

Common Event Record Concept

Representation of Events within the End-to-End Diagnostic Backplane

Introduction

With the advent of large scale systems using broadly-distributed commodity components it has become increasingly difficult to isolate specific symptoms of a problem let alone trace its root cause. A single web interaction by one user can spawn many backend transactions among multiple hosts that offer specific services such as authentication, authorization, and special services including DNS, LDAP, etc. In turn the process repeats itself recursively for each of the backend hosts until the primary transaction is satisfied. To complicate matters further, these transactions increasingly traverse multiple networks and administrative domains that link the backend transaction hosts that are managed by multiple organizations over vast distances.

State of the Practice

The challenges of diagnosing such a complex system are not solely limited to inadequacies of existing tools or the supporting technologies, surprisingly it stems mostly from the fact that the practice of distributed systems diagnostics and the management of the diagnostic data has not evolved sufficiently from the single system to the distributed system domain.

The tools that the diagnostician has available are many and can be classified in five dimensions,

1. Highly specialized tools that are focused on a specific domain: application, network, system, security or performance
2. General forensic tools, but do not span the application, network, system, security or performance domains
3. Reporting and notification tools
4. Passive vs. active, i.e. using data generated from external sources or actively probing the distributed system, injecting a test and looking for results
5. Real-time vs. historical

Most of these tools consider only one specific type of diagnostic. The developers of many of these tools acknowledge that their tools could be significantly enhanced through easy access and correlation with other sources of diagnostic data. We envision a single API to provide backend collection, normalization and secure data distribution and access services. Further, such a service would help them to develop tools more efficiently without the need to create this backend facility. This capability could foster creative approaches to diagnostic problems by giving different groups of developers access to the same data.

Most current diagnostic forensic work is still a manual process, reactively approaching the problem using multiple tools and scanning log files with scripting tools or a text editor. The type, form, and quality of diagnostic data varies greatly, from multiple variants of application and system event records generated on many types of operating systems, network flow records, or

specialized security event records. The data itself could be in the form of files, streamed data (NetFlow¹ or Syslog²) or stored within a database.

Even with the vast amount of diagnostic data available, the distributed system diagnostician faces four fundamental roadblocks to the diagnostic process,

- Cannot get access to the data because of host access policy
- Correlating different data types is difficult
- Cannot find meaningful information within the vast amount of data quickly
- Sharing diagnostic data with others to help to solve problems collaboratively is a problem because of different formats or policy issues

Assuming that the diagnostician overcame these roadblocks, providing rich diagnostic data that correlates distributed system events to a specific user may be further complicated by privacy policies of an organization. It is becoming increasingly difficult to safely include userID information within the diagnostic data to aid in the troubleshooting process.

Even if adequate diagnostic data anonymization strategies are deployed, there are multiples ways to overcome even the best implementations. Also with rich sources diagnostic data one can easily infer the actions and behaviors of a specific system, organization or an individual. The diagnostician has an added responsibility to shepherd this data based on the policy of their organization. Diagnostic data lifecycle policies and practices must be developed to securely transport, distribute, anonymize, manage, reduce and remove the diagnostic data. The risks can be great with respect to non-compliance legal issues. It may be prudent to do a one way transform or remove some or all of the diagnostic data after some period of time when it falls outside of the window of usefulness.

Requirements for a New Approach

The method for discovering a new approach to the problem was based on conversations from diagnosticians and diagnostic tool developers from the application, network, system, security, and performance domains.

- **Application** – Events produced by applications processes that do not pertain to the operation of the host or the OS. Examples are Shibboleth, BIND, SNMPD, HTTPD, LDAP, custom portal and user applications, etc.
- **System** – Events produced by the OS that pertain to itself or the underlying hardware. Examples are faults and information from the OS kernel, RAID subsystems or other hardware interfaces.
- **Network** – Events that happen on the network. Examples are NetFlow data, SNMP traps or deltas, RMON, etc.
- **Security** – Events that are generated by a system/application whose focus is security. Examples are IDS, firewalls, Snort, etc.

¹ NetFlow is a trademark of Cisco Inc. an dis a flow based network audit record generated by many devices that are mostly specific to Cisco.

² Syslog is a Unix based logging facility.

- **Analysis** - Events generated from the analysis of application, system, network or security events that reach a specific threshold defined by the application and merit reporting. Examples are events from analysis from OWAMP or IPerf data, high level IDS, etc.

