Windows 7 Security Event Log Format

Todd Heberlein
23 Sep 2010

Windows security event log provides a rich source of information to detect and analyze a wide range of threats against computer systems. Unfortunately Windows tool to view these logs, Event Viewer, is extremely limited in its functionality. Furthermore, there are very few third-party analysis tools to fill the gap between what the Event Viewer provides and the potential information that can be leveraged from the security event logs. One potential reason for this gap is that the format of these event logs is poorly documented making it very difficult for third-party developers to write tools to analyze the data. This paper documents the event log format, thus providing a blueprint for developers to create native tools to analyze Windows 7 event logs. We begin by providing an overview of the format and key concepts that are needed to understand the details. Then we dive into a detailed description of the syntax of the event log format.

Table of Contents

List of Figures

1 Introduction

Microsoft's security event log is a rich source of information about the activity happening on a computer, including which programs were run, what files those programs interacted with, and what network connections they made. While Windows includes the Event Viewer program to view the contents of the security event log, this program is cumbersome to use, especially when analyzing large log files. So there exists a tremendous gap between the analysis possibilities provided by the rich event log content and the capabilities provided by built-in Event Viewer application, and surprisingly there are few third-party tools to fill that gap.

One potential reason for the gap is that Microsoft changed the event log format between Microsoft XP and Microsoft Vista (which is also used in Windows 7), and there is very little documentation available describing the new format. Tools and knowledge developed for analyzing Windows XP and earlier Windows operating systems do not transfer easily to the new log format.

This paper begins to address the analysis tool gap by providing a detailed description of the current Microsoft Windows event log format. Developers should be able to use the information in this paper to write their own event log parsers on top of which they can build useful types of analysis tools. We have used the information in this paper to build our own C++ parser, and we are currently building analysis tools that leverage techniques we originally developed for analyzing the BSM audit trails available for Mac OS X, Solaris, and FreeBSD.

There are a couple of caveats that we need to cover.

First, in developing this paper we did not examine any internal Microsoft documentation describing their event log format. We did not look at any Microsoft source code. And we did not talk to any Microsoft developers about their event log format. Part of our reason for taking this approach was to avoid any non-disclosure issues. But this means that there remain a few gaps in our knowledge of the event log format. In particular, there are several byte sequences for which we can find no purpose. They do not appear to effect the parsing of the data, so we simply flag them in the description.

Second, the completeness of our reverse engineering is limited by the test sets we generated. We do occasionally encounter new syntax as we parse new data files, so we cannot claim that the format presented in this paper is 100% accurate and complete.

Third, since Microsoft has not documented the event log format, this certainly gives them room to modify the format at any point with no notification. Just as using undocumented, private APIs in an operating system is risky and may result in your code breaking the next time the vendor updates the operating system, using an undocumented file format carries the risk that an update by the vendor may break your analysis code.

Fourth, throughout this paper we refer to the "security event log" as an "audit trail" because this is the name by which this data has been referred to throughout the rest of the computer industry for decades.

The rest of this paper is divided into the following sections:

Section 2 introduces a number of concepts necessary for understanding the rest of the audit trails. This includes a high-level view of the audit file format (Section 2.1), how Microsoft optimized space by use of XML templates and string reuse (Section 2.2), and a number of miscellaneous notes such as Microsoft's multiple encoding techniques for strings (Section 2.3).

Section 3 provides the detailed structure of the audit trail format and is the heart of the paper. We broke down the file format into a Backus Normal Form (BNF) like specification, but we took

some liberties since the format is clearly not context free. For this paper we used a syntax diagram (or railroad diagram) approach because we felt it was easier to understand than a pure textual representation.

Section 4 describes some of our testing we conducted to give us a measure of confidence that our analysis of the format is correct. Since our analysis did not rely on any specification or source code from Microsoft, our confidence must be tempered since it is limited by the amount of test we generated.

Section 5 provides our conclusions and some closing remarks.

Finally we would like to extend a special appreciation to Andreas Schuster for writing the paper “Introducing the Microsoft Vista event log file format” [Schu 07] without which we would not have even attempted to start this project. Anyone using this paper for the purpose of writing their own parsers should probably have Schuster’s paper next to them as well.

2 Broad Concepts

2.1 File architecture

Windows 7’s audit trail format can be visualized as two systems with a mapping between them (see Figure 1). At one level is a very limited in RAM memory model containing a file header and a single chunk. The chunk contains the actual audit records. At a second level is a potentially very large file memory model (the size can be configured manually) containing a single file header and a potentially huge number of chunks. The file header in memory always maps to the file header in the file. However, the chunk in memory maps to only one chunk in the file at a time. Once the chunk in memory fills up, it is cleared, and then it is mapped to the next chunk on the file. In other words, the chunk in memory maps to one chunk in the file, and then later it maps to a different chunk, and so on.

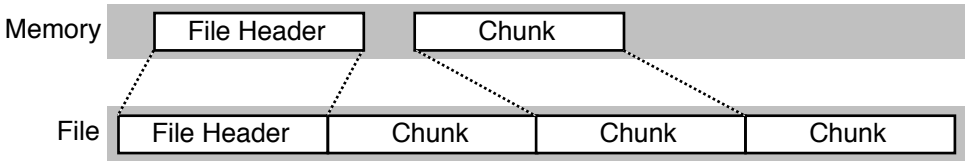

Figure 1: The Chunk Model

When the last chunk in the file is reached and fills up, depending on how the audit policy is set, the audit system can remap the in-memory chunk back to the first chunk and overwrite its contents. In other words, the chunks are treated as a circular buffer. There are two results from this policy. First, earlier audit records are permanently lost. And second, the earliest audit records available in the file may not be in first ones in the file. If you want to analyze audit records in the order that they were produced, you need to first determine which chunk in the file is the oldest.

Each chunk is completely self-contained. All information needed to reconstruct the audit records in that chunk (e.g., template definitions) is in the chunk. Each chunk is 64K in length and typically holds around 100 audit records.

