THE JOURNAL OF DIGITAL FORENSICS, SECURITY AND LAW

Journal of Digital Forensics, Security and Law

Volume 11 | Number 2 Article 3

2016

Log Analysis Using Temporal Logic and Reconstruction Approach: Web Server Case

Murat Gunestas
Security Department Gumushane

Zeki Bilgin
Security Department Gumushane

Recommended Citation

Gunestas, Murat and Bilgin, Zeki (2016) "Log Analysis Using Temporal Logic and Reconstruction Approach: Web Server Case," *Journal of Digital Forensics, Security and Law*: Vol. 11 : No. 2 , Article 3.
DOI: https://doi.org/10.15394/jdfsl.2016.1377
Available at: https://commons.erau.edu/jdfsl/vol11/iss2/3

LOG ANALYSIS USING TEMPORAL LOGIC AND RECONSTRUCTION APPROACH: WEB SERVER CASE

Murat Gunestas, Zeki Bilgin
Security Department
Gumushane, 29000, Turkey
mgunestas@egm.gov.tr, zbilgin@egm.gov.tr

ABSTRACT

We present a post-mortem log analysis method based on Temporal Logic (TL), Event Processing Language (EPL), and reconstruction approach. After showing that the proposed method could be adapted to any misuse event or attack, we specifically investigate the case of web server misuses. To this end, we examine five different misuses on WordPress web servers, and generate corresponding log files of these attacks for forensic analysis. Then we establish attack patterns and formalize them by means of a special case of temporal logic, i.e. many sorted first order metric temporal logic (MSFOMTL). Later on, we implement these attack patterns in the EPL, and performed experimental log analysis by using a time window mechanism sliding on sorted log records to evaluate effectiveness and efficacy of our proposed method. We found that our approach is potentially capable of providing a platform where investigators can define/store/share misuse patterns using a common language while providing fast and accurate forensic analysis on large log files.

Keywords: log analysis, digital investigation, network forensics, web application security, intrusion detection system, complex event processing

1. INTRODUCTION

Log files are journals of computer systems and applications, and stores valuable information for system administrators, security professionals, and digital investigators. Log analysis is a critical tool for digital investigators, as well as for cyber security professionals and system administrators. Once arrived on the cybercrime scene, law enforcement officers investigating cybercrimes need to understand what happened on the system, when did it happen, and who did it. For all those forensics procedures, there is need to collect related evidence from log files. Those evidences should soundly be linked to the misuse or crime. This is, however, not an easy task because of enormous sizes of web log files, complexity of understanding and correlating misuse patterns linked to actual cybercrime such as intrusion.

As typical log files are huge in size, it is difficult to crawl over lines along with filtering, sorting and other mining capabilities. Correlation of events that are distant to each other in large log files take so much effort in computation and manpower.

In this context, this research presents a log analysis approach using temporal logic and reconstruction approach for fast and better log analysis, which is also potentially capable of

providing a platform where investigators can define/store/share misuse patterns using a common language. Because Temporal Logic (TL) is typically used for detecting time-based complex patterns over streams in real time, we aimed at taking advantage of TL after reconstructing records in log files. We have selected Apache web server logs as a case study since Web servers are one of the most preferred targets for hackers and other cyber criminals to intrude systems because of their publicity. Web logs are set of timely recorded events occurred between web servers and clients. In general, Web log files keep each record in the form of request and response together in one line. Reconstructing web server activities as streams based on records in web log files gives us the capability of implementing TL based on streaming data. This way it becomes trivial to achieve fast and better forensics investigation because there are state-of-the-art streaming technologies such as StreamBase, Esper, etc. ("EsperTech - Esper," n.d., "StreamBase | Complex Event Processing, Event Stream Processing, StreamBase Streaming Platform," n.d.). We preferred Esper platform and Event Processing Language (EPL) as a standard language to define misuse patterns. Esper provides .NET and Java packages that are easy to implement either for a standalone application or enterprise framework along with EPL, a declarative language for dealing with high frequency time-based event data ("EsperTech - Esper," n.d.).

To the best of our knowledge, there is no platform performing post-mortem log analysis using MSFOMTL, EPL, and reconstruction approach. In addition, cyber security professionals and investigators lack a standard format or language to store and share their previous experiences on log analysis. Besides performance advantage and temporal logical capabilities, our approach would base a platform and a library to store, share, and adjust previously identified patterns of misuses for further analysis.

Through the paper, we describe previous work in Section 2, along with some background information in Section 3. Section 4 describes misuse patterns informally that could reside and could be detected from web log records. Then, in Section 5, we define formal versions of those patterns using a special case of TL. Section 6 describes EPL queries that are mapped from TL formulae given in the previous section.

2. LITERATURE REVIEW

Many researchers have studied log analysis from several aspects. We can categorize these efforts in three groups; (1) focusing on analysis of very large log files, (2) addressing vast variety of log formats, and (3) correlation of events through log entries.

In the first group, Vernekar and Buchade (2013) propose a system and claim to provide significant improvement in response time through large log file analysis, correlation of events and generating alerts by implementing MapReduce algorithm.

The Iterative Partitioning Log Mining (IPLoM) approach (Makanju, Zincir-Heywood, & Milios, 2009) divides log files into clusters to mine the appropriate patterns for further alert generation based on these patterns. The approach employs three steps through its hierarchical partitioning, followed by generation of cluster description for each clusters produced. Through another research to deal with large log file analysis problem, Kalamatianos et al. (2012) propose a technique that helps analysts to focus on a smaller collection of events related to their analysis objective rather than the entire log file. Havens et al. (2012) focus on Bayesian spam filters through categorization of log entries claiming to identify log entries only relevant to known

context, and to effectively exclude outage relationships. Similar to those studies, addressing fast and effective analysis of large log files our approach based on temporal logic brings a solution to the same problem by narrowing down the investigation scope using time windows.

Among the second group, Jayathilake (2011) mentions that many log file formats are supported by existing log analysis tools and proposes a framework that can handle many log file formats along with a new language called "Log Data Extraction Language" (LDEL). Jayathilake claims the language is capable of expressing all log file formats known after analyzing 20 different log file types, and comes with a flexible parser. Arasteh et al. (2007) propose a model checking approach addressing the problem of formal analysis of logs. They model the log as a tree labeled through a term algebra that represents the variety of actions logged.

Falling into the third group, Security Information and Event Management (SIEM) systems, today, collect information from several security software and hardware, network devices, systems, and applications. Correlating event data with other conceptual information in real time, several actions can be taken by SIEM products, such as event and activity monitoring and reporting. Although there are several commercial tools, Splunk, LogRythm, and Archsight are most widely used. There are few open source SIEM products, and OSSIM is the most popular. Like many SIEM products, OSSIM also provides its own engine to perform rule-based event correlation; thus allowing users to define dependencies between events through an xml file rather than implementing temporal logic, any event processing language, or existing any CEP engine. MASSIF, on the other hand, is a SIEM example, which is based on Complex Event Processing (CEP). MASSIF can automatically translate OSSIM's directives into CEP queries and can run queries in parallel (Kavanagh & Rochford, 2015; Jimenez-Peris, 2015).

Complex Event Processing is a technique used for processing events with the purpose of identification of complex event patterns in real time. To some extent, complex event processing systems employ temporal logic through their processes. Esper ("EsperTech - Esper," n.d.) and Streambase ("StreamBase," n.d.) are the examples of most commonly used CEP frameworks today. Since event processing languages ("EventFlow and StreamSQL," n.d.; Albek, Bax, Billock, Chandy, & Swett, 2005) provide adequate implementation of temporal logic; today, some IDS proposals prefer CEP frameworks in charge. Ahmed et al. (2011) implement MSFOMTL to formally define misuse patterns; transform these patterns into StreamSQL query language; and run those queries on Streambase platform. MONID and ORCHIDS are other examples of IDS frameworks implementing Temporal Logic (Ahmed et al., 2011). Above efforts implement CEP for real time analysis and do not intend to address post mortem investigations. Automated Forensic Diagnosis System (J. Herrerías & R. Gómez, 2010), on the other hand, proposes reconstructing attacks after incidents and carries out log analysis using Event Correlation Module. This way, the system is said to detect multi-step attacks and to reduce false positives.

3. BACKGROUND

Three important issues we need to know about log files are: the vast variety of log formats, categorization of data in log files, and enrichment of the content of a log entity.

