available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Introducing the Microsoft Vista event log file format

Andreas Schuster

Deutsche Telekom AG, Friedrich-Ebert-Allee 140, D-53113 Bonn, Germany

Keywords:
Digital evidence
Forensic examination
Microsoft Windows Vista
Windows event logging
Log file
evtx

ABSTRACT

Several operating systems provide a central logging service which collects event messages from the kernel and applications, filters them and writes them into log files. Since more than a decade such a system service exists in Microsoft Windows NT. Its file format is well understood and supported by forensic software. Microsoft Vista introduces an event logging service which entirely got newly designed. This confronts forensic examiners and software authors with unfamiliar system behavior and a new, widely undocumented file format.

This article describes the history of Windows system loggers, what has been changed over time and for what reason. It compares Vista log files in their native binary form and in a textual form. Based on the results, this paper for the first time publicly describes the key-elements of the new log file format and the proprietary binary encoding of XML. It discusses the problems that may arise during daily work. Finally it proposes a procedure for how to recover information from log fragments. During a criminal investigation this procedure was successfully applied to recover information from a corrupted event log.

1. Introduction

Log files belong to the most important sources of forensic information because they usually connect a certain event to a point in time. Major operating systems like UNIX or Microsoft Windows[1] provide system-wide services to collect, process and store event messages. Section 1.1 gives a brief introduction into UNIX syslog and the history of Microsoft Windows event logging services.

The structure of a Microsoft Vista event log file down to a single record will be documented in Section 2. Section 3 will explain the key concepts of Microsoft's binary XML, which forms the main part of an event record.

Finally Section 4 looks at the impact of the new event log file format on daily computer forensic work. This section also provides the examiner with a process to extract data from single records which were recovered from unallocated clusters or file slack.

1.1. *Event logging services*

1.1.1. *Syslog*

UNIX syslog is one of the most commonly found system loggers. The daemon accepts incoming messages from local processes and from the network. Based on the *facility* the message is originating from and the *severity* of the event the message can be written to the console, stored into a file or sent to another syslog daemon on another host over the network (Lonvick, 2001). This flexibility allows to build extensive logging architectures. System logging services derived from the original syslog provide enhanced filtering capabilities and transmit log messages through secured communication channels (Scheidler).

A typical log message consists of a constant text explaining the kind of reported event. This is complemented by variable data which provide the specifics for the individual event. The constant part comes as a format string which also defines the type (i.e. string, integer) and the position of the variable parts.

E-mail address: andreas.schuster@telekom.de

[1] This article only covers the versions of Microsoft Windows which are based on either the NT or Vista kernel. Versions based on MS-DOS do not provide a logging service.

1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.015

The following example shows how a log message gets constructed from its constant part and a single variable part, the user name:

```
syslog (LOG_DEBUG, "password changed for user %s \n", uname);
```

Then the syslog daemon would record an entry like the following in the appropriate log file:

```
Feb 26 10:18:13 mycomputer passwd: password changed for user joe
```

Note that constant and variable parts get bound together before the message is sent to the syslog service. Each event message results in a single line of plain text written into a log file. So each log record is self-contained.

1.1.2. *Microsoft Windows NT*

When Microsoft Windows NT 3.5 was released in 1994, it was shipped with an event logging service, which at that time was a novelty for the Microsoft Windows family of operating systems. Unlike syslog, the NT event logging service can receive messages only from local processes. Also it is unable to forward messages to other instances through a network.

When compared to syslog, the major difference, however, is that constant and variable parts of a message are joined at the time of viewing. While this might be surprising at first, there is a good reason for it: obviously syslog style log files contain a lot of redundant information. If in the example above three users would change their passwords this would result in three lines containing the phrase ''password changed for user''. On Windows NT the constant part of this message would be moved into a message table resource of a suitable executable file. Then this file would be registered with the event log service as an event *source*.

Now instead of a string the index and the source will be sent to the logging service along with the variable data. The service then stores all of that information in a binary format (Microsoft Corporation, 2007a).

As soon as a record is requested, the event log service will look up the file holding the message table from the source given in the event record. Then it uses the EventID, which is also given in the record, as an index into the message table and retrieves the format string. Finally the service combines the format string and the variable parts to form the message text.

This design effectively cuts down the size of log space required by eliminating large and possibly redundant strings from the log file. It also supports localized log messages: events recorded by an English version of Windows NT can be read in another language on a localized version of Windows.