2.2 Space Optimizations

Microsoft’s audit records are essentially XML objects. XML is a language that is somewhat easy for humans to read and somewhat easy for computers to parse, but the price to support both computer and human readability is space inefficiency. For small data sets this may not be an issues, but when thousands or millions of records need to be archived, this becomes a problem.

Microsoft addresses this inefficiency in two primary ways. First, Microsoft uses templates to reduce XML structural redundancy across records. Second, Microsoft uses string pointers to reduce redundancy within template definitions. Furthermore, to further reduce space, templates and strings that are never used in a chunk are not defined in a chunk. Since the audit system cannot tell ahead of time which templates and strings will be needed in the chunk, it uses a just-in-time strategy where the aren't defined until the very first time they are used. We cover these optimization techniques in the following sections.

2.2.1 XML Templates

Microsoft's audit system uses templates to remove redundancy across records. This is very important since the audit data consists of a large number of very similar small XML objects (i.e., each audit record). Any system that generates large numbers of repeating data chunks (e.g., a sensor generating regular data reports) can benefit from such an optimization.

Figure 2 illustrates the use of templates to reduce redundancy. On the left side of Figure 2 are two pseudo audit records in an XML format. Each record begins with the "EventData" subtree. In the first record user "Todd" is running the program "mspaint.exe", and in the second record "Todd" is running the program "hackme.exe". Most of the text of the two records is the same between both records, consisting of XML element and attribute names (e.g., "EventData", "Data", and "Name").

A template essentially extracts the fixed part of the XML subtree and inserts placeholders in the locations that actually change. On the right side of Figure 2 is a template representing the "EventData" subtree. There are three placeholders labeled (1), (2), and (3) in the template. Following the template definition are two substitution arrays containing the data that should be placed in the appropriate positions in the template. For example, a program that reads the audit trail recreates the first audit by taking the template and inserting "Todd" in placeholder (1), "mspaint.exe" in placeholder (2), and 321 in placeholder (3). Once the template is defined, each subsequent audit record of the same data simply inserts the substitution array data into the audit trail.

```
<EventData>
   <Data Name="user">Todd</Data>
   <Data Name="program">mspaint.exe</Data>
   <Data Name="processID">321</Data>
</EventData>
<EventData>
   <Data Name="user">Todd</Data>
   <Data Name="program">hackme.exe</Data>
   <Data Name="processID">322</Data>
</EventData>
```

Normal XML

```
<EventData>
   <Data Name="user">(1)</Data>
   <Data Name="program">(2)</Data>
   <Data Name="processID">(3)</Data>
</EventData>

"Todd", "mspaint.exe", 321
"Todd", "hackme.exe", 322
```

XML Template with Substitution Arrays

Figure 2: Template Definition

2.2.2 String Reuse

Microsoft's audit system also removes redundancy within template definitions by using string pointers. Figure 3 illustrates the problem the audit system addresses. On the left side in an example template developed in the previous section. As the grey highlights show, many of the strings are simply repeats of strings used earlier. The audit system removes this redundancy in element and attribute names by replacing subsequent use of a string by a pointer back to the original string. For closing element tags, the audit system removes the string altogether and simply uses a close element token (discussed later). The name in the closing element tag can be removed because in well-formed XML the name is not needed if you track the nesting of XML element names.

The right side of Figure 3 shows the template definition after the redundant strings are removes. A string pointer is visually represented by an asterisk, '*', and the close element token is visually represented by "</>".

```
<EventData>
   <Data Name="user">(1)</Data>
   <Data Name="program">(2)</Data>
   <Data Name="processID">(3)</Data>
</EventData>
```

Normal Template

```
<EventData>
   <Data Name="user">(1)</>
   <* *="program">(2)</>
   <* *="processID">(3)</>
</>
```

Template with String Pointers

Figure 3: Removing String Redundancy in Templates

2.2.3 Just-in-time Redundancy Removal

Microsoft's audit system could simply place all possible strings and template definitions at the front of a chunk, and then all audit records could simply be substitution arrays that begin with a pointer to its corresponding template. However this would probably be a waste since most templates won't be referenced in any given chunk. To avoid this waste, the audit system uses a just-in-time strategy for defining strings and templates where strings and templates are not defined until the first time they are needed.

A side effect of this just-in-time approach is that strings and template definitions are spread throughout the chunk. To find these strings and templates, the audit system includes a 64-bucket string table followed by a 32-bucket template table at the beginning of the chunk. Each bucket is a pointer to a string structure or template definition in the chunk. If more than one string structure or template definition hashes to the same bucket, the bucket points to the most recently added object (string structure or template) in the chunk, and then that object includes a pointer to the previously most recently added object that hashed to the same bucket. This is one of two most common ways to handle hash table collisions.

For example, Figure 4A shows a string table after two string structures have been inserted in the chunk data stream. When the third string structure needs to be inserted, it turns out that is hashes to the same string table bucket. To address this, the third string structure's "next pointer" points to str1, and then the hash table bucket is set to str3.

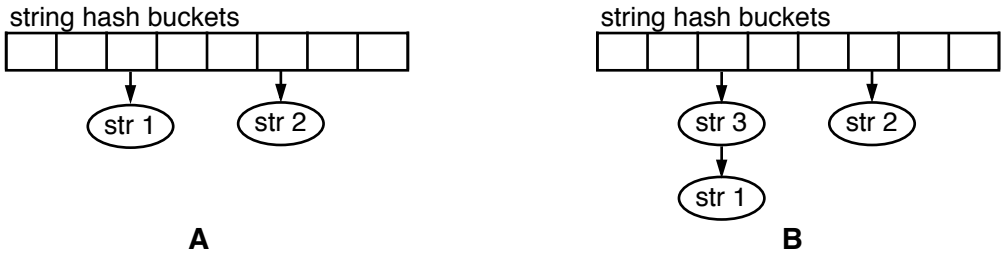

Figure 4: String Table and Linked List

2.3 Few More Notes

We have several more observations before diving into the details of the audit trail format.

