

assuria

Effective Log Formats

1 Introduction

Most SIM / SIEM solutions require all non-textual log data (e.g. Windows Event Logs) to be rewritten into a simple textual format such as syslog. By contrast, Assuria Log Manager (ALM) is a forensically-sound log management system, because it collects logs in their original format. This allows ALM to collect arbitrary log data, rather than just logs that can readily be converted to text, and means that Assuria has developed parsers for a large number of diverse log formats.

Log formats vary widely, with some variations less useful (and usable) than others. This paper describes these variations and suggests preferred approaches for log format and content in new applications, then discusses some example log formats. The intended audience is developers of software that writes to logs and developers of log formats¹.

2 Format

The most obvious distinction is whether logs are plain text (such as syslog) or binary (e.g. Windows Event Logs and most Unix audit logs). Other areas of variation include:

- Byte order for binary logs and multibyte plain text logs.
- Character encoding, both for plain text logs and for strings embedded in binary logs.
- How events in the log are identified.
- How fields within events are identified.
- Whether events are structured or free-text.
- For structured logs, whether the structure is implicit or explicit.

We shall cover structure (events and fields) in more detail in Section 3.

2.1 Binary or Plain Text

Binary logs tend to be more compact, more flexible and more easily parsed. The compactness and flexibility stem from the use of all eight bits of each byte. From a parsing perspective, plain text logs tend to mix control and data, making parsing difficult and error-prone. This is covered in more detail below.

The principal advantage of plain text logs is direct human readability. However, human readability tends to make machine readability more difficult, while converting a machine-readable format into some human-readable form is usually straightforward. The volume of log data generated in current IT environments makes machine readability (e.g. for analysis and alerting) important.

Unless direct human readability is essential, binary logs with a suitable tool to convert to text are preferable. This is the model used by Windows since NT4 and by all Unix audit log formats supported by ALM.

¹Inventing new log formats is dubious: use well-designed existing formats where possible.

2.2 Character Encoding

Log formats should define both the character set and the character encoding used for strings therein. The obvious character set for new formats is Unicode. There are two obvious choices for encoding: UTF8 and UTF16LE. For predominantly Western content UTF8 should give smaller logs. Chinese, Japanese, Korean and Arabic tend to be smaller when encoded with UTF16.

Other character sets/encodings may be justifiable in particular circumstances. For examples, EBCDIC is a reasonable choice for logs written under IBM *i*.

3 Structure

3.1 Events and Fields

A log can be considered as a sequence of events, where each event consists of a number of fields. Each field has a name and a value. It should also have a type (giving the traditional type/length/value tuple) that defines how the value's bytes are interpreted. Plain-text logs often disregard strong typing, which makes parsing and other handling more difficult. Section 3.3 covers this in more detail.

Under this model the parsing problem reduces to identifying individual events in the log and then extracting fields from those events. In general terms, the field content is *data*; everything else is *control* (i.e. stuff that lets us get at the data).

3.2 Control and Data

The overriding design principle for a well designed log format is *separation of control and data*. In a broader context, buffer overflow, cross-site scripting and SQL injection vulnerabilities are direct consequences of a failure architecturally to separate control and data. The corresponding problem in log management is the log injection vulnerability².

Separating control and data rules out some common approaches, including:

- Any format involving escaping (i.e. embedding control in data, and identifying the control bytes using some magic character that then must be escaped when it occurs in the data).
- Any format with fields whose lengths are determined by some special termination character that ought not to appear³ in the log data itself. This includes delimited formats such as comma-separated values (CSV), and plain text formats that delimit records with newlines, such as syslog⁴.

Events and fields can each be considered as blobs (Big Lumps Of Binary) that should be interpreted in some format-specific manner. The first step is extracting each blob. Control in this case is the blob's start offset and the blob's length; the datum is the blob itself. Separating control and data mandates an explicit length field distinct from the blob: a <length, value> tuple.

²See http://www.owasp.org/index.php/Log_injection and <http://www.ossec.net/main/attacking-log-analysis-tools>. The second paper illustrates why parsing of textual logs intended for humans is problematic.

³The notion that the delimiter "can't" or "will not" appear in the log is an assumption that log injection attacks seek to exploit.