If the original message tables are unavailable during an investigation, the examiner still can look up the constant part from conversion tables generated by suitable tools (Morgan, 2005), documentation provided by the vendor (Microsoft Corporation, 2007b) or specialized look-up services (Altair Technologies, 2001; Prism Microsystems, 1999).

So even though the event record does not contain the plain text message, it still provides enough information to reconstruct its meaning.

1.1.3. *Microsoft Windows Vista*

While it was in use over more than a decade from NT 3.5 to Windows Server 2003, several problems were discovered in the event log service. So Microsoft came up with a new design which was code-named *Crimson* to go with Windows Vista. It was renamed to *Windows event logging* as soon as Vista was released.

One major drawback of the NT event log service is its need to map the whole log file into memory (Hess et al.). Large files waste precious address space in a region which is also used for inter-process communication and shared memory. This could impose a problem on busy domain controllers and exchange servers. According to Microsoft the total size of all active event logs should not exceed 300 MiB[2] (Microsoft Corporation, 2007c, 2005).

The new log file format consists of a small file header (see Section 2.1) which is followed by a series of chunks (described in Section 2.2). Chunks are self-contained. No event record will extend over the boundary between two chunks. So for every event log only the current chunk (64 kiB) and the file header (4 kiB) have to be mapped into memory. This significantly reduces the impact on system resources.

The NT event log service is unable to filter records based on the message text or the binary data which are associated with an event record. Conversely, Windows event logging is based on XML technology. This allows for queries based on the XPath language (Clark and DeRose, 1999). XML also has the benefit of flexible output customization through XML Style Sheets and Transformations.

One of the drawbacks of XML is the high amount of computational resources, CPU cycles and memory, which are needed to parse the file format. Also XML is known to contain lots of redundant text, thus wasting disk space. Microsoft, like others before, mitigated this disadvantage through a binary encoding of XML. This will be covered in-depth in Section 3.

1.2. *Method*

There is no publicly available and authoritative documentation completely covering the Windows event log file format. So the most definitive source of information would be a disassembly of `wevtsvc.dll`, the library implementing the event log service. However, publicizing the results of any sort of decompilation is interdicted in the jurisdiction the author lives in.

So a clean-room analysis was conducted. For this Vista Ultimate version 6.0.6000.16386 was installed with default settings. No additional applications were installed. The applications shipped with Vista Ultimate were tried out for some time. The system was also connected with multiple wired and wireless networks. While connected to networks, the system was not exposed to targeted attacks. Afterwards the logs were exported in their native and XML format through the Event Viewer applet. This resulted in 47 log files, of which 30 files were empty. The

[2] This article follows the notation approved by the International Electrotechnical Commission to indicate a binary multiple, see also National Institute of Standards and Technology.

remaining 17 files contained 2616 event records. Five files consisted of more than a single chunk.[3]

Then both the binary and the textual representation of an event log were compared. Structures and data types discovered were implemented in a recursive-descent parser. Then the next file was fed to the parser, which marked any block of data that it was unable to parse. These blocks were compared to the corresponding XML, leading to the discovery of new structures, and so on. The iterative process came to an end as soon as no new structures could be identified in all available log files.

Additionally single log files were obtained in their native form under less common conditions. These were later converted through the Event Viewer applet into their textual form.

(1) A log file was extracted through 010 Editor version 2.1.3 (Sweetscape Software, 2007) while the file was opened by the event log service.
(2) The Application log was configured for minimum size and to reuse space as needed. Then the log was flooded with events by means of `logevent.exe` (Microsoft Corporation, 2006), forcing it to wrap around and overwrite the oldest records.
(3) The Application log was configured for minimal size and automatic maintenance mode. Then the log was flooded with events, forcing the service to save the file and to continue operation with a blank file.
(4) The Application log was configured for minimal size and manual maintenance mode. Then the log was flooded until it refused to accept new messages.

The log files that were obtained through these experiments helped to improve the understanding of several fields in the file and chunk headers.

2. Log file format

2.1. File

The file header provides some basic information about the log file (see Table 1). The magic string ''ElfFile'' along with a version of 3.1 identify the file as a Windows event logging file. While the header consists of 4096 bytes, only 128 bytes are in use. The header's integrity is protected by a 32 bit check sum.

