valid entry.

5 **Validate**: Verify that *dwDialect* is a valid entry.

6 **Remove**: Remove the associated data list chunk and associated label, note, and labeled text subchunks to avoid skipping around the audio data during playback.

7 **Replace**: Replace the ***Comment*** field with a standard template or null characters.

8 **External Filtering Required**: Pass the ***Comment*** field to an external filter.

9 **Review**: N/A

10 **Reject**: N/A

WAVE 4.16: END

WAVE 4.17: JUNK CHUNK

OVERVIEW:

In addition to WAVE chunks, some types of RIFF chunks often exist within a WAVE file as they serve a useful purpose supporting the WAVE format. These RIFF format chunks appear as nested chunks within the outer parent **RIFF** chunk, similar to WAVE chunks. The Junk chunk contains a variable number of filler bytes, typically to align RIFF chunks to certain boundaries (e.g., 2048-byte boundaries for CD-ROMs). WAVE applications typically ignore (skip over) the filler. Figure 4-15 illustrates the layout of the Junk chunk. The junk chunk consists of the following fields:

- ***ChunkID***: The ChunkID value must be **JUNK**.
- ***Size***: The chunk size is a 32-bit positive integer containing the number of bytes in the junk chunk not counting the 8 bytes for the ID and size fields.
- ***Filler***: The filler consists of bytes of unknown data. The length should be consistent with the Junk chunk's size.

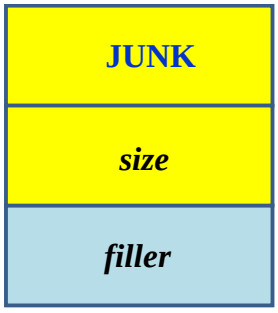

Figure 4-15. Junk Chunk Layout

CONCERNS:
Data Hiding – sensitive data can be included in the Junk chunk and ignored by most WAVE applications.

PRODUCT: WAVE

LOCATION:
Junk chunk

RECOMMENDATIONS:

1 **Validate**: Verify that that the ChunkID is **JUNK**.

2 **Validate:** Verify that the byte length of ***Filler*** is ***Size*** – 8.

3 **Remove**: N/A

4 **Replace**: Overwrite Filler data with zeros, keeping boundaries intact.

5 **External Filtering Required**: N/A

6 **Review**: N/A

WAVE 4.17: END

WAVE 4.18: PAD CHUNK

OVERVIEW:
The Pad chunk is similar to the Junk chunk and is used to pad data out to specific boundaries. It is a space filler of arbitrary size. Figure 4-16 illustrates the layout of the Pad chunk. The Pad chunk consists of the following fields:

- ***ChunkID***: The ChunkID value must be **PAD**_.
- ***Size***: The chunk size is a 32-bit positive integer.
- ***Filler***: The filler consists of bytes of unknown data. The length should be consistent with the Pad chunk's size.

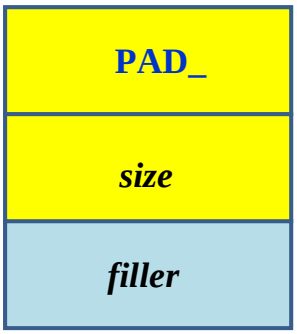

Figure 4-16. Pad Chunk Layout

CONCERNS:
Data Hiding – sensitive data can be included in the Pad chunk and ignored by most WAVE applications.

PRODUCT: WAVE

LOCATION:
Pad chunk

RECOMMENDATIONS:

1 **Validate**: Verify that that the ChunkID is **PAD_**.

2 **Validate:** Verify that the byte length of ***Filler*** is ***Size*** – 8.

3 **Remove**: N/A

4 **Replace**: Overwrite Filler data with zeros, keeping boundaries intact.

5 **External Filtering Required**: N/A

6 **Review**: N/A

WAVE 4.18: END

5. ACRONYMS

Table 5-1. Acronyms

Acronym	Denotation
ADPCM	Adaptive Differential Pulse Code Modulation
ASCII	American Standard Code for Information Interchange
AVI	Audio Visual Interleaved
BOF	Beginning of File
CD	compact disk
codec	compression/decompression
COTS	commercial off-the-shelf
CVE	Common Vulnerabilities and Exposures
D/A	digital to analog
DEFCON	Defense Condition
DoS	Denial of Service
DRM	Digital Rights Management
EOC	End of Cluster
EOF	End of File
FOURCC	Four Character Code data format
FTP	File Transfer Protocol
GB	gigabyte
ID	identifier
ISO®[16]	International Organization for Standardization
MIDI™[17]	Musical Instrument Digital Interface
MMA™	MIDI Manufacturer's Association
ms	millisecond
NULL	Containing all zeroes
OLE	Object Linking and Embedding
PC	personal computer
PCM	Pulse Code Modulation
REL-NATO	Releasable to North Atlantic Treaty Organizations
RIFF	Resource Interchange File Format
SMPTE	Society of Motion Pictures and Television E
SoX	Sound eXchange
WAV	Waveform Audio File Format