3.1 Log Formats

Today we can categorize log files in three classes: Structured, Semi-structured, and

Unstructured. Structured log records are typically defined by a schema that tells you what kind of records to expect (e.g. Oracle or MySQL database). Semi-structured logs are typically provided with some descriptive meta-data (e.g. XML), which is a type of data that partially describes the original data. In XML, that data is the element and attribute names. Unstructured log data, on the other hand, can be server logs, audio and video streams, paragraphs of text, bit streams of all kinds with no inherent meta-data, semantic, or structural design. Prior to processing, this type of log data needs to be parsed to some extent.

3.2 Categorization

As there are several types of events, many applications and logging software today enforce categorization by severity. *NIX systems provide panic, critical, error, warning, info, debug categories, while Windows provides Startup and Shutdown, Authentication Success, Authentication Failure, Access Allow, Access Deny, Audit Success, Audit Failure, Audit Other, Critical, Error, Warning, and Information categories. Facilities related to log records are also used in categorization, such as Kernel, email.

3.3 Enrichment

Enrichment of log records contributes to log analysis twofold: In the first, it provides investigators better information compared to meaningless abbreviations or numbers; secondly, investigators could employ enrichment functions in correlations, filters, and other predicates through their analysis. Some examples are given below;

- IP address can be transformed into several other information that is of investigators' interest: country information, blacklisted or whitelisted, internal or external, Proxy, VPN, or real endpoint, etc.
- Port numbers can be transformed into well-known services, blacklisted or whitelisted, etc.
- Size (Bytes) can be transformed into a scaling classification, small, medium, large.

Analysts perform enrichment through built-in or user-defined functions using internal or external sources or services. Given raw data, corresponding values can be used in correlation and definition of events.

4. MISUSE PATTERNS

In order to empower investigators with a fast, effective log analysis, we need to make sure that they have an adequate knowledge base in the context that they will perform analysis, that is, web server misuses in our case study. It is important to know exactly what to look for, or at least, some signatures or patterns representing the evidence in advance. If there is already a pattern successfully abstracted and corresponding to activity that you are after, then it is relatively easy to mine the pattern. However, this is not valid for unknown or novel misuses; thus, there is need to develop patterns for such malicious behaviors or events. Since patterns can be complex and span over time as a series of events, we have slightly changed/extended the systematic signature engineering approach of Schmerl et al. (2008) to develop new patterns corresponding to specific activities:

1. Execute the activity (benign or malicious) on the similar system to record its traces. Traces represent events observable through log records.
2. The resulting traces contain all events of the system, but try to find a small/unique fraction that could represent the activity.
3. Manually inspect log records in order to identify the activity specific events.

4. Derive the new pattern gradually, considering how the activity proceeded.
5. After specifying the pattern, its correctness and conciseness must be validated. If necessary, the pattern has to be corrected or modified.
6. Abstract from attack specific patterns to develop new patterns addressing other unknown attacks featuring same attitude.

5. MISUSE PATTERNS IN TEMPORAL LOGIC

In order to represent misuse patterns that include many types and span over time we prefer Many Sorted First Order Temporal Logic MSFOMTL. Below we describe how we can define misuses mentioned earlier using MSFOMTL.

5.1 Overview of MSFOMTL Syntax

Temporal logic, which is a special type of modal logic, is widely implemented to verify the correctness of computer systems as it provides a framework for modeling systems and a specification language for describing the properties to be verified. It brings a great advantage especially when analysing *safety-critical* systems as well as *commercially-critical* and *mission-critical* systems, and for this reason, there is a growing demand for formal verification methods from industry and research community (Huth & Ryan, 2004).

In temporal logic, the models include several states, and a formula can be true in some states and false in others, as opposed to propositional and predicate logic. Thus, the notion of truth reflects a dynamic structure as the formulae change their truth-values based on the state of the system (Huth & Ryan, 2004).

In this study, we adopt to use Many Sorted First Order Metric Temporal Logic (MSFOMTL), which is a special case of temporal logic, in order to model certain attack patterns for analyzing log records spanning over time. We select MSFOMTL because it has some special features that can be efficiently used to unambiguously and concisely describe the events to be investigated.

MSFOMTL is previously proposed and used by Ahmed et al. (2011) to develop an online network intrusion detection system based on temporal logic, and stream data processing. MSFOMTL has the following syntax given in Backus Naur form:

$$\varphi := \top | \bot | p | \neg\varphi | \varphi \wedge \varphi | (\varphi \vee \varphi) | \rightarrow \varphi |$$

$$(\forall x)\varphi | (\exists x)\varphi \mid \Diamond_{[t1,t2]}\varphi \mid \Box_{[t1,t2]}\varphi |$$

$$\blacklozenge_{[t1,t2]}\varphi | \blacksquare_{[t1,t2]}\varphi | R^{n}_{[t1,t2]}\varphi$$

where p is any propositional atom from some et Atoms, and the symbols $\top$ and $\bot$ are MSFOMTL formulas. The $\Diamond$, $\Box$, $\blacklozenge$, and $\blacksquare$ are *temporal connectives*. $\Diamond_{[t1,t2]}$ means "eventually," $\Box_{[t1,t2]}$ means "always," $\blacklozenge_{[t1,t2]}$ means "sometimes in the past," and $\blacksquare_{[t1,t2]}$ means "always in the past" between the moments t1 and t2 from now.

5.2 Developing Patterns for Apache Logs using MSFOTML

To illustrate this, we follow the same study case- that is- web server logs.

$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_1)$ where:

- x_1 : is a string variable representing the client IP address;
- x_2: is a string variable representing Host;
- x_3: is a string variable representing the User;

- x_4 : is an date-time variable representing the Timestamp;
- x_5: is a string variable representing the Request Method;
- x_6: is a string variable representing the Request Url;
- x_7: is an integer variable representing the Response Code;
- x_8: is an integer variable representing the Bytes Sent;
- x_9: is a string variable representing the Referrer;
- x_1 : is a string variable representing the User Agent;

We classify 4 pattern types: Single, Multiple, Compound, and Abstract patterns.

5.3 Single Record Patterns

Given a single log record, we can deduce a specific activity then we are assumed to define a pattern representing this activity based on a single record. To define such a pattern, we typically employ keyword or regular expression matches and/or other basic logical operators over field values of a log entry. Below are some examples of Single Record Patterns.

SQL injection attack using GET method is a form of web application attack that targets the database behind front-end web application interface. Through the input forms of a web page, an attacker would typically input a value, which is not a regular and expected by the system. For example, the attacker would submit a string like **`%' or '0'='0`** instead of typical number value, **`1`**, that can be used as an ID number for an entity stored in a table in the backend database. This way, the SQL sentence created for the endpoint database would be **`SELECT first_name, last_name FROM users WHERE user_id = '%' or '0'='0'`** instead of **`SELECT first_name, last_name FROM users WHERE user_id = '1'`**, thus leading the attacker to retrieve entire table rather than a single record. Such an attack could be identified through a single record pattern depicted below:

$$P(x_1, x_2, x_3, x_4, "GET", x_6, 200, x_8, x_9, x_1) \wedge c_I \quad \begin{matrix}(\exists x_1, x_2, x_3, x_4, x_8, x_9, x_1) \\ (x_6, "\%\ or\ \%=\%")\end{matrix} \qquad I$$

SQL injection attack using POST method is similar to the above attack, however, launched through post method, if the web site is designed to work on post methods. In this case, *URL* field in log entries does not provide any clue regarding what is requested from the web server. Therefore, we need to develop another pattern to detect the malicious activity. An additional field may help us through this, that is, *bytes* field if typical post requests for a specific page returns a fixed size response or the response is limited in a range, then we can define a threshold for normal usage and detect excessive responses by comparing bytes field to this threshold.

$$P(x_1, x_2, x_3, x_4, "POST", x_6, x_7, x_8, x_9, x_1) \wedge c_{II} \quad \begin{matrix}(\exists x_1, x_2, x_3, x_4, x_7, x_9, x_1) \\ (x_6, /query.php) \wedge x_8 > 4600\end{matrix} \qquad II$$

Today web servers are widely built upon ready-made themes such as WordPress, Joomla, Drupal, etc. Web activities represent same traces on log records when they are built upon the same theme. Therefore, once we define patterns specific to those themes, then we can search for same patterns on other websites built with the same theme and detect malicious/suspicious activities. As a case study, we have selected WordPress to define patterns and below are some examples as WordPress has the 38% of the market ("CMS technologies Web Usage Statistics," n.d.).