The header states the count of chunks in the log file. Examiners should be aware that the actual file size might be much larger than the size calculated from the header. The service seemingly pre-allocates free space suitable to hold several chunks in case of frequently used log files like the Application, Security and System logs.

Also given is the number of the chunk which is currently in use. Chunk numbers are zero-based. If the current chunk is not also the last chunk, then the oldest records were overwritten due to the retention policy.

[3] For an explanation of the term ''chunk'' please refer to Section 2.2.

Table 1 – File header

No.	Ofs	Len	Meaning
1	0x00	8	Magic string ''ElfFile'' 0x00
2	0x10	8	No. of current chunk
3	0x18	8	No. of next record
4	0x20	4	Header space used, constant 0x80
5	0x24	2	Minor version, constant 1
6	0x26	2	Major version, constant 3
7	0x28	2	Size of header, constant 4096
8	0x2a	2	Chunk count
9	0x78	4	Flags
10	0x7c	4	Check sum

Bit 0 of the *flags* indicates a dirty log. If set, the log has been opened and was changed, though not all changes might be reflected in the file header. If a dirty log is re-opened by the event logging service, it will scan through all the chunks and attempt to update the file header accordingly. Thereafter it will clear the flag and adjust the check sum.

Bit 1 of the *flags* indicates a full log. This flag is set if the log has reached its maximum configured size and the retention policy in effect does not allow to reclaim a suitable amount of space from the oldest records and an event message could not be written to the log file. For a forensic examiner this means that some information of possible evidential value was not recorded.

2.2. Chunk

Besides the file header only the current chunk needs to be mapped into memory for operation. Every chunk consists of a small header, hashed tables of strings and XML templates and finally a series of event records.

The chunk header is 128 bytes in size. It starts with the magic string ''ElfChnk'', which allows for easy identification of chunks. Pointers indicate the beginnings of the last record and free space. Again the integrity is protected by a 32 bit check sum (see Table 2).

Seemingly the new event logging service now uses two different counters for record numbers. The difference becomes evident if an event log has been configured for automatic maintenance mode. As soon as the file reaches its maximum size the event log service renames the file and creates a new

Table 2 – Chunk header

No.	Ofs	Len	Meaning
1	0x00	8	Magic string ''ElfChnk'' 0x00
2	0x08	8	Number of first record in log
3	0x10	8	Number of last record in log
4	0x18	8	Number of first record in file
5	0x20	8	Number of last record in file
6	0x28	4	Size of header
7	0x2c	4	Offset of last record
8	0x30	4	Offset of next record
9	0x7c	4	Check sum

one under the old name. Now record-based counting continues, while the file-based counter gets reset. So far clearing the log seems to be the only way to reset the log-based counter.

The chunk header is immediately followed by the string table. For every string a 16 bit hash is calculated. The hash value is divided by 64, the number of buckets in the string table. The remainder then indicates what hash bucket to use. Every bucket contains the 32 bit offset relative to the chunk where the string can be found. If a hash collision occurs, the offset of the last string will be stored in the bucket. The string object will then provide the offset of the preceding string, thus building a single-linked list.

In a similar way XML templates are handled by the template table. This table consists of 32 buckets.

Both tables help the service to avoid redundant definitions of string and template objects while it writes records into the log file. Both tables are of limited use during the reconstruction phase, though. A decoding parser would preferably build its own set of tables, based on the object's offsets instead of their hash values.

2.3. *Event record*

The event record merely is a wrapper for the event message, which is encoded in a proprietary, binary form of XML. Details regarding the encoding are provided in Section 3.

Matching length indications at the beginning and the end of the event record allow the service to traverse the list of records forward and backwards in an efficient manner (see Table 3).

Two important values, the record number as well as the date and time the event message was created are also provided outside of the XML structure. This allows to sort event messages in their chronological and numerical order without the burden to parse the whole XML data.

3. Binary XML

3.1. *Schema*

Microsoft provides a detailed documentation of the XML schema for event records in the Microsoft Developers Network (MSDN) (Microsoft Corporation, 2007d). Accordingly the *Events* element is the top-level container. It keeps all of the *Event* elements, each of them describes a single event (see Fig. 1).