2.3.1 XML Tokens

Certain XML elements are encoded as tokens and not their character values. For example, for the start of an XML element, the '<' character is encoded as the hexadecimal value 0x01. And as mentioned previously, all close elements (e.g., "</foo>", "</bar>", and "</snafu>") are all

represented by a single close token, the hexadecimal value 0x04. Figure 5 shows the token encoding IDs for these XML elements.

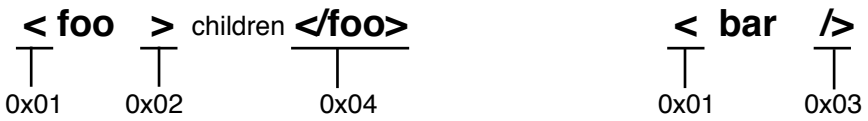

Figure 5: XML Token IDs

2.3.2 String Encodings

Character strings are encoded in three different ways in the audit trail. In general, a character is a two-byte value (UTF-16 little endian byte order), so the ASCII character 'Y', with the hexadecimal value of 59 is encoded as the following two bytes:

```
59 00
```

In our figures throughout the rest of this paper, a block labeled "char" represents a two-byte character. The "length" field for character strings is a two-byte number, but its interpretation depends on where the string is being used. Figure 6 shows the three types of string encodings. Each encoding is described below.

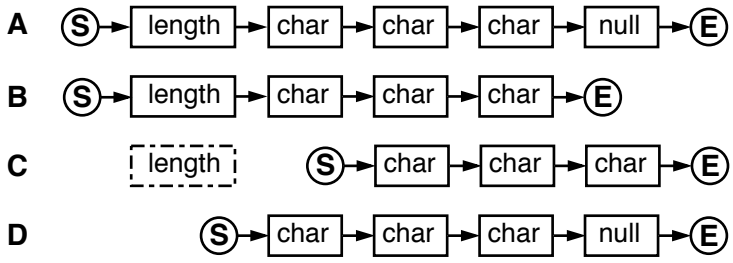

Figure 6: Strings

2.3.2.1 String Format A: Template Structural Definitions

Strings used in template structural definitions (i.e., XML element names and attribute names) are embedded in a string structures. If the same string is used more than once in a template structural definition within a single chunk, subsequent uses simply point to the original use, thus, in theory, saving space. Inside the string structures, the string is defined as shown in String A of Figure 6. Here, the "length" field is a two-byte number and only counts the primary 2-byte character and does not include the 2-byte null terminator character. For example, "YES" is encoded in 10 bytes: 2 for the length (in this case, the value is 3), 6 for the characters 'Y', 'E', and 'S', and 2 zero bytes for the NULL terminator:

```
03 00 59 00 45 00 53 00 00 00
```

2.3.2.2 String Format B: Template Value Field

Strings used in template definitions for attribute values (as opposed to attribute names) and XML child values (as opposed to XML sub-trees or substitution information) are defined as shown in String B of Figure 6. It is similar to String A except there is no NULL terminator. Here, the "length" field is a two-byte number and counts the number of characters. For example, "YES" is encoded in 8 bytes: 2 for the length (again, with a value of 3) and 6 bytes for the characters:

```
03 00 59 00 45 00 53 00
```

2.3.2.3 String Format C: Substitution Array Value

Strings embedded in substitution arrays are defined differently. First, as indicated in String B of Figure 6 the length field is not inline with the actual string. As shown in the Substitution Array Section, the length field is defined in the array index prior to the actual values. Second, the "length" field, while still a two-byte number, now represents the total number of bytes and not the number of 2-byte characters. For "YES" the length is 6 instead of 3.

```
06 00 … 59 00 45 00 53 00
```

2.3.2.4 String Format D: Array of Strings in Substitution Array Values

In addition to the audit system recording a single string in the value field of a substitution array, sometimes it records an array of strings. In this case the length field listed in the substitution array's index table has nothing to do with the length of any particular string in the array of strings but the combined length of all the strings. The individual strings are encoded as String D in Figure 6. There is no length field. The 2-byte NULL terminator identifies the end of the string. There is no indicator for the end of the array. For "YES" the encoding is:

```
59 00 45 00 53 00 00 00
```

2.3.3 Length Fields

Throughout the audit file there are many "length" fields; unfortunately the semantics of these fields vary depending on the context. As already mentioned, a "string length" may be the total number of bytes, the number of 2-byte characters, or the number of 2-byte characters but not including a 2-byte NULL terminator. In some structures, the length field in a structure may refer to only the remaining bytes of the structure, but in other structures, the length field may also refer to all the preceding bytes of the structure as well. We will explicitly state what the length field refers to in each appropriate section.

2.3.4 Pointers

All references to "pointers" in this paper refer to addresses relative to the beginning of a chunk. For example, if a pointer has the value of 1203, it is referring to the 1204th byte in the chunk. The first byte has an address of 0.

2.3.5 Data Types

Throughout the audit data Microsoft frequently references a "type" field. Locations include the value portion of an attribute name-value pair, substitution information in templates, substitution array index information, and an element's children values. The information about these types in the table below was taken from Microsoft's web site[1]. There are a few additional notes called out in the description field.

Name	Value	Description
EvtVarTypeNull	0x00	Null content that implies that the element that contains the content does not exist. (**Note: In substitution arrays most index information marked as Null have zero length, but that is not always the case.**)
EvtVarTypeString	0x01	A null-terminated Unicode string. (**Note: As discussed above, do not assume the string is null-terminated.**)
EvtVarTypeAnsiString	0x02	A null-terminated ANSI string.
EvtVarTypeSByte	0x03	A signed 8-bit integer value.