Figure ***Error! No sequence specified.****.*

WordPress failed login is an activity such that a web user attempts to log in a web site developed based on WordPress theme and cannot succeed. WordPress theme's login page, wp-login.php, has been designed to accept credentials through POST methods of the Hypertext Transport Protocol (HTTP). The theme responds to client with a code 200, which corresponds to OK in HTTP level; however, code 200 means the page returns successfully with a response message reading the failure during the login. Red rectangle in Figure 1 shows a failed login trace in an Apache Log file while Pattern III maps to the evidence of this failed login attempt.

$$P(x_1, x_2, x_3, x_4, \text{“}P\quad\text{”}, x_6, 200, x_8, x_9, x_1) \wedge c(x_6, \text{”}wp\text{-}login\text{”}) \quad (\exists x_1, x_2, x_3, x_4, x_8, x_9, x_1) \quad III$$

WordPress successful login is an activity such that a web user successfully logs in a web site developed based on WordPress. Unlike unsuccessful logins, WordPress responds to client with a code, 302, which corresponds to REDIRECTION of the page from wp-login.php. Green rectangle in Figure 1 points to a successful login trace in an Apache Log file while Pattern IV maps to the evidence of this activity.

$$P(x_1, x_2, x_3, \text{“}P\quad\text{”}, x_6, 302, x_8, x_9, x_1) \wedge c(x_6, \text{”}wp\text{-}login\text{”}) \quad (\exists x_1, x_2, x_3, x_4, x_8, x_9, x_1) \quad IV$$

5.4 Multiple Record Patterns

We define multiple records patterns when we need to represent complex events comprise more than one event spanned over time and related to each other. However, we define multiple records patterns not only that are dependent on each other, but also for corroborating evidence of complex pattern by including more single record patterns. This mitigates false positives when a single record pattern cannot feature a distinguished signature of the activity, and through the search, matches more activities than the expected one. It is often to process multiple records either backward or forward depending upon the standing point.

Forward Multiple Records Patterns: When we need complex patterns comprising different single record patterns in ascending order in time, that is, expected patterns are defined before they appear, we employ Forward Multiple Records Patterns. While defining such patterns, MSFOMTL utilizes interval values to define time differences between events, as interval values could be treated as relative values to actual timestamp values, which are definitely to differ case by case. Pattern V corresponds to a WordPress Advanced Video Plugin Local File Inclusion attack against a WordPress-based web site. Exploit Database ("WordPress Advanced Video Plugin 1.0–- Local File Inclusion LFI," n.d.) provides the exploit in detail for this attack along with instructions, exploit, and vulnerable plugin source code.

Figure 2. WordPress Advanced Video Plugin Local File Inclusion Exploit traces in an Apache Log

The exploit completes the attack in 3 phases; however, it leaves 4 records in the log file as shown in Figure 2. In the first, admin-ajax.php has been requested with action, title, short, and thumb parameters. To gain access local files in web server, attacker misuses the Thumb parameter and assigns the intended sensitive file, rather than expected image file. The webserver loads the content of the intended file instead of actual image file and stores it in a folder. Exploit tries to locate and download this file. This is, however, not straightforward; thus the link in the content of the response to the first request is requested as a second step. Since this link is the short link rather than the actual URL of the post, Server redirects the client to the actual URL. As a third step, the actual URL is requested and the post page which includes the link to the thumbnail image file is returned. Finally, this image file is requested. If all of the steps above are completed with success, then this corresponds to an evidence of Successful WordPress Advanced Video Plugin Local File Inclusion attack.

$$(\exists x_1)\left(\begin{array}{c}\left(\begin{array}{c}(\exists k_2,k_3,k_4,k_8,k_9,k_1)\\ P(x_1,k_2,k_3,k_4,"GET",k_6,200,k_8,k_9,k_1)\\ \wedge\, co\qquad (k_6,"a\qquad -a\quad .php?a\qquad =a\quad \_p\qquad hP\quad ")\end{array}\right)\\ \wedge\Diamond_{[0,1]}\left(\begin{array}{c}(\exists l_2,l_3,l_4,l_8,l_9,l_1)\\ P(x_1,l_2,l_3,l_4,"GET",l_6,301,l_8,l_9,l_1)\\ \wedge\, c\qquad (l_6,"/?p=")\end{array}\right)\\ \wedge\Diamond_{[0,1]}\left(\begin{array}{c}(\exists m_2,m_3,m_4,m_8,m_9,m_1)\\ P(x_1,m_2,m_3,m_4,"GET",m_6,200,m_8,m_9,m_1)\\ \wedge\, c\quad in\ (m_6,s\qquad\qquad (k_6,"t\quad =","\&"))\end{array}\right)\\ \wedge\Diamond_{[0,1]}\left(\begin{array}{c}(\exists n_2,n_3,n_4,n_8,n_9,n_1)\\ P(x_1,n_2,n_3,n_4,"GET",n_6,200,n_8,n_9,n_1)\\ \wedge\, c\qquad (n_6,".j\quad ")\end{array}\right)\end{array}\right) \quad V$$

Backward Multiple Records Patterns: When we need to identify patterns where a specific event is required not to be preceded by other event/s in the past; we employ Backward Multiple Records Patterns. As in Forward Multiple Records patterns, MSFOTML employs interval values to define time differences between events. For example, successful administrative access events generally entail to be preceded by successful login events. If we cannot find successful logins before successful administrative activities, then we could consider this a suspicious activity, such as session hijack, IP change (e.g. proxy-drop) in the middle of a running session, or extremely long session time-outs which is a vital security vulnerability for a web server. In Pattern VI, we define administrative access to a web server based on WordPress not preceded by a successful login in a reasonable past, which is 48 hours (172800 seconds) default

session timeout value for WordPress based web sites (“WP Login Timeout Settings — WordPress Plugins,” n.d.).

Repeating Multiple Records Patterns: When we need complex patterns in which same single record pattern repeats several times, typically,

$$(\exists x_1)\left(\begin{array}{c}\left(\begin{array}{c}(\exists y_2, y_3, y_4, y_8, y_1\ )\\ P(x_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, \text{"-"}, y_1\ )\\ \wedge\, c\qquad (y_6, \text{"wp-admin"})\\ \wedge\left(\begin{array}{c}(y_5 = \text{"GET"} \wedge f(y_7, 200))\\ \vee\,(y_5 = \text{"POST"} \wedge f(y_7, 302))\end{array}\right)\end{array}\right)\\ \wedge\neg\blacksquare_{[0,1\ \ ]}\left(\begin{array}{c}(\exists z_2, z_3, z_4, z_6, z_8, z_9, z_1\ )\\ P(x_1, z_2, z_3, z_4, \text{"POST"}, z_6, 302, z_8, z_9, z_1\ )\\ \wedge\, c\qquad (z_6, \text{"wp-login"})\end{array}\right)\end{array}\right) \quad VI$$

more than a threshold value within a time window; then we employ Repeating Multiple Records Patterns. The threshold value and time window size are assumed to be specified by the investigator based on three aspects: innards of the activity that pattern would claim to address, accuracy rates of pattern matching, and resource consumption. We prefer Repeating Multiple Records Pattern in order to address malicious activities that entail repeating pattern, such as vulnerability scanning or other brute force attacks.

*Figure 3.*Brute Force Login Attack Traces in Apache Log

Pattern VII is a Repeating Multiple Records Pattern for a sample brute force login attack against WordPress’ login page, which outputs when unsuccessful login attempts (as depicted in Pattern III) occur more than or equal to 30 times in 30 seconds by the same IP address:

$$(\exists x_1) R^{3}_{[0,3\ \ ]}\left(\begin{array}{c}(\exists y_2, y_3, y_4, y_6, y_8, y_9, y_1\ )\\ P(x_1, y_2, y_3, y_4, \text{"POST"}, y_6, 200, y_8, y_9, y_1\ )\\ \wedge\, c\qquad (y_6, \text{"wp-login"})\end{array}\right) \quad VII$$

5.5 Compound Patterns

As mentioned earlier, when single record patterns are not adequate to address an activity precisely we would take the advantage of Multiple Record Patterns including other single records related to the activity in a specific order or repetition. This is also possible to develop more complex patterns comprising several Multiple Record Patterns and Single Record Patterns together. This helps us to corroborate evidence of an alleged activity occurred in the system. We develop compound patterns in the event that particular patterns are weak to distinguish expected activity and raise vast amounts of false positives. In such cases, we look for other patterns taking part in addressing the same activity followed or preceded by a weak pattern. As an example, Pattern VIII comprises of brute force login attack followed by a successful login as this can be used to locate records shown in Figure