Each record is required to start with a *System* element. This element is ''automatically populated by the system if the event is raised or when it is saved into the log file'' (Microsoft Corporation, 2007e). It provides a lot of basic information, among them the record number, the name of the computer it originates from, time stamp, the subsystem and a number describing the event (Microsoft Corporation, 2007f).

Additionally there may appear one out of the following elements: BinaryEventData, DebugData, EventData, ProcessingErrorData, RenderingInfo and UserData. Among these the element which is most likely to be found in production logs will be *EventData*. It contains a sub-element for every parameter passed by the originating subsystem. If the event was received through the NT compatible API then it may also contain a block of binary data.

Table 3 – Event record

No.	Ofs	Len	Meaning
1	0x00	4	Magic string ''**'' 0x00 0x00
2	0x04	4	Record length
3	0x08	8	EventRecordID
4	0x10	8	TimeCreated (FILETIME)
5	0x18	var.	Event message, binary XML
6	var.	4	Length (repeated)

3.2. *Concepts*

For an event log service the most important design goals are to minimize the disk space needed to store a message and to minimize the CPU time needed to write and, even more important, to read these records.

Obviously XML files do not fit well within these requirements. They are known to contain lots of redundant data; the deflate algorithm (Deutsch, 1996) was able to reduce the author's set of XML test files to less than 10% of their original size. Also a significant amount of CPU time is needed to parse the text format.

This problem has already been solved by engineers working in the field of mobile devices. Though the solutions differ slightly, they all are based on a binary representation of XML. Where documentation is publicly available, it also adds to a better understanding of the measures which were implemented by Microsoft (Bruce, 2006; Martin and Jano, 1999). Three key concepts will be discussed below.

3.2.1. *Tokenization*

According to Bruce (2006, p. 25) tokens are ''used to encode the content of the [...] file in discrete 'packets' that correspond roughly to XML markups''. In case of the Windows event log files tokens can be further differentiated into *system tokens* and *application tokens*. System tokens are hard-coded into the executables which produce and consume binary XML. So system tokens provide a static mapping between a number and an associated function. Application tokens refer to application-specific entities like the names of elements and attributes (NameStrings, see Fig. 2) and constant blocks of XML statements (Templates, see Section 3.2.3).

```
<Events>
  <Event>
    <System>...</System>
    <EventData>...</EventData>
  </Event>
  <Event>
    ...
  </Event>
</Events>
```

Fig. 1 – Structure of an XML event log file.

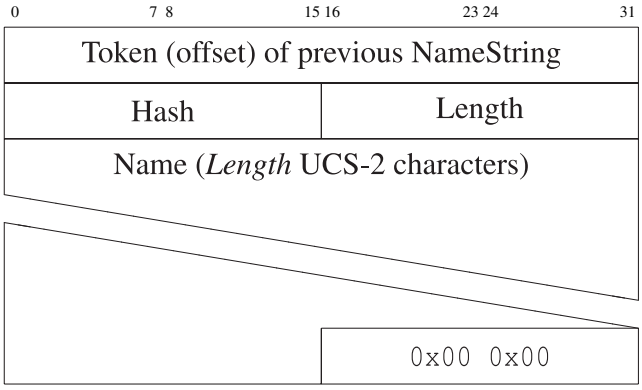

Fig. 2 – NameString structure.

Microsoft implemented system tokens as byte values. The lower nibble determines the function (see Table 4), while the upper nibble contains flags (compare to Martin and Jano, 1999). So far only one flag has been discovered: `0x40` indicates that at least one attribute will follow the XML tag (Table 4).

A group of system tokens marks start tags, end tags and empty element tags as defined by the W3C (Bray et al., 2006, Section 3.1). Other system tokens indicate attributes and values or mark the beginning and end of the binary XML stream.

The main advantage of translating XML structures into system tokens is a gain in speed. Several time-consuming calculations have to be done only once, at the time of translating the document from its textual into the binary form. This includes the check for shapeliness and the calculation of block lengths.

So far the tokenization process does not provide a significant gain in storage efficiency. This is mainly achieved through application tokens. An application token denotes the offset where the corresponding object has been defined. In case of the first reference made, the application token points to the nearest free space suitable to hold the object's data. Later references to the same object will point back to that position. So if the application token's value is lower than the current offset, the object already has been defined.