In summary, the following is a high level description of the requirements for representing diagnostic events in a way that begins to address the problems and challenges outlined earlier,

- **Reducing barriers to adoption**
 - Enable the use of existing file based and stream based events without requiring the diagnostician to make any changes to their existing event infrastructure
 - Provide a way for the developer to change log formats without reducing the correlation benefits
- **Providing a consistent representation for diagnostic data**
 - Provide a unique global identifier for each event
 - Preserve the original diagnostic data, i.e. be able to reproduce exactly what entered the system
 - The ability to indicate if the data was modified
 - Create a rich event description and representation that can vary in granularity based on policy
 - Retain flexibility in data representation to allow for a variety of formats to accommodate performance or specialized parsing needs.
 - Use a common XML based data representation
- **Providing a rich and flexible method for correlation of events**
 - Enable the diagnostician to define and tag events to significantly enhance event correlation capabilities.
- **Evolution of the CER**
 - The ability to modify and improve the CER and be backward compatible with preexisting versions

Implementation

The following is a high level representation of the data structure format of the CER. We will not go into the formal representation because this paper is intended to give an overview to give the reader a broader understanding of our approach. The following outlines version 0.6 of the specification.

CER Event Types

The CER can be represented by three major types, raw, parsed and analyzed events.

Raw: Events that represent log events from the distributed system that are native and have no analysis process associated with them. The data from this event exists in its initial data form within the CER. Examples are a record from the UNIX /var/log/maillog file or a Netflow event. The CER will preserve the raw data form but has the ability to indicate that it has been transformed. An example of a transformation is a specific field being removed because of privacy policy.

Parsed: These events are derived from raw diagnostic events. They are decompiled and tagged with information that describes specific meaning of the original data fields of the event and are represented in XML.

Analyzed: Events that were produced from a system that observed one or more raw, parsed or some other out-of-band data, and produced an analysis of some point in time. An example of this is a high level IDS system that reads first order application, network and application events and determines that there is a denial of service attack being directed at some part of the infrastructure

CER Components and Data Structures

The following describe the components of each of the three types of CER. Both the raw and parsed CER types represent specific events in the application, network, system or security domains. Appendix A illustrates the types and structures that make up the CER.

Base Descriptor - The base descriptor is a data structure that includes common information among all types of CERs. The data includes,

- **Version (required)** – version number of the base component data structure.
- **CER Type (required)** - Raw, Parsed or Analyzed

Event Descriptor - includes data that describes the fundamental aspects of classifying an event. The internal data representation of a base component includes,

- **Version (required)** – version number of the base component data structure
- **TimeStart (required)** – time that the event was observed
- **TimeStop (optional)** – time that the event has finished. Note most events are recorded from log files and no stop time is recorded.
- **TimeAccuracy (optional)** – accuracy description of the clock of the observed time of the event
- **ServerHostName (optional)** – node name where the events are being created and observed.
- **ServerAddress (required)** - address where the events are being created and observed
- **ServerAddressType (required)** - IPV4, IPV6, or MAC.
- **CollectorHostname (optional)** – node name where the events are being collected
- **CollectorAddress (required)** – address where the events are being collected
- **CollectorAddressType (required)** - IPV4, IPV6, or MAC
- **CollectorName (required)** – name of the normalization agent that collects the events
- **CollectorVersion (required)** – version number of the normalization agent that collects the events
- **Service (STRING/required)** - service that created the event. Examples are, OS name, IMAP, IDS, Snort, SQL, NetReg, NetMon, HTTP, etc.
- **ImplementerName (required)** - software name and version that produced the event in its raw format. Examples are, IIS:4.0, Apache:3.1, BIND:9.1, etc. Note that this name ties closely with the backplane normalizer that created it.

- **ImplementerVersion (required)** - version of the implementation that produced the event. Note that this version ties closely with the backplane normalizer that created it.
- **User Tag (optional)** - name value pares defined by the user. Examples are, app:astronomy, app:portal, etc.
- **Event Type (required)** - application, system, network or security.
- **WarnLevel (required)** – metric that describes the severity and type of event.
 - **EMERG** – system is in an unusable state
 - **ALERT** – action must be taken immediately to prevent the system to be unusable within a short period of time
 - **CRIT** – critical conditions, system is operating but is in danger of eventually being unusable
 - **ERR** – error conditions, system is operating but is reporting errors
 - **WARNING** – warning conditions
 - **NOTICE** – normal but significant condition, major changes in state
 - **INFO** – informational, notices of changes in configurations or state
 - **DEBUG** – debug level events
- **Raw Event Encoding Mechanism (required)** – Binary, ASCII, ASN.1, or XML.
- **Raw Event (required)** – raw event message collected

Parsed CERs - The parsed CER is made up four additional sub-event type data structures in addition to the base descriptor where are called elements. They represent the four classes of non-analyzed events, process, system, network, and security. By combining these elements in the following way we can define the four major types of events in great detail.