[16] ISO is a registered trademark of the International Organization for Standardization

[17] MIDI and MMA is a trademark of MIDI Manufacturers Association

WAVE	Waveform Audio File Format

6. REFERENCED DOCUMENTS

1. IBM Corporation and Microsoft Corporation, *Multimedia Programming Interface and Data Specifications, Version 1.0*, August 1991.
2. Microsoft Corporation, *Microsoft Multimedia Standards Update – New Multimedia Data Types and Data Techniques, Revision 3*, April 15, 1994.
3. Microsoft Corporation, *Microsoft Multimedia Standards Update, Revision 1.0.97*, November 28, 2006.
4. Sonic Spot, *Wave File Format*. Available at http://www.sonicspot.com/guide.
5. J. Pinquier, J. Rouas, and R. André-Obrecht, *A Fusion Study in Speech/Music Classification*, International Conference on Audio, Speech and Signal Processing, Hong Kong, April 6-10, 2003. (**Note:** Conference cancelled due to SARS outbreak; however, submitted papers were published.)
6. Peng, Yuh-Fen, *WAVE File Inspection and Sanitization Overview*, PowerPoint Presentation, The MITRE Corporation, 22 May 2007, Unclassified//For Official Use Only.
7. A.K. Jain, R. Duin, and J. Mao, "Statistical Pattern Recognition: A Review," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 22, No. 1, January 2000.
8. Microsoft Corporation, *Microsoft Global Development and Computing Portal – Code Pages*. Available at http://www.microsoft.com/globaldev/reference/cphome.mspx.
9. International Standards Organization (ISO), *8859-1 Standard Character Coding for the Latin Alphabet -1*, December 1998. Available at http://www.microsoft.com/globaldev/reference/iso.mspx.
10. L. Ericsson, *Automatic Speech/Music Discrimination in Audio Files*, Royal Institute of Technology, Sweden, 2009.
11. R. Huang and J. Hansen, *Advances in Unsupervised Audio Classification and Segmentation for the Broadcast News and NGSW Corpora*, IEEE Transactions on Audio, Speech, and Language Processing, Vol. 14, No. 3, May 2006.
12. J. Saunders, "Real-Time Discrimination of Broadcast Speech/Music," Proceedings of ICASSP96, vol. II, pp. 993-996, Atlanta, May 1996.
13. E. Scheirer and M. Slaney, "Construction and Evaluation of a Robust Multifeature Speech/Music Discriminator," Proceedings of ICASSP97, Munich, Germany, 1997.

14. L. Wyse and S. Smoliar, "Toward Content-Based Audio Indexing and Retrieval and a New Speaker Discriminator Technique," Institute of Systems Science, National Univ. of Singapore, Dec. 1995.

15. D. Kimber and L. Wilcox, "Acoustic Segmentation for Audio Browsers," Proceedings of Interface Conference, Sydney, Australia, July, 1996.

16. J. Benesty, M. Sondi, and Y. Huang, *Springer Handbook of Speech Processing*, Springer-Verlag, 2008.

Appendix A

WAVE File Audio Classification

7. APPENDIX A: WAVE FILE AUDIO CLASSIFICATION

A.1 Introduction

The WAVE file format is Microsoft Windows' native file format for the storage of digitized sound. The format was developed in the early 1990's and its use has been growing ever since. Given the increasing use of digital audio, the WAVE file format is likely to experience even wider usage.

One of the goals of Inspection and Sanitization Guidance (ISG) for a variety of file formats is to identify locations for data hiding and data attack. For the WAVE file format, the portion of the file containing audio data is the primary threat for data hiding and data attack.

As an application of ISG, the ability to automatically distinguish between conventional audio and manipulated audio in WAVE files is a desirable goal. The objective is to circumvent the necessity of listening to hours of audio while approaching the performance of human sound source recognition.