⁴Syslog doesn't prohibit newlines within events: you therefore have an ambiguous format (in formal grammar terms) and have to guess whether a line is a new event or a continuation of the previous event.

Type	Representation
Boolean	Single byte, 0x00 or 0xFF
Unsigned integers: 8, 16, 32 and 64 bytes	Binary, big-endian
UID, GID, PID (Unix)	As 32 bit unsigned integer
SID (Windows)	As Windows' binary representation
UUID	Two 64 bit unsigned integers
IPv4 address	As 32 bit unsigned integer
IPv6 address	128 bit blob
String	32 bit number of bytes, then characters in UTF8
Timestamp	Seconds since the epoch (midnight on 1 January 1970, UTC), as a 32 bit unsigned integer.

Table 1: Example type system

Windows Event Logs before Windows Vista (i.e. NT 4.0 through Server 2003) encode events in <length, value> format as discussed above. However, they extend this to <length, value, length>, which has the interesting property of supporting reading events backwards as well as forwards. The cost of this is an additional 32 bits per event (assuming 32 bit lengths, which are prudent).

3.3 Interpreting Bytes

After identifying the events in a log file, the next step is ascribing meaning to the bytes that each event consists of. For a human reading a plain-text log this is usually trivial. However, for a machine the format of the log has a significant bearing on the amount of information that can be extracted and on the robustness of the extraction (aka parsing) process.

The conventional approach to interpreting bytes, and one that is overlooked by most textual formats, is to use a type system. A type is a set of values; we also need a reversible representation of the values in the log format. Table 1 shows an example type system and representation.

3.3.1 Syntax and Semantics

The type system describes the format (i.e. syntax) of the bytes; it does not define their meaning (semantics). For example, source and destination addresses for a network connection are both (say) IPv4 addresses so have the same data type, but are semantically different. We thus have both semantic names and syntactic types, where the former usually dictates the latter. We shall henceforth refer to these as "field names" and "data types" respectively. For example: "The field with name 'source address' has data type 'IPv4 address'."

3.3.2 Implicit or Explicit Fields

An event consists of a number of fields, each of which has a field name and a data type. In many cases the different events that can be written to a log, the fields that are written for each event, and the order of those fields in the log file are known beforehand. If we can identify each event (for example with a numeric event ID) then we can omit fields' names and data types from the log file. The parser must then know, for example, that the data bytes for event 73 are the source IPv4 address (32 bits), the destination IPv4 address (32 bits), etc. The fields are *implicit*, rather than being specified explicitly in the log data.

The principal advantage of this is space: we're embedding knowledge in the parser rather than in the log, so the log tends to be smaller. There is one significant disadvantage: parsers can extract nothing of value from events defined after the parser was deployed, because the format is not self-describing. If the complete event does not include a length then the parser can't even skip unknown events: all subsequent events in the file are lost.

The alternative (explicit fields) gives each field and/or data type an ID and records these in the log before the data bytes. The format can record the field name, the data type or both. Field names can be enumerated and recorded as an ID, or stored as strings. Explicit formats tend to have simpler parsers that rarely need updating, and since the format is self-describing new fields and events can be added relatively easily. Adding new data types is less flexible though, because the means to interpret those remains in the parser.

3.4 Compression

Popular bulk compression algorithms tend to perform well on log data, with compressions ratios better than 50%. Long-established design principles such as layering and separation of concerns dictate that bulk compression should not be defined within the log format, but instead applied to log files as a whole.

However, it is tempting to exploit the redundancy in logs (which tend to contain lots of similar events) more directly. Two examples are Windows Vista event logs (also used in Windows Server 2008 and Windows 7), and HP-UX audit logs.

Events in Windows Vista event logs are fragments of XML, typically with a substantial amount of fixed structure and some small number of varying text elements or attributes. The log files encode this XML in a proprietary binary encoding that compresses events via templates: the first time that a particular event is written, its XML is defined as a template with substitution parameters. This template is then instantiated whenever similar events occur by writing just the template ID and the data to substitute into the template.