Application Event = Base Descriptor + Process Element
System Event = Base Descriptor + Process Element + System Element
Network Event = Base Descriptor + Network Element
Security Event = Base Descriptor + Network Element + Security Element

Note that each parsed CER has been generated by a normalization agent which decomposed the detail for each specific type of event. There is a facility for a generic representation of events in the application, network, system and security domains to allow rapid incorporation of events into the CER structure and to server as a template for more detailed highly focused CERs.

Parsed CER Data Structure Definitions (ELEMENT structure)

- **ProcessElement** – defines the process that determined and generated the event. This is a physical running process on a host or a network device such as a switch, router or security device such as a firewall or IDS. The internal data representation of a process component includes,
 - **Version (required)** - version number of the process element structure
 - **ProcessID (required)** – physical process ID defined by the OS on the device that generated the event

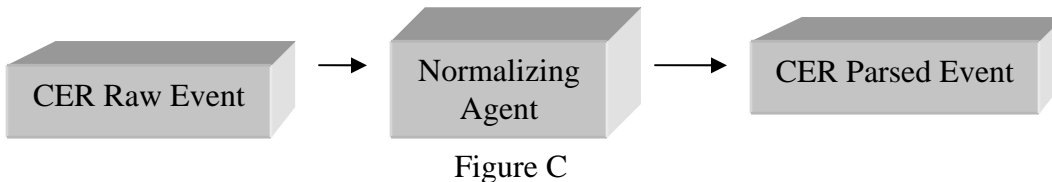
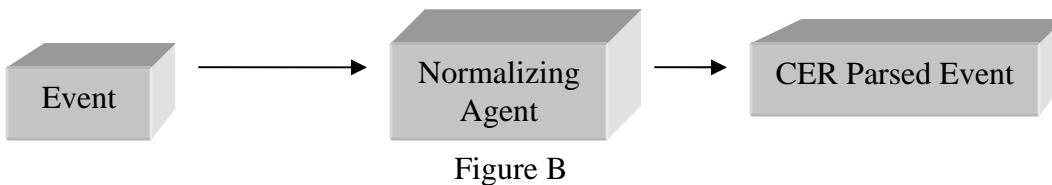
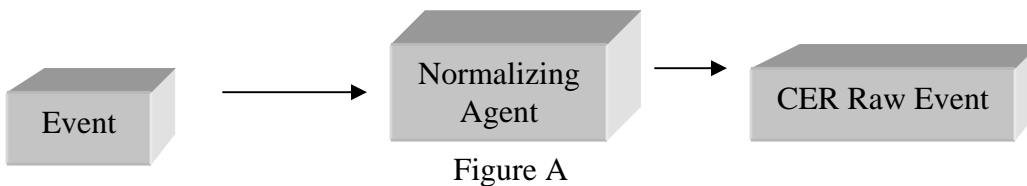
- **ProcessName (required)** – physical name of the process defined by the OS on the device that generated the event. Examples of names are processes such as Syslog on Unix based OSs.
- **ProcessOwner (required)** – physical owner name defined by the OS on the device that generated
- **SystemComponent (optional)** – system component is only included if the event is of type system.
- **SystemElement (Generic)** – defines the low level facility that determines the fundamental events of an OS and the marriage with the underlying hardware
 - **Version (required)** - version number of structure
 - **Facility (required)** – component of the OS that generates the event. Example: kernel.
 - **SubSystem (optional)** - lower level sub-system that created the event. Example: raid controller.
- **NetworkElement (Generic)** – defines the event that occurs on a network.
 - **Version (required)** - version number of network element data structure
 - **AddressType(required)** - IPV4, IPV6, MAC, etc.
 - **SourceIP (required)** – address of the source of the flow
 - **DestinationIP (required)** – address of the destination of the glow
 - **PacketsSource (required)** – total source packets in the flow
 - **PacketsDestination (required)** – total destination packets in the flow
 - **OctetsSource (INT/required)** – total source octets in the flow
 - **OctetsDestination (required)** - total destination octets in the flow
 - **ErrorsSource (INT/required)** – total source errors in the flow
 - **ErrorsDestination (required)** - total destination errors in the flow
 - **SourcePort (required)** – source port of a TCP or UDP flow
 - **DestinationPort (required)** – destination port of a TCP or UDP flow
 - **Protocol (required)** – Protocols such as IP, TCP, ICMP, etc.
 - **TCPOptions (optional)** – TCP options that are set
 - **TCPState (optional)** - state of the TCP flow.
 - **Payload (optional)** - data portion of the flow.
- **SecurityElement (Generic)** – defines the event that occurs within the security domain.
 - **Version (required)** - version number of the security element data structure.
 - **AddressType (required)** - IPV4, IPV6, MAC, etc.
 - **PolicyViolation (optional)** – a description of the policy exception that prompted the generation of this event.