3. The first part locates the records in green while the second locates the record in red.

$$(\exists x_1)\left(\begin{array}{c} R^{30}_{[0,3\ ]}\left(\begin{array}{c}(\exists y_2,y_3,y_4,y_6,y_8,y_9,y_1\ )\\ P(x_1,y_2,y_3,y_4,\ \textit{“POST”},y_6,200,y_8,y_9,y_1\ )\\ \wedge c_{\ }\qquad\qquad (y_6,\textit{”wp-login”})\end{array}\right)\\ \wedge\Diamond_{[0,1]}\left(\begin{array}{c}(\exists z_2,z_3,z_4,z_6,z_8,z_9,z_1\ )\\ P(x_1,z_2,z_3,z_4,\ \textit{“POST”},z_6,302,z_8,z_9,z_1\ )\\ \wedge c_{\ }\qquad\qquad (z_6,\textit{”wp-login”})\end{array}\right)\end{array}\right)\quad \textit{VIII}$$

For versions before 2.0.2., WordPress has a vulnerability that could be exploited by (cache) Remote Shell Injection Exploit (“WordPress <= 2.0.2 – cache Remote Shell Injection Exploit,” n.d.). Through the exploit, attackers could run arbitrary commands during profile updates. Those commands are appended into files in wp-content/cache/userlogins/ and wp-content/cache/users/ directories that are further included by cache.php through “cache_file = $group_dir.md5($id.DB_PASSWORD)..php;” code, thus allowing attacker to execute commands in his/her favor. One of the challenges in this attack is to find names of the files in wp-content/cache/userlogins/ and wp-content/cache/users/ directories without traversing, as web masters generally ban this. As md5 hash of the user password is used as the file name, Remote Shell Injection Exploit identifies filenames by calculating md5 hashes of possible passwords. Calculated hash values submitted to web server in the form of a URL through GET requests. Until received a positive response, 200,

Figure 4. (cache) Remote Shell Injection Exploit Traces in an Apache Log

$$(\exists x_1)\left(\begin{array}{c} R^{3}_{[0,2]}\left(\begin{array}{c}(\exists y_2,y_3,y_4,y_6,y_8,y_9,y_1\ )\\ P(x_1,y_2,y_3,y_4,\ \textit{“GET”},y_6,404,y_8,y_9,y_1\ )\\ \wedge c_{\ }\qquad\qquad (y_6,\textit{”wp-content/cache”})\end{array}\right)\\ \wedge\Diamond_{[0,1]}\left(\begin{array}{c}(\exists z_2,z_3,z_4,z_6,z_8,z_9,z_1\ )\\ P(x_1,z_2,z_3,z_4,\ \textit{“GET”},z_6,200,z_8,z_9,z_1\ )\\ \wedge c_{\ }\qquad\qquad (z_6,\textit{”wp-content/cache”})\end{array}\right)\end{array}\right)\quad \textit{IX}$$

as HTTP response, attacker receives several negative responses, 404, in a small time range. We have employed this attack and Figure 4 depicts the traces of this attack, while Pattern IX is an example of Compound Pattern for such an attack.

5.6 Abstract Patterns

After exhaustive examination on traces of log records, we observed that some compound patterns have commonalities, thus, could be abstracted from. After abstraction from similar corroborative patterns, we can obtain more

general patterns that can address more misuse activity types rather than being stuck to a specific activity. Patterns VIII and IX address similar activities. Therefore, we can abstract another pattern from these two patterns using their commonalities in order to address similar misuse activities in one pattern. Besides brevity, this will mitigate the number of searches over log records. As you can see below, the abstract pattern in Pattern X identifies events when excessive (=>30 times in 2 seconds) events with same ip, method, and response are followed by an individual event within at most 1 second with only one difference, that is, the response of the server.

$$(\exists x_1)\left(\begin{array}{c}(\exists k_5,k_7)R^3_{[0,2]}\begin{pmatrix}(\exists y_2,y_3,y_4,y_6,y_8,y_9,y_1\ )\\P(x_1,y_2,y_3,y_4,k_5,y_6,k_7,y_8,y_9,y_1\ )\end{pmatrix}\\ \wedge\Diamond_{[0,1]}\begin{pmatrix}(\exists z_2,z_3,z_4,z_6,z_8,z_9,z_1\ )\\P(x_1,z_2,z_3,z_4,k_5,z_6,z_7,z_8,z_9,z_1\ )\\ \wedge(z_7!=k_7)\wedge(z_7!=404)\end{pmatrix}\end{array}\right) \quad X$$

6. MISUSE PATTERNS IN EVENT PROCESSING LANGUAGE

Although we are able to define our developed patterns using temporal logic, there is need to implement temporal logic in order to analyze patterns. We prefer Event Processing Language (EPL) to define patterns and Esper Complex Event Processing (CEP) Engine to query these patterns.

6.1 Overview of Event Processing Language

Event Processing Language (EPL) has syntax similar to Standard Query Language SQL with additional properties that allow running queries over time windows. Below are brief descriptions of primary clauses commonly used through EPL.

SELECT: Given input streams SELECT statement allows a user to retrieve specific fields or expressions (e.g. using built-in or user-defined functions) based on fields among all fields through the input stream tuples.

FROM: FROM statement allows users to define from which input streams tuples are selected.

WHERE: Given logical operators a user could define filters over input streams using WHERE clause. Like SELECT statement through WHERE clause, users could employ built-in and user-defined functions for logical operators used through predicates.

PATTERN: Unlike traditional SQL, EPL provides PATTERN clauses to define forward multiple records patterns.

OUTER LEFT JOIN: EPL introduces OUTER LEFT JOIN clause in order to query patterns that are based on backward multiple records and looking for absence of some events in a specified past.

win: This statement added to the input stream to define the size of windows over streams.

6.2 Simple EPL Queries

Simple queries are designed to filter log records one at a time over only one input stream, thus consuming small memory space. We select

fields or additional expressions or user-defined functions over those fields that have key roles in introducing the activity to the user in SELECT clause. For single record pattern queries, we typically have one input stream. We design filters through WHERE clauses where we can employ logical operators along with built-in or user-defined functions. Below we provide EPL queries designed to query single record patterns that are depicted in patterns I – IV.

SQLi-GET is an EPL query depicted in Query 1, which defines the SQL injection attack using GET method defined in Pattern I. The query receives Apache log records in line 2, filter out records of which *response* value is equal to 200, *method* is equal to "GET" and *URL* value contains the signature, "% or %=%," that most SQL injection attacks feature in line 3. In line 1, we output the filtered records as an intended activity along with a "SQLi" phrase that address the type of activity, timestamp, IP address, Country of the IP address, URL containing the signature and agent used through the attack.

```
   @NAME('Output::SQLi-GET')
1  SELECT "SQLi-GET" as activity, timestamp, ip,
   countryFromIP(ip) as country, url, agent
2  FROM ApacheLogRecordStream
3  WHERE response = 200 and method = 'GET' and url
   like "% or %=%";
```

Query 1. SQL injection attack using GET method

SQLi-POST is an EPL query depicted in Query 2, which defines the SQL injection attack using POST method defined in Pattern II. The query receives Apache log records in line 2, filter out records of which *response* value is equal to 200, *method* is equal to "POST," *URL* value contains location of the dynamic query page (**$querypage**) which has potential for SQLi attack in line 3. Since Apache log records do not store content through POST methods, it is worthless to look for attack signatures in URL field. As mentioned earlier, depending upon the query page and its behavior, we can define a maximum size value (**$threshold**) for responses. In line 1, we output the filtered records as an intended activity along with a "SQLi" phrase that address the type of activity, timestamp, IP address, Country of the IP address, URL containing the signature and agent used through the attack.