HP-UX audit logs contain system call events. Each event has some information related to the process that invoked the system call, such as effective and real user IDs, executable (program) name, process ID and parent process ID. Since a process makes a large number of system calls but this standard information rarely changes during a process' lifetime, HP-UX writes a "PID Record" whenever the standard information is established or changed, and thereafter just records the PID with each event. The parser refers to the most recent PID Record with matching PID to identify the relevant user and process IDs etc.

These approaches yield appreciable space savings at the expense of parser complexity. However, there is one other drawback: you can't interpret an event without first parsing every preceding event in the log file. Even if you know the offset of an event within the file, you can't just seek to that offset and read the event. If random access to log files is important and audit log files are large (tens of megabytes and upwards) then this imposes a significant performance penalty.

4 Content

The previous sections covered the format and structure of logs. This section discusses what information the logs should contain. The format and structure dictate how easily and reliably the logs can be interpreted by a computer; the content determines how useful the logs are to the user. Compliance standards tend to disregard format and structure but are more prescriptive with content.

4.1 Timestamps

Every event in the log should have a timestamp that records when the event occurred. If this differs significantly from when the event is written to the log then the write time should be recorded as well. However, few standard log formats record timestamps adequately. Timestamps should include, at a minimum:

- Year, month, day of month, hour, minute and second.
- An indication of whether the timestamp is local time or UTC.
- The offset of local time from UTC.

Some formats record finer-grained timestamps (milli- or microseconds); the absolute value of these is generally unreliable and of little use. The important capability is being able to order events with respect to time, for which a simple monotonically increasing sequence number suffices. However, a total order of the events in a log with respect to time is impossible if there are multiple CPU cores or events otherwise happening simultaneously. Whether a (partial) time ordering is well-defined and what events it should cover depend on the source of the events. For example, ordering HTTP requests from different clients to a clustered HTTP server within, say, a second is of little value, whereas ordering the system calls executed by a thread is essential to trace filesystem activity accurately.

Of greater importance than strict total ordering of events is the ability to compare timestamps in different logs and from different machines. This means that synchronising clocks in an IT system is essential.

A common error in plain text logs is to record timestamps in a human-readable form that makes parsing difficult or impossible. The following should be avoided:

- Locale-specific formats (e.g. `strftime %c, %D, %p, %P, %x, %X, %Z`).
- Non-numeric months or days-of-week (`strftime %a, %A, %b, %B, %h`).

If in doubt use ISO 8601 notation with a timezone offset.

4.2 Users and Groups

Users and groups in current operating systems usually have a human-readable name but are represented by some binary ID internally, such as UIDs, SIDs or UUIDs. The log should ideally contain both, because:

- Names can be ambiguous. For example, there are several versions of a Windows username (`domain\alice, alice@domain.local`).
- Names may or may not be case-sensitive.
- IDs avoid ambiguity but are meaningless to a human, so must be resolved by the log management system and/or viewing software.

Where space concerns dictate storing one or the other, prefer IDs, and leave conversion to names to the parser or display utility.

4.3 Sessions

Tracking user activity within a session is an increasingly common requirement. A "session" depends on the context, but may be an interactive session at a workstation, a financial transaction involving multiple events or a series of HTTP requests. However a session is defined, all events attributable to a session should include some form of session identifier so that sessions can be reconstructed.

Session identifiers should either be unique in time (i.e. never be reused within a machine) or should provide some suitably large time window before being eligible for reuse (i.e. the length of time before reuse should be significantly greater than the length of the majority of sessions). For example, Unix PIDs are inadequate for session tracking due to the potential frequency of reuse on a busy system.

4.4 Other Notable Fields

Events should include, where appropriate:

- An ID that identifies the event type (see Section 3.3.2).
- A success or failure indication. (Is "access denied" a *failure* to gain access, or a *successful* denial of an illegal access attempt?)
- For changes, relevant data values before and after the change.
- The ID/name of the user causing the event to be logged.
- The source of the event (e.g. program or subsystem).
- The operation being attempted.
- Whether the operation requires privileges above those granted to ordinary users of the system and, if so, the relevant privileges.
- Any relevant addressing information, such as IP addresses, email addresses, filenames or URLs.
- A reason for failure.