Analysis Event Data Structure Definitions (Generic) - defines the event that occurs **from an al**

- **Analysis Descriptor** - describes the process that analyzed the event(s) that produced an analysis event.
 - **Version (required)** - version number of the analysis data structure.
 - **AnalysisSubsystemName (required)** - component of the analysis system that made the assertions
 - **EventID(s) (required)** - CERs that were used to produce the analysis.
- **Analysis Assertion** - multiple analysis assertions and exist
 - **Name (required)** - name tag of the
 - **Assertion Encoding Mechanism (required)** – Binary, ASCII, ASN.1, or XML.
 - **Assertion (required)** - Content of the analysis.

Normalization of Events

The process of transforming an event from the application, network, system, security or analysis domains is done by a normalizing agent. The sole purpose of the normalization agent is to accurately convert the event to the CER format. The normalization agent has the option to convert the event directly to a CER raw event or to add a high level of detail represented by a parsed event. The following diagram illustrates the process of conversion. Note that in figure C a normalization agent can also convert a raw CER event to a parsed CER.



Conclusion

Prior to writing this document, experimentation on this concept was done through a pilot project named ccBay sponsored by The National Science Foundation through a contract with UCAID/Internet2. The goals of that project were to test the effectiveness of an early version of the CER and the event dissemination backplane. Upon conclusion of this pilot, some of the advantages of this approach included:

Carnegie Mellon University
Draft

- Enabling the creative and automated correlation of events from different areas (network, application, security and system) in a simple and effective manner
- Gives flexibility for representing specialized meaning that is specific to the environment where it is deployed.
- Leverage metrics about the severity of an event based on Syslog(3) conventions
- OS and platform independence by writing normalization and transfer agents for the Linux, Fedora and Windows platforms

We also found that our 0.5 version of the CER (note that this document describes version 0.6) had the following limitations that were addressed in this document.

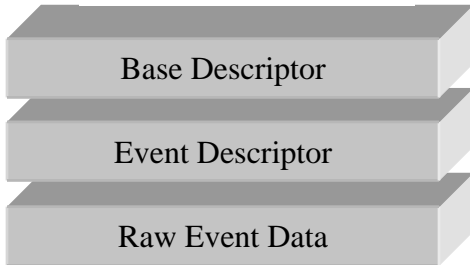
- Severity metrics not always standard across hosts
 - Addressed by allowing different versions of parsed event records
 - Addressed by allowing analysis of events through the analysis CER
- Address scaling issues and consider other data representation formats.
 - Addressed by designing the raw CER
- Need to include second order events from the Measurement/Performance domains
 - Addressed by allowing analysis of events through the analysis CER
- Event message must be parsed to provide a more granular representation of an event to aid in correlation
 - Addressed by allowing different versions of parsed event records
- Overhead of XML may outweigh the advantages with respect to event record size and the time it takes to parse
 - Addressed by designing the raw CER

The next phase of this effort is to continue from the groundwork of the pilot and implement a design and product that can be used by the general community. Please note that the CER format is still not finalized, and any comments to its design and direction are highly welcome. More information about the effort and the CER can be found at the Internet2 site, <http://middleware.internet2.edu/e2ed/>.