SQLi-POST is an EPL query depicted in Query 2, which defines the SQL injection attack using POST method defined in Pattern II. The query receives Apache log records in line 2, filter out records of which *response* value is equal to 200, *method* is equal to "POST," *URL* value contains location of the dynamic query page (**$querypage**) which has potential for SQLi attack in line 3. Since Apache log records do not store content through POST methods, it is worthless to look for attack signatures in URL field. As mentioned earlier, depending upon the query page and its behavior, we can define a maximum size value (**$threshold**) for responses. In line 1, we output the filtered records as an intended activity along with a "SQLi" phrase that address the type of activity, timestamp, IP address, Country of the IP address, URL containing the signature and agent used through the attack.

```
   @NAME('Output:: SQLi-POST'')
1  SELECT "SQLi-GET" as activity, timestamp, ip,
   countryFromIP(ip) as country, url, agent
2  FROM ApacheLogRecordStream
3  WHERE response = 200 and method = 'POST' and url
   like "%$querypage%" and bytes > $threshold ;
```

Query 2. SQL injection attack using POST method

WPFailedLogin is an EPL query authored to query Unsuccessful login attempts through web sites developed on WordPress theme and defined in Pattern III above. The query receives Apache log records in line 2, filter out records of which *response* value is equal to 200, *method* is equal to "POST" and *URL* value

contains the login signature, "%wp-login%," that feature login actions performed through WordPress theme in line 3. In line 1, we output the filtered records as an intended activity along with a "WPFailedLogin" phrase that address the type of activity, timestamp, IP address, Country of the IP address, URL containing the signature and agent used through the attack.

```
   @NAME('Output::WPFailedLogin')
1  SELECT "WPFailedLogin"' as activity, timestamp, ip,
   countryFromIP(ip) as country, url, agent
2  FROM ApacheLogRecordStream
3  WHERE response = 200 and method = 'POST' and url
   like '%wp-login%';
```

Query 3. WordPress Failed Login

WPSuccessfulLogin is an EPL query authored to query Unsuccessful login attempts through web sites developed on WordPress theme and defined in Pattern IV above. The query receives Apache log records in line 2, filter out records of which *response* value is equal to 302, *method* is equal to "POST" and *URL* value contains the login signature, "%wp-login%," that features login actions performed through the WordPress theme in line 3. In line 1, we output the filtered records as an intended activity along with a "WPSuccessfulLogin" phrase that address the type of activity, timestamp, IP address, country of the IP address, and URL containing the signature and agent used through the attack.

```
   @NAME('Output::WPSuccessfulLogin')
1  SELECT "WPUnsuccessfulLogin"' as activity, timestamp,
   ip, countryFromIP(ip) as country, url, agent
2  FROM ApacheLogRecordStream
3  WHERE response = 302 and method = 'POST' and url
   like '%wp-login%';
```

Query 4. WordPress Successful Login

Investigators could combine single record pattern queries.

6.3 EPL Query for Multiple Records Pattern

Unlike EPL queries for single record patterns, EPL queries for multiple records are a little more complex. They generally include sliding time windows rather than processing over very large log files and enormous number of log records that would probably lead to overhead in memory and unnecessary computation over the records that are irrelevant from temporal perspective. Below is the EPL query of Pattern V depicted above.

Forward Multiple Records EPL Queries: When we need to query Forward Multiple Records Patterns described in Section 5, we employ Forward Multiple Records EPL Queries. While defining such patterns, EPL utilizes PATTERN clause to define different events that are expected to emerge consecutively. We use "->" symbol to define the order of the events. Further events can employ predicates using the values in previously matched events. While events are processed based on real time through typical event processing model, there are cases through which specific timestamp fields in stream tuples could be defined externally, thus events are processed based on this timestamp values rather than actual time. Predicates employing time criteria use time interval values between events. Query 5 maps to a successful WordPress Advanced Video Plugin Local File Inclusion exploit attack mentioned earlier on a WordPress-based web site. Line 1 refers to adequate descriptive information for investigators through SELECT statement. Instead of typical input stream through FROM statement, Line 2 defines a multiple forward records pattern including consecutive traces that comprise of an evident WordPress Advanced Video Plugin Local File Inclusion exploit attack, which is provided in Exploit Database along with instructions, source, and vulnerable code. We have launched the exploit on our test website and successfully retrieved the content of sensitive files at the server side.

The exploit code in python leaves 4 records in a web server log file as depicted in Figure 3 and Pattern V: Line 3 points to the request of admin-ajax.php along with **`action = ave_publishPost`**, **`title = $randomTitle`**, **`term = $random`**, **`short = $random`**, and **`thumb = $targetedSensitiveFile`** parameters; Line 4 corresponds to the second request using the URL including **`"/?p=%"`** phrase in response page for the first request; Line 5 is the trace of the third request which is held after web server redirects the client to actual URL that includes the value of title parameter in the first record. Finally, Line 6 refers to the last event of Pattern V, that is, the file request to jpeg file including intended local file on the web server. As we have seen in Figure 3, all requests are performed using GET method and response codes are expected to be 200 for a successful attack. Expected time intervals between events are defined to be equal or less than 1 second for this query.

```
   @Name('Output::WPAVEPluginExploit')
1  SELECT "WPAVEPluginExploit" as source, a.ip, a.url,
   a.timestamp as starttime , b.url, b.timestamp as endtime
2  FROM PATTERN [every
3          a=ApacheLogRecordStream(url like '%/wp-
   admin/admin-ajax.php%action=ave_publishPost%' and
   method='GET' and response=200)->
4          b=ApacheLogRecordStream(ip = a.ip and url like
   '%/?p=%' and method='GET' and response=301 and
   (timestamp.toMillisec() - a.timestamp.toMillisec()) <=
5  1000)->
           c=ApacheLogRecordStream(ip = b.ip and url like
   '%'||substringBetween(a.url,'title=','&')||'%' and
   method='GET' and response=200 and
   (timestamp.toMillisec() - b.timestamp.toMillisec()) <= 1000)
6  ->
           d=ApacheLogRecordStream(ip = c.ip and url like
   '%.jpeg' and method='GET' and response=200 and
   (timestamp.toMillisec() - c.timestamp.toMillisec()) <= 1000)
   ];
```

Query 5. WordPress Advanced Video Plugin Local File Inclusion

Backward Multiple Records EPL Queries: When we need to query Backward Multiple Records Patterns, we first define and filter specific events that will constitute the compound pattern. Since we need to search for absence of an event as well as existence of events, we need to define the time range as a scope of our search. In order to find the evidence of absence of some records we employ LEFT OUTER JOIN statement along with a window as a scope in which to search. Such windows are helpful twofold: (1) they lead process engines to limit their search space thus being memory efficient; (2) the pattern itself might be bound with specific time ranges as time-out values play critical role through identification of session based activities. For example, if login time out value is 60 seconds for a web site, then it is unnecessary to perform searches over time windows larger than 60 seconds. If we are not bound with any time limit, and deal with a reasonable amount of records that will not put any burden on memory, then we may not want to draw a scope; thus performing full-log search.

Query 6 corresponds to Pattern VI and search for instances of *WordPress Suspicious Administrative Access Without Login* pattern. Line 1 accepts the log file as an input stream and based on predicates in Line 4 and 7 input stream is filtered and split into two separate event streams. In Line 10, IP addresses of administrative activities are selected and looked up among successful login events in Line 11 and 12; when there is no match, then the pattern is assumed to match in Line 13.


```
   @NAME('Split:: WPAdminAndLoginActivity')
1  ON ApacheLogRecordStream
2      INSERT INTO WPLoginSuccessfulStream
3      SELECT timestamp, ip, url, agent
4      WHERE response = 302 and method = 'POST'
       and url like '%wp-login%'
5      INSERT INTO WPAdminSuccessfulStream
6      SELECT timestamp, ip, url,method, response,
       referer, agent
7      WHERE url like "%wp-admin%" and
       ((method="GET" and response=200) or
       (method="POST" and response=302))
8  OUTPUT ALL;

   @NAME('Output::
   WPSuspiciousAdminActivityWithoutLogin')
9  SELECT "admtime:", adminStream.timestamp,
   "admin:" || adminStream.ip, "admurl:" ||
   adminStream.url, adminStream.agent
10 FROM
   WPAdminSuccessfulStream().win:ext_timed(timest
   amp.toMillisec(), 1 sec) as adminStream
11 LEFT OUTER JOIN
   WPLoginSuccessfulStream().win:keepall() as
   loginStream
12 on adminStream.ip=loginStream.ip
13 WHERE loginStream.ip is null and
   adminStream.referer="-";
```

Query 6. WordPress Suspicious Administrative Access Without Login

Repeating Multiple Records EPL Queries: When we need to query a Repeating Multiple Records pattern, we need to define the size of time window and the threshold value as an identifier where the counts of repeats are above it. Repeating patterns are critical to identify misuses at *scanning* and *gaining access* levels of intrusion attacks as well as *Denial of Service* (DoS) attacks. Unlike intrusion attacks, DoS attacks are designed to consume resources at the target site and they typically employ repetitive actions with specially crafted payloads and/or through vulnerable spots. The difference between intrusion and DoS attacks is that in the former, repeating pattern is sometimes followed by a successful gain, in the latter, however, attackers aim no unauthorized access at all. Through developing EPL queries for repeating patterns, the threshold value and time window size are assumed to be specified by the investigator based on three aspects: innards of the activity that pattern would claim to address, accuracy rates of pattern matching, and resource consumption.