Table 4 – System token codes

Value	Meaning	Example
0x00	EndOfBXmlStream	
0x01	OpenStartElementTag	<name>
0x02	CloseStartElementTag	<name>
0x03	CloseEmptyElementTag	<name/>
0x04	EndElementTag	</name>
0x05	Value	attribute="value"
0x06	Attribute	attribute="value"
0x0c	TemplateInstance	
0x0d	NormalSubstitution	
0x0e	OptionalSubstitution	
0x0f	StartOfBXmlStream	

3.2.2. *Substitution*

The substitution mechanism allows to segregate structure from content.

For example every log record starts with a *System* element, which will look almost the same for every record. The varying parts, the contents of XML elements and the values of attributes, can be replaced with a substitution token and moved into a *SubstitutionArray*. The token then points to the array's element which stores the varying data. The token also informs about the data type (as defined in Microsoft Corporation, 2007g).

The SubstitutionArray is the last structure in a record's binary XML stream. For every entry it lists the size and data type. This is succeeded by the actual data for every non-empty element (see Fig. 3).

There are two different kinds of substitution tokens. A *normal substitution* token will unconditionally insert the data from the SubstitutionArray into the XML structure.

But XML is also known to support optional elements and attributes. This case will be handled by *optional substitution* tokens. If the substitution array's element the token is pointing to is not empty, then the optional substitution will work like a normal substitution. However, if the element is empty, which will be indicated by a type of *EvtVarTypeNull*, then the optional substitution will suppress the corresponding element or attribute.

3.2.3. *Templating*

As mentioned earlier, every event message will start with a *System* element. The variable parts were already factored out by the substitution mechanism discussed above. Optional substitutions also take care of optional elements. So, finally, every event record will start with the same sequence of tokens.

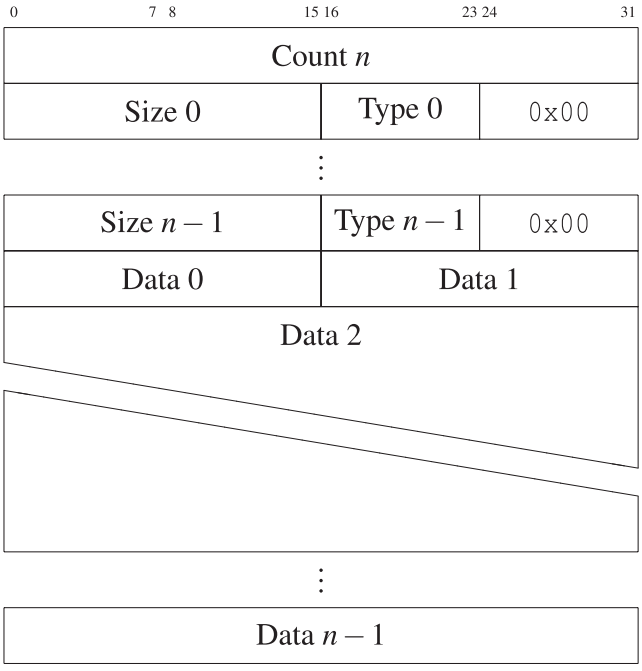

Fig. 3 – SubstitutionArray structure.

This never changing sequence of binary XML statements will be called a *Template*. Henceforth it is represented by an application token. Whenever needed, a template can be instantiated through the *TemplateInstance* system token in conjunction with the proper application token. Obviously the use of templates dramatically reduces the demands for storage space. A sample template for the *System* element is shown in Fig. 4.

From the same template every record will produce a slightly different XML structure, controlled by the data in its substitution array. The substitution array frequently contains other XML structures or refers to other templates. For example the *EventData* element gets appended to a record's *System* element this way. Unfortunately the corresponding data type 33 is not documented by Microsoft.

The log files analyzed during this study defined between 10 and 20 different templates per chunk. Of course the exact numbers depend on the degree of freedom allowed by the log file's XML schema. So for example more templates could be found in the Application log than in the Security log or a log file dedicated to a single application.

4. Impact on the forensic practice

Log files generally are an important source of information in a forensic examination. This also applies to log files created by Windows event logging, possibly more than ever before: the new service promises a lot more of flexibility and less impact on system resources. So, hopefully, it will find wide acceptance among application developers and system administrators.

Months after Microsoft Vista became publicly available, still none of the major forensic tools supports the new log file format. Until now the only means to analyze a given binary Vista log file is to open it in the Event Viewer applet and convert it into a textual form.