[1] http://msdn.microsoft.com/EN-US/library/aa385616.aspx

EvtVarTypeByte	0x04	An unsigned 8-bit integer value.
EvtVarTypeInt16	0x05	An signed 16-bit integer value.
EvtVarTypeUInt16	0x06	An unsigned 16-bit integer value.
EvtVarTypeInt32	0x07	A signed 32-bit integer value.
EvtVarTypeUInt32	0x08	An unsigned 32-bit integer value.
EvtVarTypeInt64	0x09	A signed 64-bit integer value.
EvtVarTypeUInt64	0x0A	An unsigned 64-bit integer value.
EvtVarTypeSingle	0x0B	A single-precision real value.
EvtVarTypeDouble	0x0C	A double-precision real value.
EvtVarTypeBoolean	0x0D	A Boolean value.
EvtVarTypeBinary	0x0E	A hexadecimal binary value.
EvtVarTypeGuid	0x0F	A GUID value.
EvtVarTypeSizeT	0x10	An unsigned 32-bit or 64-bit integer value that contains a pointer address.
EvtVarTypeFileTime	0x11	A FILETIME value.
EvtVarTypeSysTime	0x12	A SYSTEMTIME value.
EvtVarTypeSid	0x13	A security identifier (SID) structure
EvtVarTypeHexInt32	0x14	A 32-bit hexadecimal number.
EvtVarTypeHexInt64	0x15	A 64-bit hexadecimal number.
EvtVarTypeEvtHandle	0x20	An EVT_HANDLE value.
BinaryXmlStream	0x21	(**Note: Not documented, but is a nested Binary XML Stream.**)
EvtVarTypeEvtXml	0x23	A null-terminated Unicode string that contains XML.

2.3.6 Graph Key

Figure 7 is the key for graphs used through Section 3. The primary elements are:

- Start and end tags – Parsing for each block begins at the start tag and finishes at the end tag.
- Leaf – Rectangles represent a leaf in the graph tree; it is not expanded later on.
- Subtree – Capsules represent an intermediate node in the graph tree that is expanded in another section.
- Length – For leafs the length (i.e., number of bytes) should be known and are listed above the leaf. Sometimes the subtree's length is also known, so the length is shown above it as well.
- Value – For some leafs the value is predetermined. This is usually the case for flags indicating how to interpret the following bytes. In these cases, the hexadecimal value is shown below the leaf.
- Look ahead – Some pathways have a value written above them. If you look ahead at the next byte in the audit file, this value will often tell you which path to take when parsing the data file. These values that determine which path to take are written above the path. These look ahead values are not consumed when parsing here (hence the term "look ahead").
- Path joins and splits – graphs use small circles to simplify the graphs, usually by collapsing the number of arcs that would otherwise be needed.

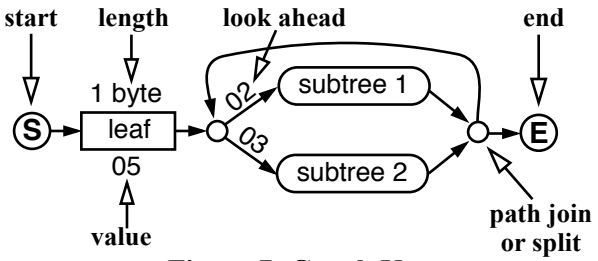

Figure 7: Graph Key

3 Audit Trail Format

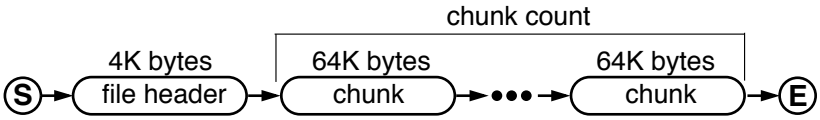

Figure 8: File Format

Figure 8 is the top level of the graph. A Windows audit trails consists of a fixed sized header followed by number of fixed sized chunks. All the information in a chunk is self-contained and does not require any information from any of the other chunks.

3.1 File Header

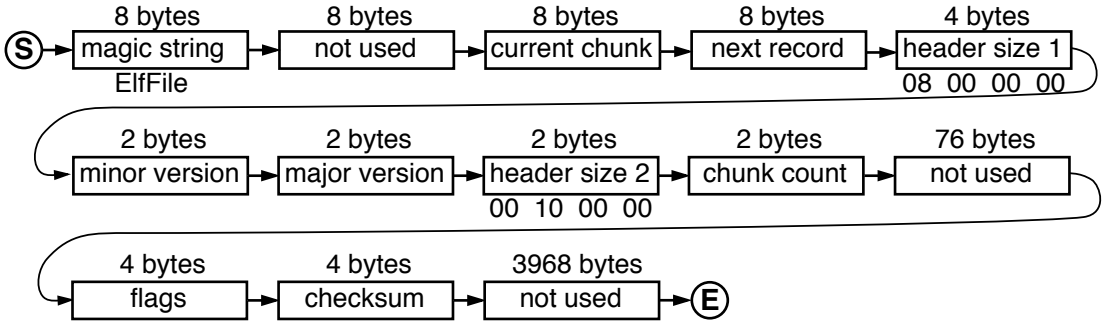

Figure 9: File Header

Figure 9 is the fixed sized file header at the beginning of the audit trail. Some of the information (e.g., current chunk and next record) is primarily useful when the audit file is still being actively used to record audit date.

- Magic string – The file begins with 8 bytes that spell out the ASCII string “ElfFile” followed by a null-terminator byte.
- Not used – The next 8 bytes, as far as we can determine, are not used for anything.
- Current chunk – This is the index to the current chunk in the file that the audit system is currently writing to. This seems primarily useful when watching life audit data.
- Next record – This is the record of the next audit record to be written.
- Header size 1 – The header includes two size fields. This first one represents the size of the header that is actively used.
- Minor version & major version – these values specify the audit file format version.
- Header size 2 – This size fields specifies the total size of the header, including the 3,968 bytes that are currently not used.
- Not used – A large block of bytes currently unused.
- Flags – This identifies information about the current state of the audit file.

- Checksum – Integrity check on the content of the data.

3.2 Chunk

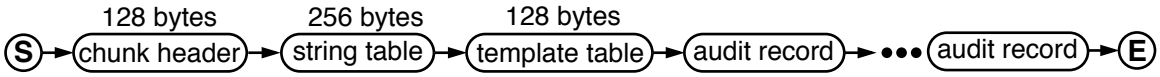

Figure 10: Chunk Structure

Figure 10 is the overall structure of a chunk. It begins with a fixed sized chunk header followed by fixed sized string and template hash tables. Following these structures are the actual audit records.