5 Recommendations

1. Don't reinvent the wheel. If a suitable standard mechanism exists then use it. However, evaluate it in the context of the above discussion, and don't equate popularity with quality. For example, syslog has few redeeming features, and does little to justify its ubiquity. If you define your own log format then:
2. Define a data type system, then fields in terms of those data types, then events in terms of those fields. Write a serialiser and a deserialiser for each data type. Expressing the events and fields in a machine-readable form (such as XML) and automatically generating the serialisation and deserialisation code for each event from that description (e.g. via XSLT) is a productive and robust implementation strategy.
3. Using large (32 bits) and random event and field IDs allows a degree of error recovery for corrupted log files.
4. Do not dump C structure values directly to log files: you will have problems with byte order and structure padding.
5. Refer to Section 4 and any relevant compliance standards when deciding what events and fields to include in the log.

6. Prefer binary formats to plain text: they're more compact, more robust and easier to parse.
7. Consider whether random access is important. If so, avoid relying on information cached from parsing previous events in the log.
8. Consider whether reading files backwards is important. If so then succeed as well as precede blobs with their lengths.

6 Examples

6.1 Syslog

Few formats (of logs or otherwise) have the ubiquity of syslog, and few formats have as large a disparity between popularity and quality:

```
Feb  2 07:36:06 localhost hp: unable to open /var/run/hpiod.port
Jan 29 11:01:34 wear auth|security:notice su: from root to test
at /dev/pts/1
Oct 30 05:13:00 1,2009/10/30
05:13:00,0004A100288,SYSTEM,general, 0,2009/10/30
05:13:00,,unknown,,0,0,general,informational, 192.168.0.12
Reading vadition\technical Membership: 11..
```

Observations:

1. It's plain text. Although there's notionally a standard, vendors apply it loosely if at all. Since the standard has many shortcomings, some attempts (e.g. rsyslog, syslog-ng) have been made to improve upon it, with varying degrees of success. This means that there is no single "syslog" format, but instead multiple incompatible formats that might all end up intermingled in the same log file because of the way that syslog messages are forwarded. We therefore have multiple "standards", all purporting to define a single format, in mutually incompatible ways, with no reliable (or even unreliable) means of disambiguating the result.
2. Timestamps: no year (!), no time zone, difficult to parse. Some variants have added the year, but this just means that there is no standard timestamp format, so we have to try a selection and see which works best on an event-by-event (not even log-by-log) basis.
3. Despite various standards' claims to the contrary, there are no consistent fields other than the ill-defined timestamp.
4. Although the syslog protocol (i.e. the transport mechanism) defines *facilities* (i.e. sources) and *priorities*, these are not recorded in the log.
5. The only application-defined field is a textual string. This has no standard format, as the above three examples illustrate.

6.2 Windows Event Log (Pre-Vista)

This is a reasonably well-designed binary format, albeit with one significant flaw. It is also not documented. Open documentation of audit log formats is important because audit logs can have long lifetimes and may need to be read on machines/systems that are unrelated to the source machine.

Log files consist of a header that facilitates fast access to the first and last events in the file. Events then consist of a fixed part, a variable data part, then a trailing copy of the record's length. The fixed part contains:

