Log Correlation Engine TASL Reference Guide

January 5, 2015

(Revision 1)

Table of Contents

Introduction

This document describes writing event correlation algorithms for Tenable Network Security's **Log Correlation Engine**. Please share your comments and suggestions with us by emailing them to support@tenable.com.

It is assumed that the reader has the following prerequisite skills:

- A working knowledge of Secure Shell (SSH) and key exchange.
- Experience with the Linux operating systems, including regular expressions.
- Familiarity with Tenable's Log Correlation Engine (LCE) operation and architecture as described in the "Log Correlation Engine Administration and User Guide".
- Knowledge of general log formats from various operating systems, network devices and applications.

Standards and Conventions

Throughout the documentation, filenames, daemons, and executables are indicated with a **`courier bold`** font such as **`gunzip`**, **`httpd`**, and **`/etc/passwd`**.

Command line options and keywords are also indicated with the **`courier bold`** font. Command line examples may or may not include the command line prompt and output text from the results of the command. Command line examples will display the command being run in **`courier bold`** to indicate what the user typed while the sample output generated by the system will be indicated in `courier` (not bold). Following is an example running of the Linux **`pwd`** command:

```
# pwd
/opt/lce
#
```

Important notes and considerations are highlighted with this symbol and grey text boxes.

Tips, examples, and best practices are highlighted with this symbol and white on blue text.

What is TASL?

TASL stands for the "Tenable Application Scripting Language" and is a library based on NASL (the scripting language used for Nessus vulnerability scanner scripts), but independent from the Nessus backend. The format is similar to that used for writing NASLs for the Nessus vulnerability scanner (http://www.nessus.org/), making it easier for those familiar with writing NASLs to write event correlation algorithms for events in the LCE.

TASL is a scripting language, similar to the C programming language in its syntax, without the inherent problems due to memory management and explicit variable definitions. TASL is more secure than C, in the sense that a script cannot dereference a NULL pointer or be vulnerable to a buffer overflow or format string attack. This means that TASL scripts are more robust to coding bugs than anything doing the equivalent task in C.

TASL comes with a set of functions and operators that perform very basic tasks like string manipulation, local command execution, file manipulation and so on. TASL has some object-oriented logic that goes beyond the function calls and makes it easy to perform in an event-driven environment.

The “Object” Concept

TASL is an event-oriented language. This means that a typical TASL script does not have a “main” loop, but creates “objects” that register callbacks that will be called by the **`lced`** process when a specific condition occurs.

Note that the term “object” is a bit of a misnomer here. TASL is not really object oriented. What we define by “object” is a set of functions and statistics associated to a set of events.

After a user creates an object, it calls the function **`object.filter.event.id.add()`** to tell the LCE which events it is interested in (we call that process “subscribing to events”). Then it registers one or several callback functions by calling **`object.callback.add()`**, and the LCE will call the callback(s) whenever the events the object subscribed to occur. This will become clearer with a quick example:

```
SNORT_UDP_EVENT = 5102;    # Snort-UDP_Event
obj = object();
obj.filter.ip.src.add("0.0.0.0/0");
obj.callback.add("myCallback");
obj.filter.event.id.add(SNORT_UDP_EVENT);

function myCallback(obj)
     {
     local_var count, last30;
     last30 = unixtime() - 30;
     count = obj.event.count.get(id:SNORT_UDP_EVENT,
                            src:obj.event.src(),
                            newer_than:last30);
     if ( count >= 3 )
          {
          obj.log(src:obj.event.src(),
                  dst:obj.event.dst(),
                  protocol:IPPROTO_UDP,
                  sport:obj.event.sport(),
                  dport:obj.event.dport(),
                  msg:'Worm_Probe - Multiple SQL Worm Probe -
                  Snort Detect');
       # Don't report this host for another 24 hours
       obj.filter.ip.src.remove.until(ip:obj.event.src(),
                           until:unixtime() + 24*3600);
          }
     }
```

In this example, the first line defines a variable named SNORT_UDP_EVENT and it is assigned to the LCE plugin ID #5102, which is used to produce that event. The following four lines create the “object”, accept any event from any IP address, invoke our function whenever we have a matching event and then filter our event stream for just SNORT_UDP_EVENT messages. This script then “sits” in the LCE’s memory until a matching event occurs. At that point, the LCE invokes the function “myCallback”. In this case, the function counts all Snort UDP events that have occurred from the source IP of the current event in the last 30 seconds. If there are three or more events, a new LCE event is created and the source IP address of the event is ignored for 24 hours.