- Chunk header – This is a fixed size header for the chunk. It is useful for determining how many audit records are in the chunk, and if this is the active chunk for an active audit file, it points to where the next audit record will be written.
- String table – This is a hash table of 64 pointers to string structures distributed in the audit records.
- Template table – This is a hash table of 32 pointers to audit record templates distributed in the audit records.
- Audit record – An audit record records an event that occurred on the system. It will always include a substitution array, and it may include template definitions.

3.2.1 Chunk Header

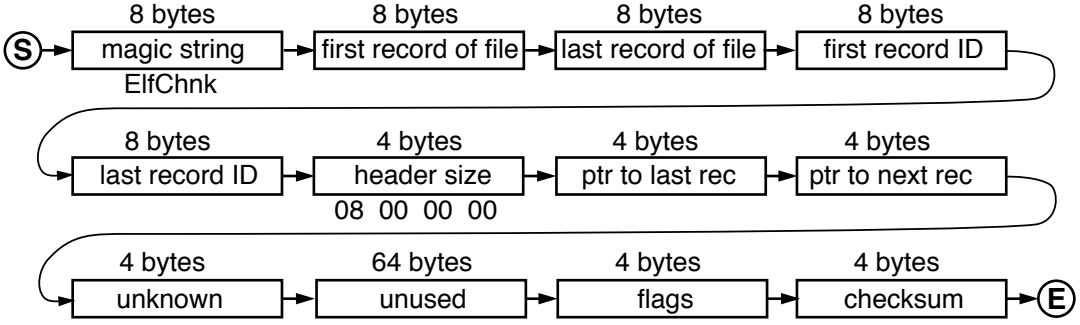

Figure 11: Chunk Header

Figure 11 is the fixed header at the beginning of a chunk. The audit system uses two different counters for audit records. The first is counter for records written into the file. For example, the first audit record would have the value 1. The second is a global counter independent of the audit file. For example, for this counter (described in the figure as a "record ID"), the first audit record written to the file may have the number 314,159. Because of these two different counters, the chunk header keeps tracks of four counters: the file counter and global counter for the first audit record in this chunk and the file and global counter for the last record in this chunk.

- Magic string – The file begins with 8 bytes that spell out the ASCII string "ElfChnk" followed by a null-terminator byte.
- First record of file – the file counter for the first audit record in this chunk.
- Last record of file – the file counter for the first audit record in this chunk.
- First record ID – the global counter for the first audit record in this chunk.
- Last record ID – the global counter for the last audit record in this chunk.
- Header size – size of the chunk's header.
- Ptr to last rec – offset in this chunk to the beginning of the most recent audit record written in this chunk.

- Ptr to next rec – offset in this chunk to the location where the next audit record will be written.
- Unknown – a sequence of (so far) non-zero bytes the purpose of yet we don't know.
- Unused – padding.
- Flags – additional details about the chunk's state.
- Checksum – Integrity check on the content of the data.

3.2.2 String and Template Tables

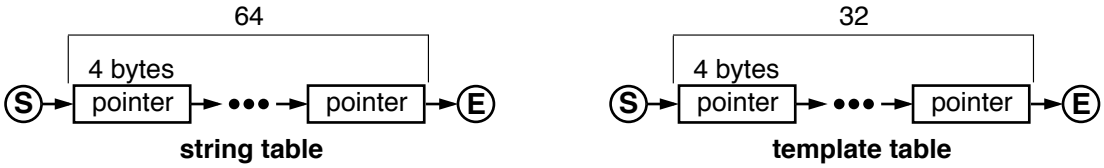

Figure 12: String and Template Tables

The string and template tables can be used to identify all the templates and strings used in those templates in this particular chunk. Each pointer is an offset from the beginning of the chunk. See Section 2.2.3 for more details.

3.2.3 Audit Record

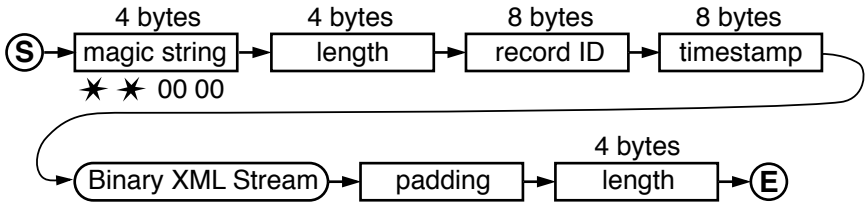

Figure 13: Audit Record

Figure 13 is the top-level description of an audit record. It consists of some administrative data and the beginning and ending with the meat of the audit record in the Binary XML Stream.

- Magic string – the record begins with two asterisks ('*', ASCII 0x2A) followed by two zero bytes.
- Length – the length of the entire audit record, including these four bytes and the four bytes of the magic string.
- Record ID – the audit system's global counter for this audit record.
- Timestamp – time this event took place.
- Binary XML Stream – the heart of the audit record.
- Padding – optional padding. Sometimes there is random data in this padding.
- Length – the length of the entire audit record. This is the same value previously listed as "length". By putting the value at the front and end of the audit record analysis code can easily move forwards and backwards through the audit trail.

3.2.4 Binary XML Stream

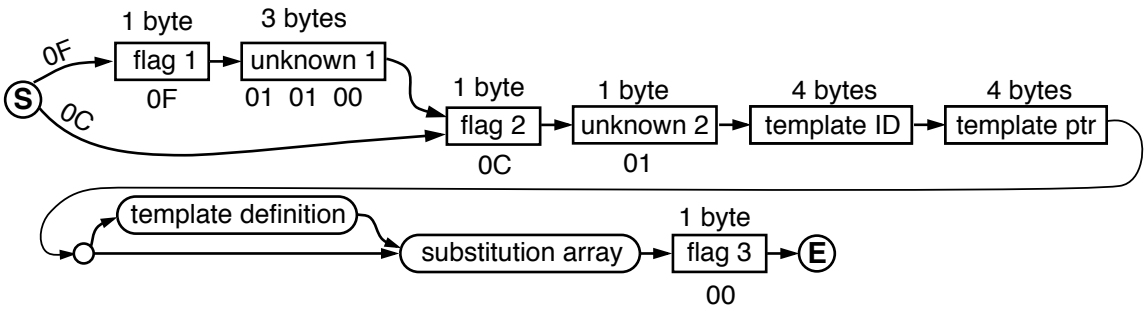

Figure 14: Binary XML Stream

Figure 14 is the standard binary XML representation of an audit record. While Microsoft may have been inspired by other binary implementations of XML such as [Bruc 06], we are not aware of any other implementations that are just like this one. There are also two paths from the start tag: one leading directly to flag 1 and eventually to flag 2, the other jumping directly to flag 2. In almost every instance, the first path (flag 1 then flag 2) is followed. However, there are some cases where the second path is followed. We have not determined how to interpret these two different paths.