For this paper, conventional audio is defined as speech or music which has not been manipulated for hidden data or data attack. Human sound source recognition of conventional audio would readily classify it as speech or music.

On the other hand, manipulated audio may or may not be recognized as such; moreover, sophisticated data hiding using steganography such as least-significant bit (LSB) embedding requires special processing to detect the hidden data and is not addressed in this paper. Rather, the intent is to design a software filter which can flag WAVE file content such as text, images, executable code or other non-audio content that human sound source recognition would readily identify.

A.2 Current Technology

With the development of multimedia and web technology, the rapid increase in the sheer volume of multimedia data has resulted in demands for automatic management and retrieval. Content-based audio data management application areas include:

- Audio Segmentation and Classification
- Content-Based Audio Retrieval
- Audio Analysis for Video Indexing

A specific application example of audio segmentation and classification is the discrimination between speech and music since they are the most important and common types of digital audio. Several algorithms for accomplishing this have been published over the past several years ([12], [13]).

All of these approaches use a combination of waveform temporal features such as zero-crossing rate and energy profile, as well as spectral features such as bandwidth, pitch, and harmonicity to achieve classification accuracies over 90%. Since speech and music are quite different in spectral distribution and temporal change pattern, it is generally not very difficult to achieve relatively high discrimination accuracy.

A further classification of audio data may include sounds other than speech and music. In [14], audio signals were classified into "speech", "music", or "others" for the purpose of parsing news stories. An acoustic segmentation approach was used in [15], where audio recordings were segmented into speech, silence, laughter, and non-speech sounds. Cepstral coefficients were used as features and a hidden Markov Model (HMM) was used as the classifier. The method was primarily applied to the segmentation of meeting discussion recordings.

A.3 ISG Requirements

In order to design a practical WAVE filter, several requirements must be met. First, a definition of manipulated audio must be made precise to contrast with conventional audio. Depending on the application, this can include:

- Conventional audio with hidden content
- Audio other than speech or music (e.g., laughter, applause)
- Non-audio such as text, images, executable code, or sensor data

Second, the classifier must be designed with enough training data to discriminate effectively between conventional audio and manipulated audio. Samples of conventional audio can be easily obtained for training. In contrast, development of training data for manipulated audio can be challenging given the application.
If development of non-audio training data is impractical, a WAVE filter can be designed based on comparison of extracted audio stream parameters with those parameter values typical for speech and music. Although the filter would be less robust than a trained classifier, it would still provide the capability to identify non-audio for many applications.

Third, given that the data hiding or data attack duration can be small compared with the WAVE file duration, the filter must operate over short epochs (e.g., 1 sec) and

output a decision (conventional audio / manipulated audio) every epoch. This allows the filter the flexibility to retain the epochs of conventional audio at output, while removing identified non-audio content.

A.4 Non-Audio Content Flags

Initial processing of the WAVE file should begin with extraction of audio parameters. These include WAVE header parameters applicable to the entire audio stream as well as audio waveform parameters which vary over short segments (e.g., 25 milliseconds). The filter should identify parameters which are atypical for speech and music and report these to the user. Parameters in the WAVE header applicable to the entire file include:

- *Sampling rate*: typical values (samples per second) are 8000, 8192, and 16000 for speech, and 48000, 44100, and 22050 for music. Since the allowable sampling rate for WAVE files is between 1 Hz and 4.3 GHz, values other than typical rates for speech and music should be reported to the user.
- *Number of channels*: the typical value is two channels both speech and music although surround sound (7.1) audio includes 8 channels. Since the allowable number of channels is between 1 and 65536, values larger than 8 should be reported to the user.
- *Number of bits per sample*: the usual value for speech and music is 16, although 8 and 24 are also used. Values other than these are rounded up to the nearest multiple of 8, and should be reported to the user.

In addition to alerting the user to WAVE file header entries, parameters associated with the audio stream which vary over time include the following:

- *Bi-polar indicator*: 16-bit signed integers are typical for speech and music which results in waveform values between -32768 and 32767. The filter should report epochs during which the waveform values are all positive or all negative as can occur for text and images embedded in the audio portion of a WAVE file.
- *Dynamic range indicator*: epochs during which the waveform is confined to a small percentage of the dynamic range associated with the number of bits should be reported. For example, waveform values between -1000 and 1000 for

16-bit signed integers (or approximately 3% of the dynamic range) should be reported.