WPBruteForceLoginAttempt is an EPL Query where Apache Log file is accepted as an input stream within a **`$WindowSize`** variable in seconds basing timestamp values in tuples in Line 2 in Query 7. Since we are looking for repetitions of specific events we may need to group tuples by related fields. Line 3 allows us to count the events that have same IP addresses. Since for web sites with huge amount of users, unintended login failures of real users that are not part of any attack may lead this pattern to produce false positives; Pattern VII is designed to identify this malicious behavior based on IP addresses. In Line 4, the variable **`$RepeatThreshold`** allows investigators to focus only on IP addresses which preform excessive amount of login attempts and fail in a small time range that are not expected through normal behavior of any client.

```
   @Name('Output::WPBruteForceLoginAttempt')
1  SELECT ip, min(timestamp) as starttime,
   max(timestamp) as endtime, count(*) as cnt
2  FROM ApacheLogRecordStream (response = 200 and
   method = 'POST' and url like '%wp-
   login%').win:ext_timed(timestamp.toMillisec(),
   $WindowSize sec)
3  GROUP BY ip
4  HAVING count(*) > $RepeatThreshold;
```

Query 7. Repeating WordPress Failed Logins

6.4 EPL Queries for Compound Patterns

Scanning phase in cyber-attacks is generally a stepping-stone to gain unauthorized access at the victim site. Almost none of the countries in the world consider scanning, itself, a crime, unless any unauthorized access or damage occurs at the target. This makes scanning a most widely used tool among hackers, penetration testers, researchers, etc. Thus, for many web sites it is often used to observe huge

amounts of scan activities among their log records. This makes the investigators work harder and leads them to narrow their investigations down only to scans followed by successful unauthorized access or damage at the target. This can be achieved developing EPL queries that can search compound patterns that comprise other patterns, such as single record or multiple records patterns.

WPSuccessfulBruteForceLoginAttack defined in Query 8 below, extends the EPL query in Query 7 such that repeating failed logins are followed by a successful login defined in Query 4. As defined in Pattern VIII, in Lines 1-6 the query filters repeating login attempts more than 30 times in 30-second windows while in Lines 7-9 it filters successful login records. Finally, Lines 10-11 correspond to a compound pattern through which both events are consecutively defined, that is, repeating failed logins immediately (1 second in this case) followed by successful logins by the same IP address.

```
    @Name('Split:: WPBruteForceLoginAttempt')
1   INSERT INTO RWULAStream
2   SELECT ip, min(timestamp) as starttii
    max(timestamp) as endtime, count(*) as cnt
3   FROM ApacheLogRecordStream(response = 200 a
    method = 'POST' and url like '%v
4   login%').win:ext_timed(timestamp.toMillisec(), 30 sec)
5   GROUP BY ip, response
6   HAVING count(*) > 30;

    @Name('Split:: WPSuccessfulLogin')
7   INSERT INTO WSLStream
8   SELECT max(timestamp) as timestamp,
    min(timestamp) as starttime, count(*) as cnt
9   FROM ApacheLogRecordStream(response = 302 a
    method = 'POST' and url like '%wp-login%');

    @Name('Output::
10  WPSuccessfulBruteForceLoginAttack')
    SELECT ": Successful Wordpress Login Scan Att
11  from " || a.ip as source, a.starttime, a.endtime, b.timesta
    as endtime
    FROM PATTERN [every
            a = RWULAStream() ->
            b = WSLStream(ip = a.ip and
    (timestamp.toMillisec() - a.endtime.toMillisec()) <= 1000)];
```

Query 8. Wordpress Successful Brute Force Login Attack

WPCacheScanAttack similar to Query 8 and is used to identify in terms of traces left in the log file. However, as described in Section 5, (cache) Remote Shell Injection Exploit has a scanning component targeting caching mechanism of WordPress to gain access first and then to inject malicious code. We define only to identify the successful scanning part here as we want to illustrate how scanning activities have similar behavior. As defined in Pattern IX, through Lines 1-4 the query filters repeating unsuccessful cache requests more than 30 times in 2-second windows, while through Lines 5-8, it filters successful cache request records. Finally, Lines 9-10 correspond to a compound pattern through which both events are consecutively defined, that is, repeating failed cache requests immediately (1 second in this case) followed by successful cache request by the same IP address.

```
    @Name('Split::WPCacheScans')
1   INSERT INTO WPUnsuccessfulCacheStream
2   SELECT ip, min(timestamp) as starttime,
    max(timestamp) as endtime, count(*) as cnt
3   FROM ApacheLogRecordStream (response = 404 and url
    like '%wp-
    content/cache%').win:ext_timed(timestamp.toMillisec(),
    2 sec)
4   GROUP BY ip, response having count(*) > 30;

    @Name('Split::WPCacheSuccess')
5   INSERT INTO WPSuccessfulCacheStream
6   SELECT timestamp, ip, url, agent
7   FROM ApacheLogRecordStream
8   WHERE response = 200 and method = 'GET' and url
    like '%wp-content/cache%';

    @Name('Output::WPCacheScanAttack')
9   SELECT a.ip||": Successful Attack" as source,
    a.starttime , a.endtime as _endtime, b.timestamp as
10  endtime
    FROM PATTERN [every
            a=WPUnsuccessfulCacheStream()->
            b=WPSuccessfulCacheStream(ip=a.ip and
    (timestamp.toMillisec() - a.endtime.toMillisec()) <=
    1000)];
```

Query 9. Scanning Component of (cache) Remote Shell Injection Exploit

6.5 EPL Queries for Abstract Patterns

As mentioned earlier, some compound patterns are similar, and so are EPL queries corresponding to them. EPL queries, designed to address abstract patterns, can contribute to Log Analysis twofold; (1) detecting other unknown misuses that are similar to each other and (2) addressing several types of patterns querying only one pattern. However, since the pattern is in more generalized form, it may produce more false positives than patterns dedicated to a specific misuse type.

WPSuspiciousScan is an EPL query to search for Pattern X. Unlike Queries 8 and 9, Query 10 employs no predicates on the input stream rather than grouping and counting based on these groups through Lines 1-6.

7. IMPLEMENTATION

In order to take the advantage of CEP for forensic analysis of large log files, we were to implement an existing CEP software package, or create our own. Among others, we have selected Esper as it is open source and provides engine API that allows for creating standalone applications. We have built our proof of concept (PoC) application and authored the code in Java based on Esper's Java

```
   @Name('Split::Scans)
1  INSERT INTO ScanStream
2  SELECT ip, method, response, min(timestamp) as
   starttime, max(timestamp) as endtime, count(*) as cnt
3  FROM ApacheLogRecordStream().win:ext_timed
   (timestamp.toMillisec(), 1 sec)
4  GROUP BY ip, method, response
5  HAVING count(*) > 30;

   @Name('Output::WPSuspiciousScan')
6  SELECT a.ip||": Successful Attack at: "||b.url as source,
   a.starttime , a.endtime as _endtime, b.timestamp as
7  endtime
   FROM PATTERN [every-distinct(ip, method,
   response, starttime, endtime)
            a=ScanStream() ->
            b=ApacheLogRecordStream(ip=a.ip and
   method=a.method and response!=a.response and
   response!=404 and (timestamp.toMillisec() -
   a.endtime.toMillisec()) <= 2000)];
```

Query 10. Suspicious Scan

package. As CEP engines typically process events in real time, we have decided to reconstruct events in order to take the advantage of the complex event-processing model. After reconstruction, we need to make it certain our PoC application makes CEP engine run queries safely, that is, consuming reasonable memory and CPU resources along with producing accurate output.

7.1 Reconstruction of Log Records

In order to reconstruct the activities through Apache Log entries we first need to read records from log files and then send those records to the engine as events. To do so, we have created an adapter in Java, called ApacheLogInputAdapter, for the Apache log forma, because ESPER provides only 7 adapters (e.g. amqp, csv, http, etc) and none of them have been supporting Apache Log files as of writing.

Through the adapter, we have employed special regular expression (a.k.a regex) to parse log records. After running the adapter for several log files, we encountered log files and records with errors. To eliminate distraction, we have excluded non-compliant records from the search process.