- Flag 1, flag 2, & flag 3 – These are, according to [Schu 07], the tokens StartOfBXmlStream, TempateInstance, and EndOfBXmlStream respectively. Unfortunately, we see the TemplateInstance token whether or not a template follows, and as mentioned some binary XML streams do not begin with the StartOfBXmlStream.
- Unknown 1 – Following the 0x0F flag are a sequence of bytes whose purpose is unknown, but they are always the same value. They may represent version numbers.
- Unknown 2 – Following the 0x0C flag is byte whose purpose is unknown; however, the value is always the same.
- Template ID – Each template has a unique ID. This template ID will be the same across all chunks. Each audit record consists of at least one template and one substitution array that are combined to create a full XML object.
- Template ptr – pointer to the template definition in the chunk. If address of this pointer immediately follows, then the template definition immediately follows. Otherwise, the substitution array immediately follows.
- Template definition – the definition of the XML template used by this (and possibly subsequent) audit record(s).
- Substitution array – This contains the values that will be inserted into the previously identified template.

3.2.5 Template Definition

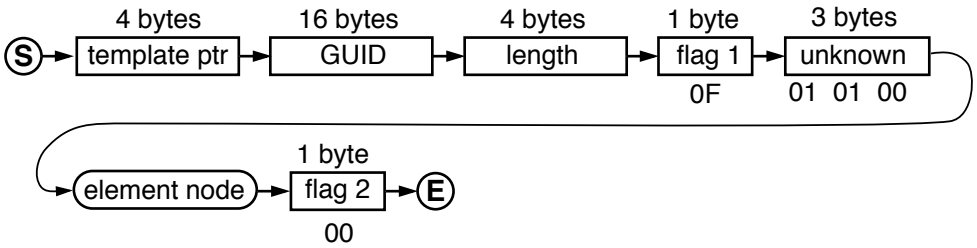

Figure 15: Template Definition

Figure 15 is the template architecture. It consists of some administrative items at the beginning and ending, but the heart of the definition is the XML tree rooted at the element node.

- Template ptr – This is the "next" pointer to another template definition that hashed into the same hash table entry. This is used to form a linked list of template definitions. If this value is zero, there are no more template definitions to follow.
- GUID – global identifier for the template. The first four bytes are the same as the template ID in Section 3.2.4.
- Length – Number of bytes remaining in the template definition. It does not include this length field or the previous 8 bytes.
- Flag 1 & unkown – According to [Schu 07] this is the StartOfBXmlStream token. It is always followed by the same three bytes, the purpose of which we do not know.
- Element node – root of XML tree.
- Flag 2 – According to [Schu 07] this is the EndOfBXmlStream token.

3.2.6 Element Node

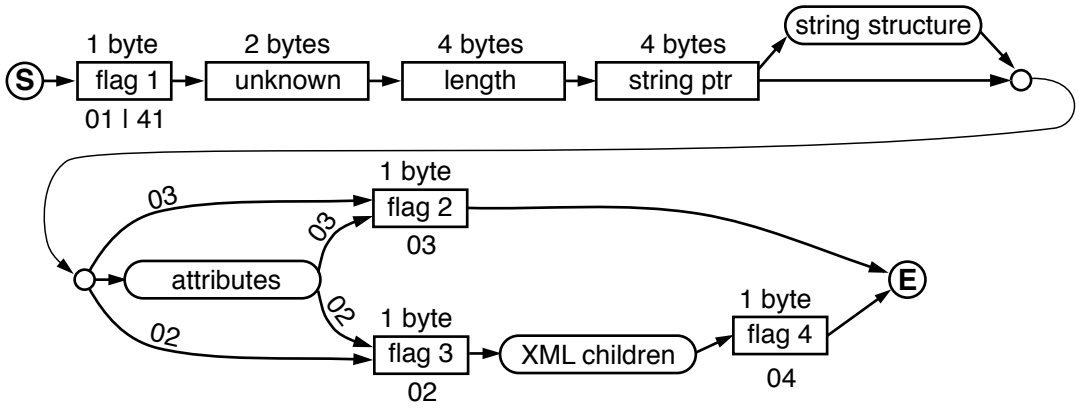

Figure 16: XML Element Node

Figure 16 is the basic XML element. As an example of what this looks like textually see Section 2.2.1.

- Flag 1, flag 2, flag 3, and flag4 – These are the XML tokens. For Flag 1, if the upper 4 bits are set to 4 (i.e., 0x41), then there is at least one attribute.
- Unknown – These two bytes are used inconsistently but as far as we can tell they can be safely ignored. In some templates that have a substitution index for a child, these two bytes have the substitution index as well (i.e., it is redundant). But this is not always the case.
- Length – number of bytes remaining in the node definition.
- String ptr – pointer to the string structure holding the XML element's name. If address of this pointer immediately follows, then the string structure immediately follows.
- Attributes – the XML's attributes (if any).
- XML children – one or more XML children. This can include values that don't change, other XML subtrees, and substitution information.