Fortunately the Windows event logging service seems to be more robust against light cases of file corruption than its predecessor. During an investigation a computer running Microsoft Vista was shut down ungracefully. The log files showed slight signs of distortion, for example the record count as reported in the header did not match what could be found in the chunks. The new logging service still was able to open the files in most cases. Seemingly triggered by the header's *dirty* flag, the service scanned the whole file and corrected chunk and record counts accordingly.

Unfortunately this robustness will not help if only parts of a deleted or corrupted log file can be recovered from a system. The remainder of this section will provide some guidance how parts of Vista log files can be recovered through file carving. It also describes a technique how to extract a basic set of information from single event records and chunk fragments.

4.1. *File carving*

Unique magic strings and a block-oriented file layout will help a forensic examiner to carve out the file header and chunks of Vista event log files from a raw file system image.

```
<Event xmlns="...">
<System>
<Provider Name="..." Guid="..." />
<EventID Qualifiers="#slot 4, type 6, optional#">
   #slot 3, type 6, optional#
</EventID>
<Version>#slot 11, type 4, optional#</Version>
<Level>#slot 0, type 4, optional#</Level>
<Task>#slot 2, type 6, optional#</Task>
<Opcode>#slot 1, type 4, optional#</Opcode>
<Keywords>#slot 5, type 21, optional#</Keywords>
<TimeCreated SystemTime="#slot 6, type 17, optional#" />
<EventRecordID>#slot 10, type 10, optional#</EventRecordID>
<Correlation ActivityID="#slot 7, type 15, optional#"
   RelatedActivityID="#slot 18, type 15, optional#" />
<Execution ProcessID="#slot 8, type 8, optional#"
   ThreadID="#slot 9, type 8, optional#" />
<Channel>...</Channel>
<Computer>...</Computer>
<Security UserID="#slot 12, type 19, optional#" />
</System>
   #slot 19, type 33, optional#
</Event>
```

Fig. 4 – The XML template with unresolved substitutions.

The file header and chunks are expected to start at a sector boundary. They are easily identifiable through their magic strings ''ElfFile'' and ''ElfChunk'', respectively. The blocks to be carved out are always 4096 bytes and 65,536 bytes in size.

The file header will be of little forensic value, though. The flags tell whether the file was properly closed and if data were lost due to exceeding the available log space.

The predominant part of information is contained within the chunks. Again the information provided by the chunk header might be useful, but strictly speaking it is not required to transform the binary data in a textual representation. This also applies to the NameString and Template tables following the header.

Single event records are easy to spot by their 32 bit magic string `0x2a 0x2a 0x00 0x00`. The following length indication could instruct a carver how much data need to be processed. The matching length field at the record's end allows for a speedy verification.

The last event record found in a chunk may be incomplete. Obviously the service starts writing the event record to file while it is still working on it. If the remaining space does not suffice to complete the record, the service will close the chunk and open a new one. Then it will write the record from its beginning.

4.2. *Transformation into a textual form*

So far the recovered data are in a binary form, which generally will be of little help in an investigation. A transformation into a textual representation, be it either the explanatory message shown by the Event Viewer applet or the textual XML, is badly needed.

In order to transform binary XML into a textual form the author developed a simple recursive-descent parser (Schuster, 2007a). While processing all the event records of a chunk the parser builds NameString and Template tables on-the-fly as it encounters suitable definitions. The produced XML is similar to what will be generated by the Vista Event Viewer applet. In contrast to the system tools provided by Microsoft this parser is not bound to the Microsoft Vista platform. Furthermore it is able to operate on single chunks.

4.3. *Recovering single records*

Even if event log files already were deleted and partially overwritten, the examiner may still recover single records from unallocated clusters or file slack. Transforming an isolated record into its textual form unveils a problem: the record is likely to refer to NameStrings and XML templates which have been defined earlier and now are not available for analysis. So strictly speaking it is impossible to transform the binary data into text under that conditions.

In this situation an educated guess may help: as already mentioned the XML schema requires every event record to start with a *System* element containing a well-defined set of other elements. This part of an event record is populated by the operating system and beyond control of the application programmer. It is safe to assume, and backed by observation, that the mapping between variable elements in the XML structure and slots in the SubstitutionArray is constant.

The parser was modified to provide information about the mapping instead of actually performing the substitution. This produced output as shown in Fig. 4.