Since we deal with post mortem identification and extraction of evidence from log records, we lack capturing events in real-time; thus, we need timestamp values bound to records. Unfortunately, log files may not store records in the correct order all the time. According to our examinations on log files, timely disordered events occur in small sizes of time windows. That is, an event occurred earlier may be stored only a few seconds later and after some records with bigger timestamp values, although it has smaller timestamp value. To deal with this problem, we employ sorting through an EPL query with a reasonable time window and make sure the

actual queries do not get confused with timing disorder. Query 11 is the sample query to sort the records over timestamp field.

```
   @NAME ('Sort::ApacheLogEvent')
1  INSERT INTO ApacheLogRecordStream
2  SELECT * FROM ApacheLogEvent.win:length(10)
3  OUTPUT EVERY 1 events
4  ORDER BY timestamp;
```

Query 11. Sorting

As a part of the enrichment process, forensic analysts may need some transformations of some field values into some other related information, such as IP addresses to country match. Although we have not utilized all of them, we have created some user-defined functions, not only to use through the predicates of EPL queries, but also to provide better output through SELECT clauses.

7.2 Adjusting EPL Queries

Investigators could create EPL queries that can query same patterns with slight differences. For example, in Pattern VI we have developed a pattern to match the administrative activity using only "%wp-admin%" and ((method="GET" and response=200) or (method="POST" and response=302))" predicate. However, having inspected the results of those queries, we have observed so many false positives. That is, there are pages, scripts, or files under wp-admin directory that can be requested and retrieved, while do not require the user to be logged in and to have administrative privileges. Therefore, in order to mitigate the false positives, we adjust our query adding additional predicates, such as "URL not like '%http%' and URL not like '%/wp-admin/css/%' and URL not like '%/wp-admin/js/%' and URL not like '%/wp-admin/admin-ajax.php%'"

Besides adjustment based on predicates, when not tuned well, threshold values of repeating patterns and size values for sliding time windows might cause redundancy in output or false negatives. For example, if we define window size using **win.ext_timed**, then the query slides smoothly over the records and results in producing vast amount of alerts pointing to the same event. Although this mitigates the false positive rate, it raises redundant outputs. There are two possible solutions to mitigate this; (2) changing the statement to **win.ext_timed_batch** that slides windows as batches rather than smooth-slides or (2) adding **every-distinct** statement before pattern definitions through queries. In the former, it is critical to know that batch queries might produce false negatives depending upon the record density.

7.3 Experiments and Results

In order to test and observe how those queries are effective and efficient over actual log files, we have used a system that is running Yosemite-Mac OS on 8Gb Memory and Intel i5 Core 2 CPU. We developed and executed our code using Eclipse and monitored resource consumption through VisualVM.

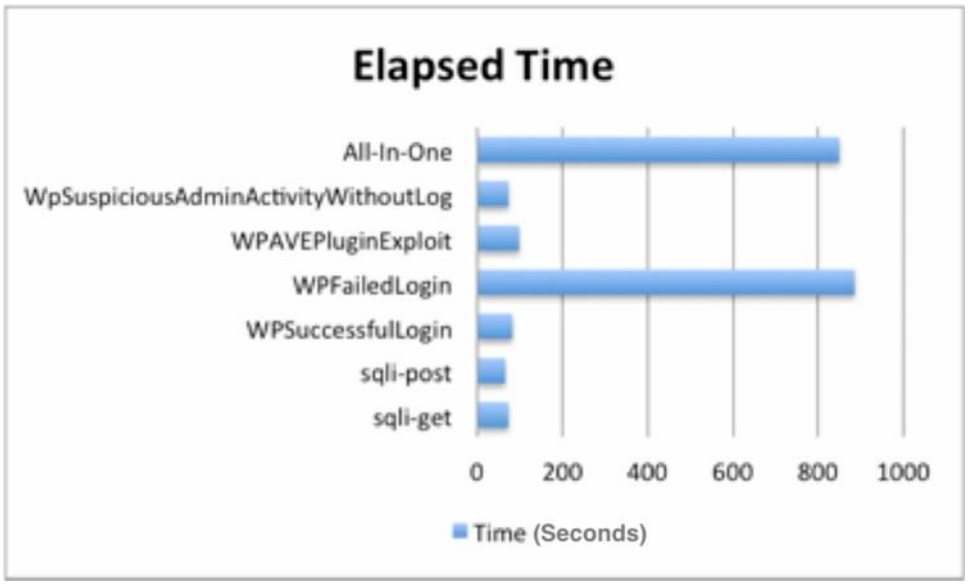

Figure 5. Memory Consumption without Sorting

We have run our queries on real Apache Log files of which web servers are based on WordPress theme and we merged two large log files in 1 Gb sizes and obtained a log file around 2 GB in size. When we have run 6 queries provided in this paper earlier, we have observed periods as depicted in Figure 5. When we gathered all queries in one big query, notice All-In-One in the figure, we have observed elapsed time no more than the longest one among individual runs. This is good news for investigators who want to run several queries once and significantly narrow down their investigations.

Investigators may also need to sort log records depending upon the situation of log records or queries required. Figures below illustrate EPL query-runs over a 10 GB Apache log file. Figure 6.b depicts the query-run with a sorting query, Query 11. With sorting memory consumption raises around 100 mb heap usage.

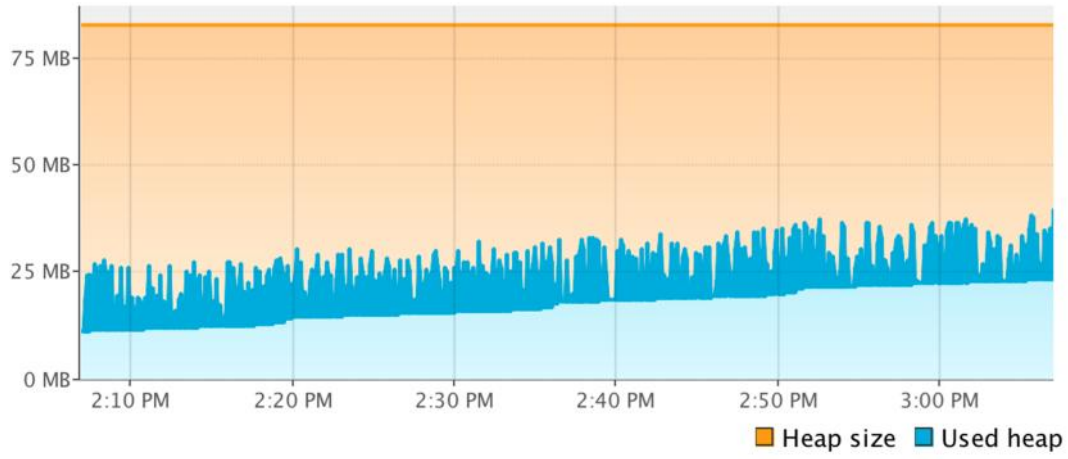

Figure 6a. Memory Consumption without Sorting

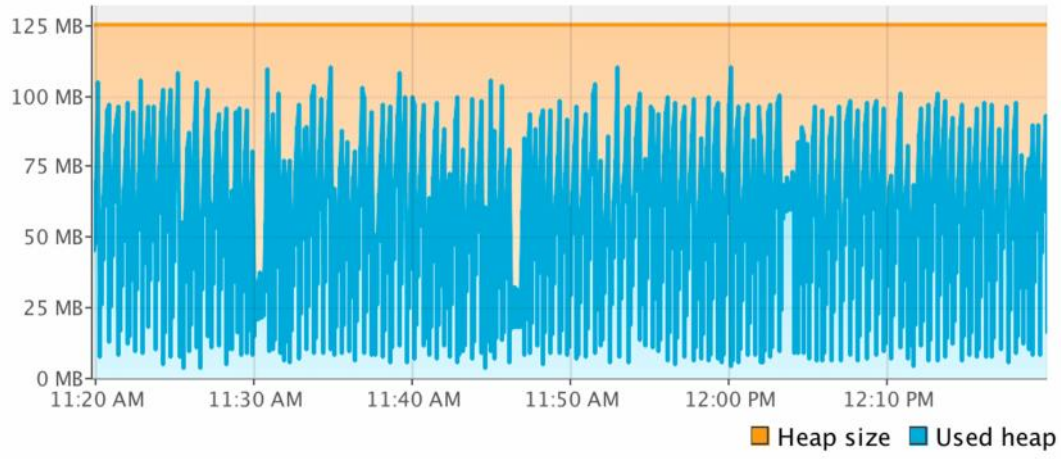

Figure 6b. Memory Consumption with Sorting

Besides the size, content of log files also impact the speed of query-runs. When we have log file with high density of records that match our search patterns which feature groupings over large sliding windows then we observe higher memory consumption and even out of memory errors through smaller heap size. For log file including the traces of (cache) Remote Shell Injection Exploit attack depicted in Figure 4, we have run Query 9. The log file has almost 400K records related to this attack spanning over only 100 minutes. The engine has consumed memory space more than 400 mb as shown in Figure 7.a. CPU usage was also high during this query run.