3.2.7 String Structure

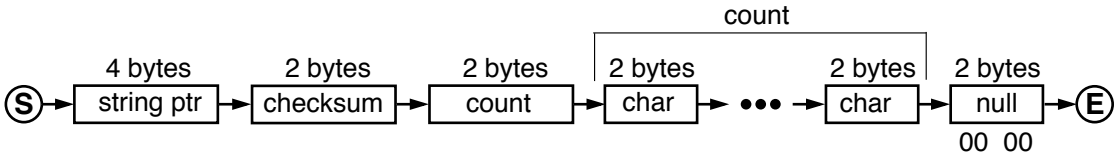

Figure 17: String Structure

Figure 17 is the string structure. It allows us the audit system to only write a string once into a chunk. Every time the same string is needed in a template, it just points back to this definition.

- String ptr – This is the "next" pointer to another string structure that hashed into the same hash table entry. This is used to form a linked list of string structures. If this value is zero, there are no more string structures to follow.
- Checksum – This is checksum of the string. The string table bucket this string hashes to can be found by performing "mod 64" on this checksum.
- Count – This is the number of characters for the string.
- Char – This is a 2-byte, UTF-16 little endian character.
- Null – This is the string terminator.

3.2.8 Attributes

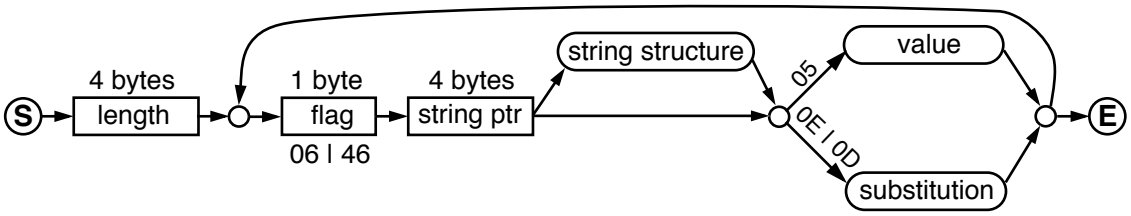

Figure 18: Attributes

Figure 18 is the attribute name-(value or substitution) pairs for an XML element.

- Length – the number of remaining bytes for the attribute section.
- Flag – This value is actually 0x06 with the upper 4-bits set to 4 if there are more attributes to follow this one. [Schu 07] defines this as the Attribute token.
- String ptr – This is the pointer to the string structure holding the attribute's name. If the address of this pointer immediately follows, then the string structure immediately follows.
- Value – This is a fixed value for the attribute.
- Substitution – This is the substitution index information.

3.2.9 XML Children

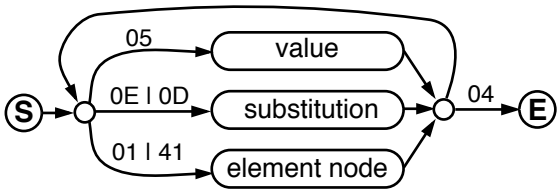

Figure 19: XML Children

Figure 19 represents and XML element's children. There can be a combination of values, substitutions, and other element subtrees. When the look ahead byte is 0x04, the end of the children list has been reached.

- Value – This is a fixed value that doesn't change.
- Substitution – This is information that will be substituted here. It can be a simple value or an entire XML subtree that will be substituted later.
- Element node – This is an XML subtree.

3.2.10 Value

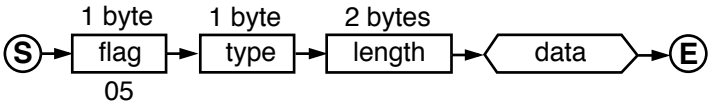

Figure 20: Value

Figure 20 shows how fixed values are defined in a template.

- Flag – This will always be the value 0x05.
- Type – This is the data's type. See Section 2.3.5 for the various values.
- Length – This is the "length" of the data. If the type field is 0x01 (UTF-16 string), then the length is the number of 2-byte characters and not the number of bytes (see Section 2.3.2.2). We have not observed other types other than 0x01, so we don't know how to interpret the length field in these other cases.
- Data – This is some number of bytes.

3.2.11 Substitution

Figure 21: Substitution

Figure 21 is the substitution information in a template definition.

- Flag – This is the flag that indicates this is substitution information. If the flag is 0x0E this is option substitution (i.e., the substitution array may have no value for this field). If this value is 0x0D this is a normal substitution.
- Index – This index into the substitution array whose value will be inserted here. For example, if this value is 7, then the 7th data element in the substitution array will get inserted into this spot in the XML template.

3.2.12 Substitution Array

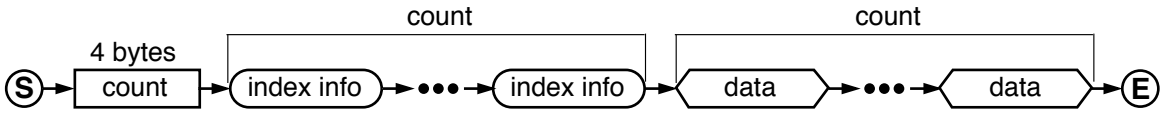

Figure 22: Substitution Array

Figure 22 is the substitution array. This contains the actual data for the audit record.

- Count – This is the number of elements in the substitution table.
- Index info – This is meta information about the data.
- Data – This is the actual data that is inserted into template.

3.2.13 Index Info

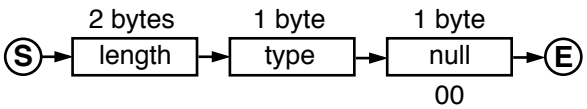

Figure 23: Index Info

Figure 23 is the meta information about the data in the substitution array.

- Length – This is the number of bytes of data for this particular substitution. This value may be zero.
- Type – This tells how those bytes should be interpreted. This field should match the type field in the template's substitution information (see Section 3.2.11). The range of values for this field is listed in Section 2.3.5. A note of caution, usually when the type field is NULL (0x00), then the length field is usually zero, but this is not always the case.
- Null – This is an unused byte.