Of course there is no guarantee that this particular mapping will remain unchanged in future versions of Microsoft Vista. So an examiner should carefully validate the correlation using either intact log files which were found on the system or which were obtained from a clean system running the very same version of Microsoft Vista.

The next step is to locate the SubstitutionArray that is associated with the *System* element. Every record will start with a similar byte sequence. First comes the *StartOfBXmlStream* system token, which consists of a byte sequence of `0x0f 0x01 0x01 0x00`. Next is the *TemplateInstance* system token (`0x0c`), followed by an unknown byte (usually `0x01`), the TemplateID (4 bytes) and the template's application token (4 bytes).

Usually the *System* element will not be (re-)defined. The next four bytes will match the SubtitutionArray's element count (20 elements, that is `0x14 0x00 0x00 0x00`), thus marking the array's location.

However, if the next four bytes contain a different value (usually just null bytes) and are followed by four bytes repeating the TemplateID, then the examiner has to skip over the embedded binary XML stream. The proper way to do so would be to interpret the stream until one reaches the *EndOfBXmlStream* system token.

As a rule of thumb the examiner can skip bytes until he/she reaches a sequence of `0x04 0x04 0x00 0x14 0x00 0x00 0x00`. This translates into the two *EndElementTag* system tokens for </System> and </Event>, followed by the *EndOfBXmlStream* marker. The next four bytes already belong to the SubstitutionArray.

The examiner now can interpret the SubstitutionArray according to the structure laid out in Fig. 3. Two values, EventRecordID and CreateTime, which are given in the event record structure as well as in slots 10 and 6 of the SubstitutionArray allow for further cross-checking. While this can be done by hand, a hex editor with templating capabilities will help to speed up the process (Schuster, 2007b).

This procedure can recover most of the information that is known from the days of the NT event log. Unfortunately it does not enable an examiner to find out the subsystem which created the event message, because the *Provider Name* and *GUID* strings are not kept in the SubstitutionArray but in the template. Depending on the case these strings may be found in an earlier record.

5. Conclusion and future work

Windows event logging, the new event logger service of Microsoft Windows Vista, is expected to consume less resources than its predecessor. Therefore one can hope to find more event data than ever before. Without question this will be beneficial to a forensic examination. And also without question the undocumented, proprietary binary XML format that Microsoft designed will be a major obstacle in accessing all of this information. Because event records are no longer self-contained but now depend on information stored elsewhere, it may be almost impossible to fully decode an event record which has been carved from unallocated clusters or file slack.

Through the comparison of Vista event log files in their binary and textual form the author gained sufficient knowledge in order to develop a parser which transforms slightly damaged log files and single chunks into plain XML. A slightly modified version of this parser was successfully used to reconstruct a corrupted log file during an investigation conducted by the Criminal Police at Lörrach in south-western Germany.

Constructing a static mapping between parts of the XML structure and slots in the SubstitutionArray can help to retrieve at least basic information from single event records.

Further research is needed to unveil more elements of Microsoft's binary XML. The ''holes'' in the table of system token codes are obvious. When compared to other implementations of binary XML and the XML standard, language elements like CDATA sections and processing instructions seem to be missing. Also some data types that were documented by Microsoft could not be found in the test data.

Several system binaries of Windows Vista contain WEVT_TEMPLATE resources. Further examination shows that the resources refer to ''Crimson'', the former code name of Windows event logging, as well as to other elements of the new logging architecture like template, opcode, level, task and keyword. Potentially this could be helpful in reconstructing even more information from single event records.

REFERENCES

Altair Technologies Ltd. eventid.net, <http://www.eventid.net/>; 2001 [accessed 2007-03-18].

Bray Tim, Paoli Jean, Sperberg-McQueen CM, Maler Eve, Yergeau François. Extensible Markup Language (XML) 1.0. W3C recommendation, World Wide Web Consortium; 29 September 2006. <http://www.w3.org/TR/2006/REC-xml-20060816/> [accessed 2007-03-18].

Bruce Craig. Binary Extensible Markup Language (BXML) Encoding Specification. Technical Report OGC 03-002r9. Open Geospatial Consortium Inc.; 13 January 2006. <http://portal.opengeospatial.org/files/?artifact_id=13636> [accessed 2007-03-18].