- *Bandpass filter indicator*: since the human hearing range is nominally 20 Hz – 20 kHz, there is potential for data hiding and data attack outside this band. An acoustic energy detector in the bands [0 Hz, 20 Hz] and [20 kHz, fs/2] (for sampling rates fs > 40 kHz) can alert the user to this condition and implement a bandpass filter to attenuate these bands.

- *Speech/Music Detection*: Speech consists of both periodic (e.g., voiced speech) and a-periodic components (e.g., unvoiced speech). Similarly, music consists of both periodic and a-periodic components. Linear predictive coding (LPC) has been successfully used for decades as an all-pole model of speech and music [16]. A simple indicator of periodicity is the ratio of the energy of the LPC residual signal to the energy of the original waveform segment.

Figure A-1 illustrates linear predictive coding applied to speech. Figure A-1(a) shows 0.25 seconds of speech which includes both voiced and unvoiced segments. Figure A-1 (b) shows the predicted signal after applying 10^{th}-order LPC to ten consecutive 0.025 second epochs. The LPC residual signal (difference between original and predicted) is shown in Figure A-1 (c). Finally, the ratio of the energy of the residual signal and original speech is shown in Figure A-1 (d). Note how the energy ratio approaches zero over the voiced segment but is significantly higher over the unvoiced segments.

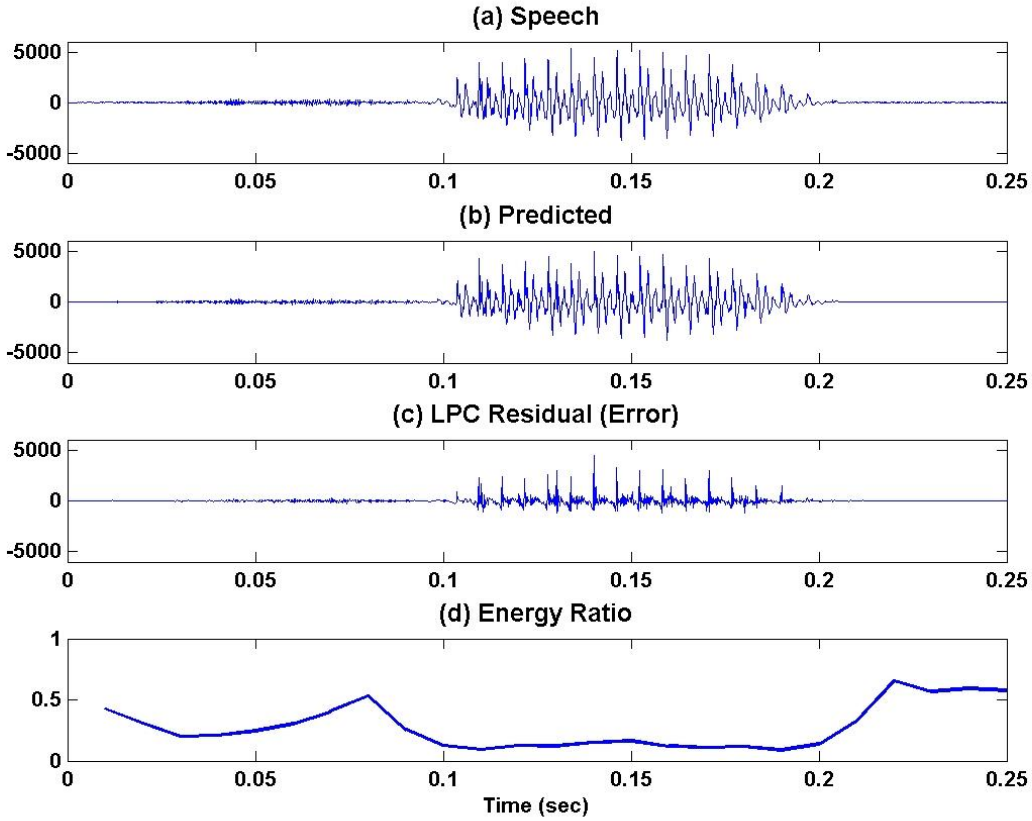

Figure A-1. Illustration of Linear Prediction Applied to Speech

A.5 Performance for Typical Audio Streams

The energy ratio described above approaches zero for periodic segments where linear prediction is effective and approaches unity for a-periodic segments where it is not effective. Data hiding or data attack segments in the WAV audio stream should typically be a-periodic and consequently produce energy ratio values near unity.