4 Testing

To test our analysis of the audit trail format we built a parser in C++ running on a Mac (Mac OS 10.6.4). Clearly no Windows APIs were used to help parse the data. We turned on all of the audit flags available in Windows 7. For activity we booted and shutdown the system several times, ran a few programs, and performed some network activity using Internet Explorer. We then saved the existing audit file and exported it to a Mac for analysis. Some of the basic statistics include:

Statistic	Value
File size	1,076,957,184 bytes
Number of chunks	16,419
Number of records	1,426,254
Number of event types	50
Number of templates	48

Figure 24 shows the template that was most frequently used by audit records – over 99.99% of all records used this template. The fixed structure is shown (e.g., element and attribute names), and the fields to be filled in are shown as a number in brackets (optional field) or parentheses (required field). For example the "Provider" element includes an optional attribute named "Guid" that will be the in the 15th field in the substitution array. For this template, all fields are optional. Interestingly, the order of the fields in the audit records (i.e., substitution arrays) have nothing in common with the order in which they appear in the template. For example, the "Correlation" element has two attributes that can be found in the 7th and 13th fields. The 17th field will be filled in with a nested substitution array. The template for this nested template will depend on the EventID.

```
<Event xmlns=http://schemas.microsoft.com/win/2004/08/events/event>
    <System>
        <Provider Name=[14,0x01] Guid=[15,0x0F] />
        <EventID Qualifiers=[4,0x06]>[3,0x06]</EventID>
        <Version>[11,0x04]</Version>
        <Level>[0,0x04]</Level>
        <Task>[2,0x06]</Task>
        <Opcode>[1,0x04]</Opcode>
        <Keywords>[5,0x15]</Keywords>
        <TimeCreated SystemTime=[6,0x11] />
        <EventRecordID>[10,0x0A]</EventRecordID>
        <Correlation ActivityID=[7,0x0F] RelatedActivityID=[13,0x0F] />
        <Execution ProcessID=[8,0x08] ThreadID=[9,0x08] />
        <Channel>[16,0x01]</Channel>
        <Computer>WIN-5L6MN5BLMKU</Computer>
        <Security UserID=[12,0x13] />
    </System>
    [17,0x21]
</Event>
```

Figure 24: Most Frequent Template Used

Figure 25 is the second most frequently used template, but it is always used as a nested value in another template. For example, 34.3% of audit records that use the template shown in Figure 24

have the template in Figure 25 embedded in their 17th field. Figure 26 shows how this second template is embedded into the first template.

```
<EventData>
    <Data Name=SubjectUserSid>(0,0x13)</Data>
    <Data Name=SubjectUserName>(1,0x01)</Data>
    <Data Name=SubjectDomainName>(2,0x01)</Data>
    <Data Name=SubjectLogonId>(3,0x15)</Data>
    <Data Name=ObjectServer>(4,0x01)</Data>
    <Data Name=ObjectType>(5,0x01)</Data>
    <Data Name=ObjectName>(6,0x01)</Data>
    <Data Name=HandleId>(7,0x10)</Data>
    <Data Name=AccessList>(8,0x01)</Data>
    <Data Name=AccessMask>(9,0x14)</Data>
    <Data Name=ProcessId>(10,0x10)</Data>
    <Data Name=ProcessName>(11,0x01)</Data>
</EventData>
```

Figure 25: Second Most Frequent Template Used

```
<Event xmlns=http://schemas.microsoft.com/win/2004/08/events/event>
    <System>
        <Provider Name=[14,0x01] Guid=[15,0x0F] />
        <EventID Qualifiers=[4,0x06]>[3,0x06]</EventID>
        <Version>[11,0x04]</Version>
        <Level>[0,0x04]</Level>
        <Task>[2,0x06]</Task>
        <Opcode>[1,0x04]</Opcode>
        <Keywords>[5,0x15]</Keywords>
        <TimeCreated SystemTime=[6,0x11] />
        <EventRecordID>[10,0x0A]</EventRecordID>
        <Correlation ActivityID=[7,0x0F] RelatedActivityID=[13,0x0F] />
        <Execution ProcessID=[8,0x08] ThreadID=[9,0x08] />
        <Channel>[16,0x01]</Channel>
        <Computer>WIN-5L6MN5BLMKU</Computer>
        <Security UserID=[12,0x13] />
    </System>
    <EventData>
        <Data Name=SubjectUserSid>(0,0x13)</Data>
        <Data Name=SubjectUserName>(1,0x01)</Data>
        <Data Name=SubjectDomainName>(2,0x01)</Data>
        <Data Name=SubjectLogonId>(3,0x15)</Data>
        <Data Name=ObjectServer>(4,0x01)</Data>
        <Data Name=ObjectType>(5,0x01)</Data>
        <Data Name=ObjectName>(6,0x01)</Data>
        <Data Name=HandleId>(7,0x10)</Data>
        <Data Name=AccessList>(8,0x01)</Data>
        <Data Name=AccessMask>(9,0x14)</Data>
        <Data Name=ProcessId>(10,0x10)</Data>
        <Data Name=ProcessName>(11,0x01)</Data>
    </EventData>
</Event>
```

Figure 26: Nested Template

5 Conclusions

Windows 7 audit data can be a powerful tool for understanding a wide range of threats on your computer system. Unfortunately are very few tools available to analyze this data, and the Windows supplied tool, Event Viewer, has limited functionality. One reason that there are so few tools and so few people taking advantage of the capabilities provided by the audit trail is that the audit

trail format is poorly documented. This goal of this paper is to provide a detailed analysis of the file format, so that engineers can build their own parsers of the data.

We used Andreas Schuster's paper [Schu 07] as a jumping off point and then performed our own exhaustive analysis of the binary data in the audit log. To avoid any risk of non-disclosure issues, we never looked at any Windows code during our analysis. One side effect of this data-only reverse engineering is that we had to guess at the meaning of some of the data, and in a few cases we could not understand why the field held particular values. However, so far these few instances have not had negative effects on our parsing of the data.

6 References

[Bruc 06] Craig Bruce, "Binary Extensible Markup Language (BXML) Encoding Specification", Open Geospatial Consortium, Inc., OGC 03-002r9, 13 Jan 2006.

[Schu 07] Andreas Schuster, "Introducing the Microsoft Vista event log file format", Digital Investigation, Volume 4, Supplement 1, pp. 65-72, Sep 2007.