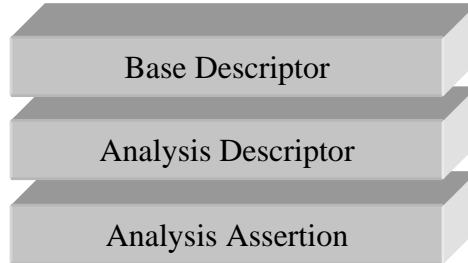
Appendix A

Graphic Representation of CER Types

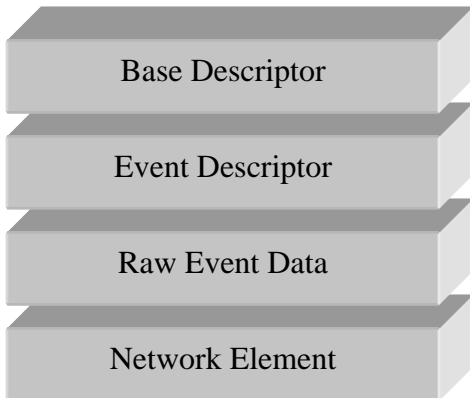
Raw CER



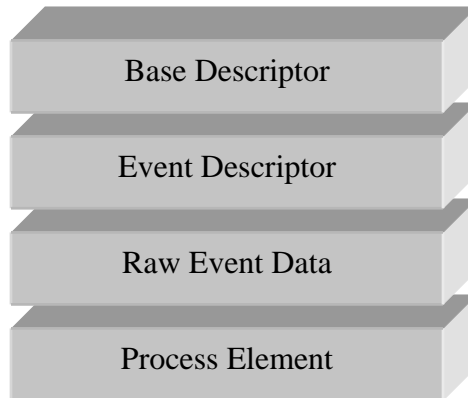
Analysis CER



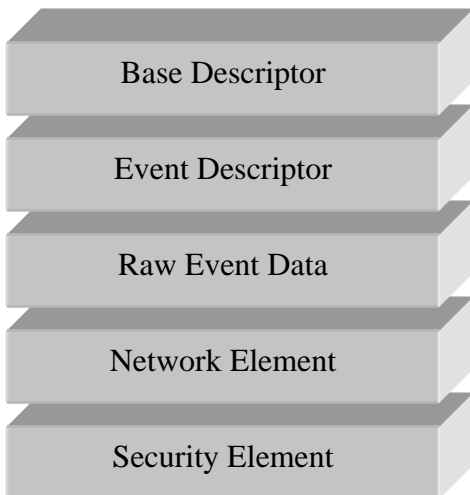
Parsed CER (Network)



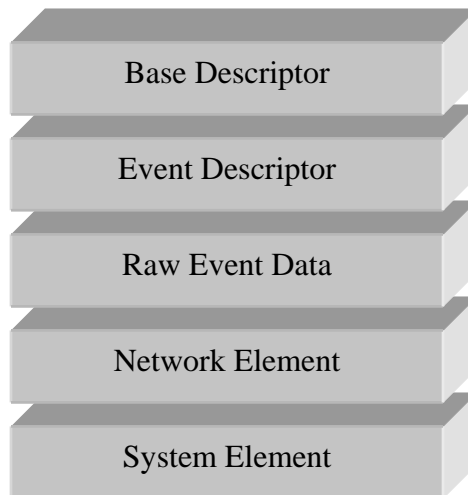
Parsed CER (Application)



Parsed CER (Security)



Parsed CER (System)



Appendix B

Examples of Specific Parsed CER Types

Network (NetFlow Version 5)

- **Version (required)** - version number of network element data structure
- **SourceIP (required)** – IPV4 address of the source of the flow
- **DestinationIP (required)** – IPV4 address of the destination of the flow
- **NextHopIP (optional)** – IPV4 address of the next hop in the flow if known
- **InterfaceIn (optional)** – flow input interface if known
- **InterfaceOut (optional)** – flow output interface if known
- **Packets (required)** – total packets in the flow
- **Octets (required)** – total octets in the flow
- **SourcePort (optional)** – source port of a TCP or UDP flow
- **DestinationPort (optional)** – destination port of a TCP or UDP flow
- **IPTypeOfService (optional)** – type of IP service
- **Protocol (required)** - IP Protocol number (UDP, TCP, etc.)
- **TypeOfService (required)** – type of service
- **SourceAutonomousSystem (optional)** – source autonomous system number
- **DestinationAutonomousSystem (optional)** - destination autonomous system number
- **SourceMask (optional)** – IPv4 netmask of source flow address
- **DestinationMask (optional)** – IPv4 netmask of destination flow address

Security (Snort)

- **Name (required)** - Name of Structure. Examples are generic, Snort, Tripwire, etc.
- **Version (required)** - version number of structure
- **ReferenceSystem (required)** – system that produced the security event (Example: Snort)
- **TCPHeader (optional)** – TCP header of IP packet generating security event
- **UDPHeader (optional)** – UDP header of IP packet generating security event
- **ICMPHeader (optional)** – ICMP header of IP packet generating security event
- **RawData (optional)** - event data