Clark James, DeRose Steve. XML Path Language (XPath), Version 1.0. W3C recommendation, World Wide Web Consortium; 16 November 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116>.

Deutsch L. Peter. DEFLATE compressed data format specification version 1.3. RFC 1951; May 1996. <ftp://ftp.rfc-editor.org/in-notes/rfc1951.pdf> [accessed 2007-03-18].

Hess Robert, Sutton Alex, Hough Marty. Management services. The .NET show (52). <http://msdn.microsoft.com/theshow/transcripts/Episode52Transcript.aspx> [accessed 2007-03-18].

Lonvick Chris. The BSD syslog protocol. RFC 3164; August 2001. <ftp://ftp.rfc-editor.org/in-notes/rfc3164.txt> [accessed 2007-03-18].

Martin Bruce, Jano Bashar. WAP Binary XML Content Format. W3C Note, World Wide Web Consortium; 24 June 1999. <http://www.w3.org/1999/06/NOTE-wbxml-19990624> [accessed 2007-03-18].

Microsoft Corporation, Redmond. Threats and countermeasures. Chapter 6: Event Log, <http://www.microsoft.com/technet/security/guidance/serversecurity/tcg/tcgch06n.mspx>; 27 December 2005 [accessed 2007-03-18].

Microsoft Corporation, Redmond. How to use the event logging utility (Logevent.exe) to create and log custom events in Event Viewer in Windows 2000, <http://support.microsoft.com/kb/315410/en-us/>; 28 February 2006 [accessed 2007-03-18].

Mircrosoft Corporation, Redmond. EVENTLOGRECORD, <http://msdn2.microsoft.com/en-us/library/aa363646.aspx>; 2007a [accessed 2007-03-18].

Microsoft Corporation, Redmond. Events and Errors Message Center, <http://www.microsoft.com/technet/support/ee/ee_advanced.aspx>; 2007b [accessed 2007-03-18].

Microsoft Corporation, Redmond. Event log may not grow to configured size, <http://support.microsoft.com/kb/183097/en-us>; 22 January 2007c [accessed 2007-03-18].

Microsoft Corporation, Redmond. Event schema, <http://msdn2.microsoft.com/en-us/library/aa385201.aspx>; 2007d [accessed 2007-03-18].

Microsoft Corporation, Redmond. EventType complex type, <http://msdn2.microsoft.com/en-us/library/aa384584.aspx>; 2007e [accessed 2007-03-18].

Microsoft Corporation, Redmond. SystemPropertiesType complex type, <http://msdn2.microsoft.com/en-us/library/aa385206.aspx>; 2007f [accessed 2007-03-18].

Microsoft Corporation, Redmond. EVT_VARIANT_TYPE, http://msdn2.microsoft.com/en-us/library/aa385616.aspx; 2007g [accessed 2007-03-18].

Morgan Timothy. GrokEVT, <http://projects.sentinelchicken.org/grokevt/>; 2005 [accessed 2007-03-18].

National Institute of Standards and Technology, Gaithersburg, MD. Prefixes for binary multiples, <http://physics.nist.gov/cuu/Units/binary.html> [accessed 2007-03-18].

Prism Microsystems, Inc. EventTracker knowledge base, <http://www.evtcatalog.com/>; 1999 [accessed 2007-03-18].

Scheidler Balazs. syslog-ng. BalaBit IT Ltd. <http://www.balabit.com/products/syslog_ng/> [accessed 2007-03-18].

Schuster Andreas. A parser to transform vista event log files into plain text, <http://computer.forensikblog.de/en/2007/08/.evtx_parser.html>; 10 August 2007a.

Schuster Andreas. A template to parse substitution arrays, <http://computer.forensikblog.de/en/2007/08/evtx_substarray_template.html>; 1 August 2007b.

Sweetscape Software. 010 Editor, <http://www.sweetscape.com/010editor/>; 2007 [accessed 2007-03-18].

Andreas Schuster is a Computer Forensic Specialist with the security department of Deutsche Telekom AG since December 2003. Previously he led a commercial computer incident response team and had worked in the Internet business for about seven years. He had got his first computer in 1981. In order to make the most out of 1024 bytes of main memory he had to acquire low-level programming skills. Though times have significantly changed he regularly falls back to low-level tools like disassemblers and hex editors when he explores the inner mechanics of an operating system or a new piece of malware.