An example of the speech/music detection algorithm is shown in Figure A-2 which illustrates one-second epochs of speech, music, and a portion of a webpage embedded as unsigned 8-bit integers into the audio portion of a WAVE file. In all three cases, linear prediction was performed over consecutive 20-millisecond segments and the energy ratio was computed for each segment.

Note that the energy ratio of speech rises and falls between approximately 0.1 and 0.6 depending on whether the segment is voiced or unvoiced. For music, the energy ratio

is close to zeros for all segments indicating good linear prediction over all segments. On the other hand, the energy ratio for the embedded webpage is close to unity for all segments indicating the a-periodic nature of all segments.

During audio stream processing, all three of the waveform indicators as well as the speech/music detector described above are extracted over 1-second epochs and the results compiled for the user. A user-defined threshold (default = 0.5) for the speech/music detector allows for the identification of non-audio embedded content and its location in the WAVE file. At completion, the filter allows the user the option of removing non-audio content in the output WAVE file.

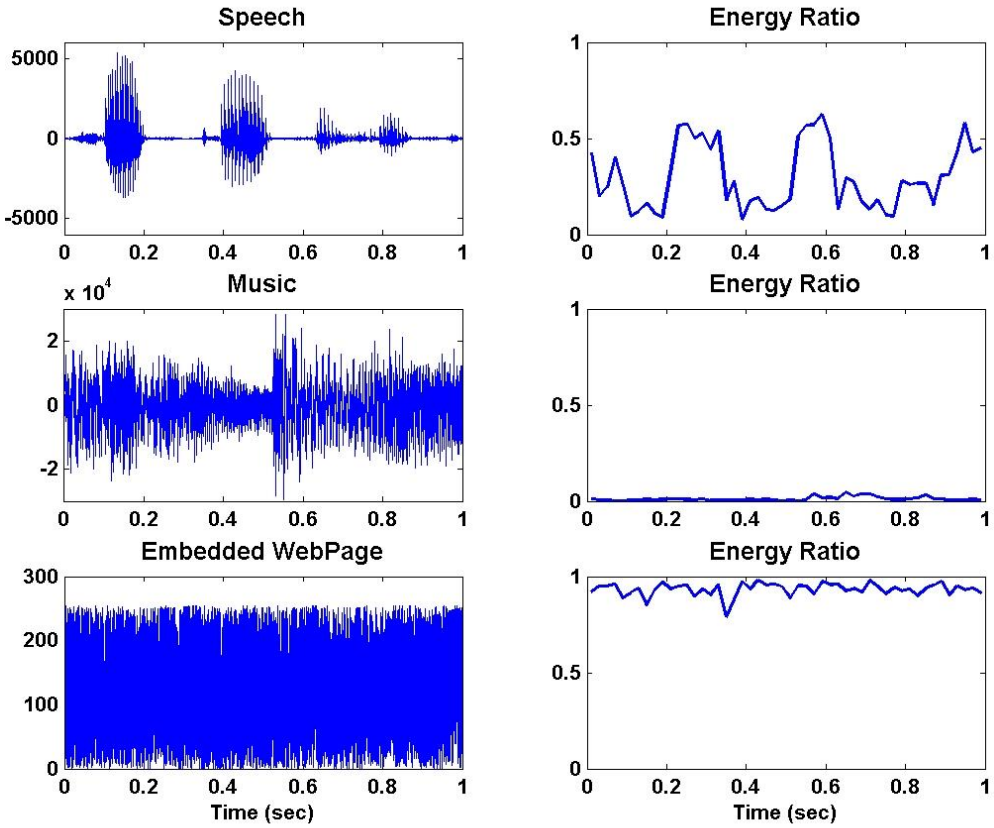

Figure A-2. Energy Ratio for Speech, Music, and Embedded Webpage

A.6 Disruption of WAVE File Embedded Content

The preceding sections present a methodology for *detection* of non-audio in WAVE files by inspecting header parameters applicable to the entire file as well as processing the

audio stream to identify a-periodic waveform components typical of non-audio embedded content. Another approach is *disruption* of non-audio content by altering the audio stream.

This can be accomplished by de-compression of the audio stream followed by re-compression using a different compression format. (If the audio stream is in uncompressed PCM format, the first step would be omitted.) The objective is to disrupt embedded content such as executable code. This requires resetting the header parameter describing the compression format in the output file.