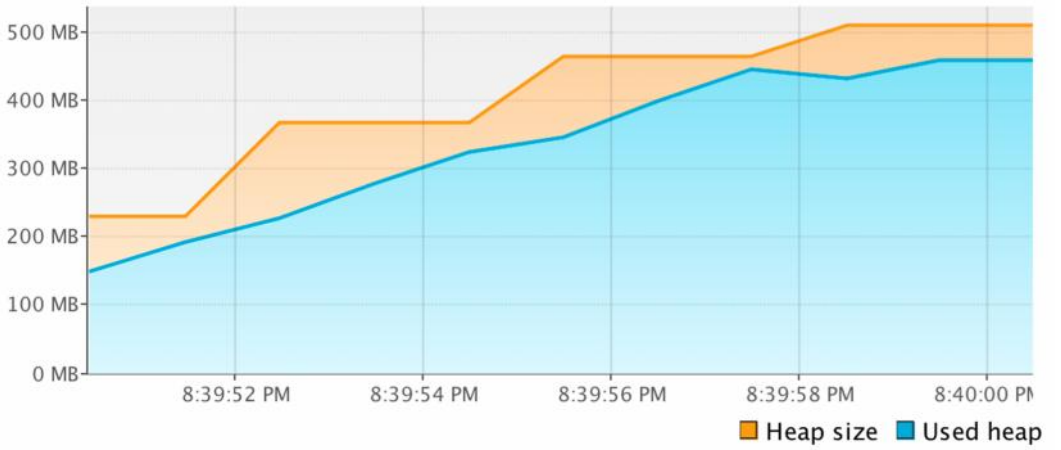

Figure 7a. Memory Usage over High Density Log File

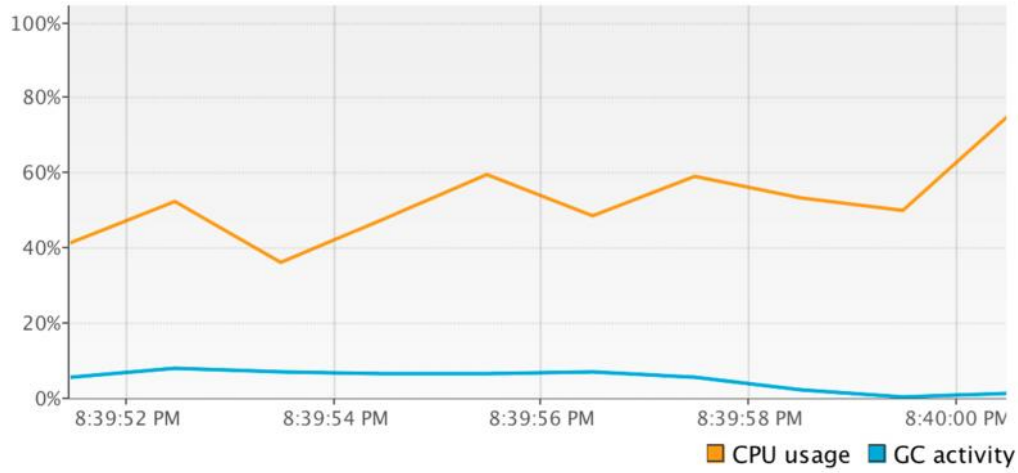

Figure 7b. CPU Usage over High Density Log File

8. CONCLUSION

We have proposed a novel log analysis method using TL based on reconstruction. As they are ubiquitous, we focused on Web Server misuses and Apache log files as a case study. Through definition of misuses, we have developed formal patterns in MSFOMTL and corresponding queries in EPL. In order to run EPL queries, we have built our PoC application upon an open source CEP engine, Esper. Our tests revealed this method can be very efficient and effective through log analysis and would contribute to digital forensics field in several aspects.

Such approach could be used not only through digital investigations for misuse identification and location, but also through business intelligence for use case analysis and cyber-security for enhancing SIEM and host based intrusion detection mechanisms. Investigators can run several queries that have previously been tested and stored in EPL libraries. This would speed their investigations up and allow saving time by focusing on something novel rather than wasting time through the search of already known patterns.

REFERENCES

Ahmed, A., Lisitsa, A., & Dixon, C. (2011). A misuse-based network Intrusion Detection System using Temporal Logic and stream processing. In Network and System Security (NSS), 2011 5th International Conference on (pp. 1–8). http://doi.org/10.1109/ICNSS.2011.6059953

Albek, E., Bax, E., Billock, G., Chandy, K. M., & Swett, I. (2005). An Event Processing Language (EPL) for Building Sense and Respond Applications. In Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International (p. 136b–136b). http://doi.org/10.1109/IPDPS.2005.97

Arasteh, A. R., Debbabi, M., Sakha, A., & Saleh, M. (2007). Analyzing multiple logs for forensic evidence. Digital Investigation, 4, Supplement(0), 82 – 91. http://doi.org/http://dx.doi.org/10.1016/j.diin.2007.06.013

[CMS technologies Web Usage Statistics. (n.d.). Retrieved April 22, 2016, from http://trends.builtwith.com/cms

EsperTech - Esper. (n.d.). Retrieved April 1, 2016, from http://www.espertech.com/esper/

EventFlow and StreamSQL | StreamBase. (n.d.). Retrieved April 23, 2016, from http://www.streambase.com/products/streambasecep/streamsql/

Havens, R. W., Lunt, B., & Teng, C. C. (2012). Naive Bayesian filters for log file analysis: Despam your logs. In 2012 IEEE Network Operations and Management Symposium (pp. 627–630). http://doi.org/10.1109/NOMS.2012.6211972

Huth, M., & Ryan, M. (2004). Logic in Computer Science: Modelling and Reasoning About Systems. New York, NY, USA: Cambridge University Press.

Jayathilake, P. W. D. C. (2011). A novel mind map based approach for log data extraction. In 2011 6th International Conference on Industrial and Information Systems (pp. 130–135). http://doi.org/10.1109/ICIINFS.2011.6038054

J. Herrerías, & R. Gómez. (2010). Log Analysis Towards an Automated Forensic Diagnosis System. Availability, Reliability, and Security, 2010. ARES '10 International Conference on, 659–664. http://doi.org/10.1109/ARES.2010.120

Jimenez-Peris, R. (2015). MASSIF: A Highly Scalable SIEM. Presented at the DEMONS Workshop.

Kalamatianos, T., Kontogiannis, K., & Matthews, P. (2012). Domain Independent Event Analysis for Log Data Reduction. In 2012 IEEE 36th Annual Computer Software and Applications Conference (pp. 225–232). http://doi.org/10.1109/COMPSAC.2012.33

Kavanagh, K. M., & Rochford, O. (2015). Magic Quadrant for Security Information and Event Management. Retrieved from https://www.gartner.com/doc/reprints?id=1-2JNUH1F&ct=150720&st=sb

Makanju, A. A. O., Zincir-Heywood, A. N., & Milios, E. E. (2009). Clustering Event Logs

Using Iterative Partitioning. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1255–1264). New York, NY, USA: ACM. http://doi.org/10.1145/1557019.1557154

Schmerl, S., Koenig, H., Flegel, U., Meier, M., & Rietz, R. (2008). Systematic Signature Engineering by Re-use of Snort Signatures. In Computer Security Applications Conference, 2008. ACSAC 2008. Annual (pp. 23–32). http://doi.org/10.1109/ACSAC.2008.20

StreamBase | Complex Event Processing, Event Stream Processing, StreamBase Streaming Platform. (n.d.). Retrieved April 23, 2016, from http://www.streambase.com/

Vernekar, S. S., & Buchade, A. (2013). MapReduce based log file analysis for system threats and problem identification. In Advance Computing Conference (IACC), 2013 IEEE 3rd International (pp. 831–835). http://doi.org/10.1109/IAdCC.2013.6514334

WordPress <= 2.0.2 - cache Remote Shell Injection Exploit. (n.d.). Retrieved April 1, 2016, from https://www.exploit-db.com/exploits/6/

WordPress Advanced Video Plugin 1.0 - Local File Inclusion LFI. (n.d.). Retrieved April 19, 2016, from https://www.exploit-db.com/exploits/39646/

WP Login Timeout Settings — WordPress Plugins. (n.d.). Retrieved March 26, 2016, from https://wordpress.org/plugins/wp-login-timeout-settings/screenshot