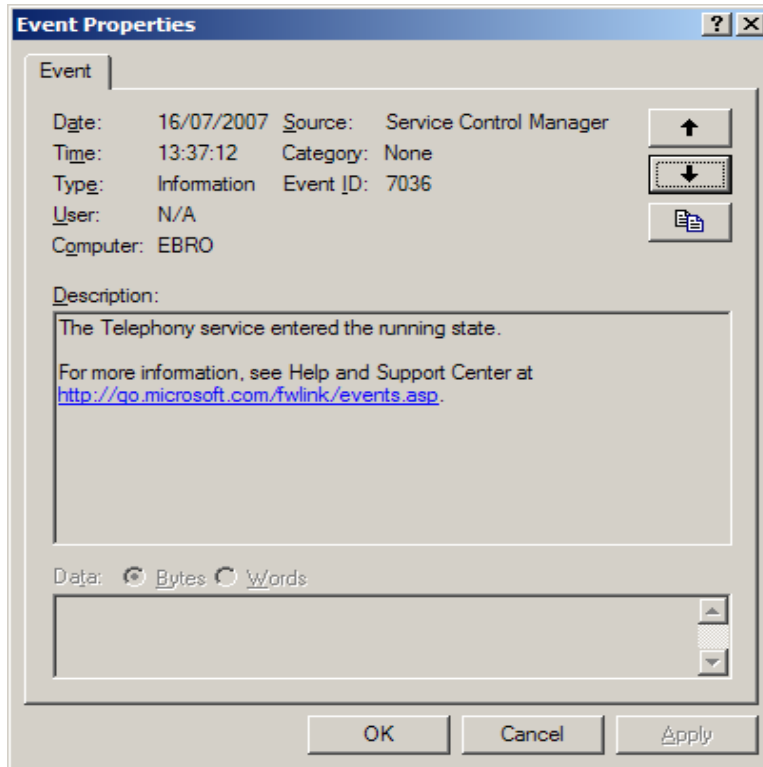


Figure 1: Windows (pre-Vista) Event Viewer

- “An infinite string of percent characters: %1.”

Windows XP originally imposed an arbitrary limit on the number of substitutions. A patch introduced around 2006 appears to detect such abuse and refuses to perform any substitutions at all. (It probably hits the arbitrary limit internally but then gives you the original string rather than that achieved on reaching the limit. A more robust approach would involve solving the Halting problem.)

We therefore have an audit log format that is not self-contained, produces messages that differ depending on the patch level of the machine *viewing* (not generating) the log, and contains an interesting avenue for denial-of-service attacks, but for the ability to include a simple message string.

6.3 Solaris 10 Basic Security Module Audit Log

Most Unix systems have a system call auditing facility that generates binary log files. The Solaris and HP-UX files (next section) are useful examples of explicit and implicit fields respectively.

Solaris audit log files are defined in terms of tokens. Leading and trailing file tokens give filenames and timestamps for audit trail continuity. Events then consist of a header token followed by an arbitrary number of other tokens. The header token contains:

- Token ID.
- Event byte count.
- Version.
- Event type.
- Nano-second granularity timestamp, though without time zone details.

The header token therefore defines an event as a <type, length>-encoded blob, where the blob is a sequence of tokens. There is a relatively rich set of tokens, describing IP addresses, processes, sockets, users, privileges etc. in terms of basic types such as 1, 2 and 4 byte integers and '\0'-terminated strings. For example, the "return" token, which describes the return value of a system call, contains:

- Token ID: 0x27 (32 bit platform) or 0x72 (64 bit platform).
- Error number (i.e. errno).
- Return value (4 or 8 bytes depending on 32/64 bit platform).

The parser must therefore know how to parse each token, but it does not need to know *a priori* which tokens are logged for a given event type. This is an example of explicit fields.

The format is well-documented, although the documentation contains the occasional error. One other drawback is that the format depends on whether the source machine has a 32 or 64 bit kernel, and on whether the CPU is big- or little-endian. Both of these can be deduced from the log file, but they aren't explicitly recorded therein.

6.4 HP-UX 11.11 and 11.23 Audit Logs

These are conceptually similar to Solaris BSM logs: they are binary logs that contain system call audit data. Records consist of a header and a data part. The header contains:

- Header length.
- Timestamp (second granularity).
- Process ID.
- Error number (errno).
- Event ID.
- Data part length.

The Event ID defines the format of the data part, using a type system. For example, event ID 16 represents the chown system call, and has the following fields:

- Return value: 32 bit integer.
- User ID: UID (32 bits).
- Group ID: GID (32 bits).
- Filename: a "Filepath" type, which gives inode and device numbers, plus the filename.

The key difference between the Solaris and HP-UX logs is that with HP-UX the parser must know which fields are present for each event, whereas in Solaris each field (i.e. token) in the event has an ID.

This lack of field IDs, plus the PID record compression discussed in Section 3.4, tend to make HP-UX logs more compact than Solaris logs. The trade-off is parser complexity and inflexibility in modifying existing events and in adding new events.

The HP-UX audit format appears to be documented. However, the documentation is inadequate for parser implementation, serving only to provide the illusion of an open, documented format.