Subscribing to a set of events is a costly operation - for every event that we subscribed to, the LCE will keep track of all its occurrences over time. This means that the more events your objects subscribe to, the more memory the **`lced`** process will consume. Be sure to only subscribe to the events you are really interested in. Each object tracks the last five million occurrences of the events it has been subscribed to.

Once an object subscribes to events, it can query their amount of occurrences over time, by source IP, source port, destination IP or destination port. This allows a script-writer to write very efficient scripts that can have very versatile functions, from detecting a Distributed Denial of Service (DDoS) attack against a web server to detecting an attacker sweeping the network for open ports.

Architecture

Once an object receives events, it can perform several actions. The following is a list of some example uses for TASL scripts.

- It can query the number of occurrences of a particular event with a powerful set of filtering functions (event occurrences can be filtered by source IP, destination IP, source port, destination port, protocol and time). For instance, if an object subscribes to the event "a root-privileged application crashes", it can perform a simple query to know how many times the host in question had a root-privileged application crash over the last five hours.
- It can query the content (text) of the event that triggered the callback. This can be used to write better parsers than what can be done with the **`.prm`** files. Note that TASL scripts are noticeably slower than **`.prm`** plugins, and may therefore cause a performance hit. For example, do not use TASL scripts for basic pattern matching when the same job can be done much more efficiently in a **`.prm`** file.
- It can execute commands on the local system. This can be used to integrate with a wide variety of applications. A mail program is one such application that can be integrated in order to provide real-time alerts when a host is being compromised.
- It can read from and write to local files.
- It can inject an event in the LCE queue, which itself may be processed by **`.prm`** files and TASL scripts.

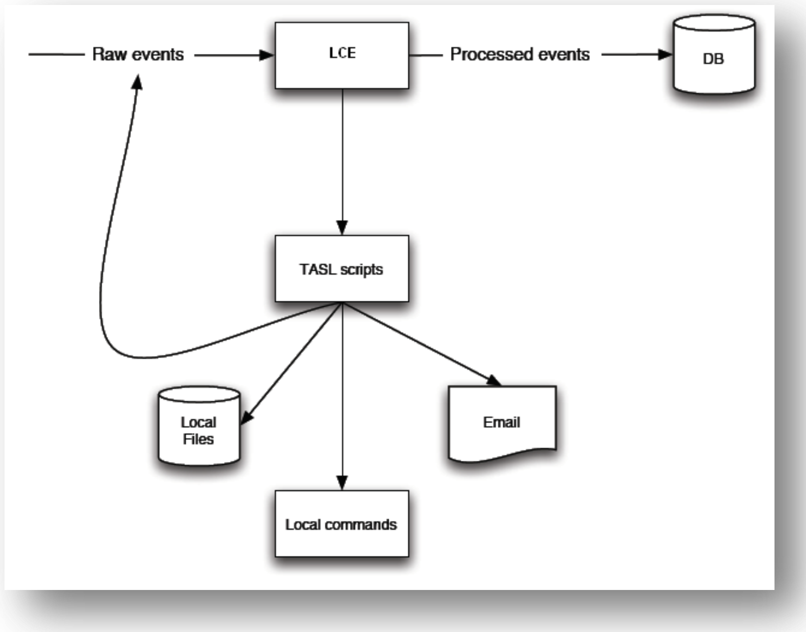
Log messages arrive at the **`lced`** process via SYSLOG or passed by LCE clients. If the log message matches a particular **`.prm`** plugin, and the plugin has been subscribed to by one or more TASL objects, the logic of the corresponding TASL scripts will be invoked.

We will learn later in this document about how to write various algorithms with TASL. However, it is important to realize how TASL output can fit into the overall architecture of the LCE and the SecurityCenter.

When a TASL script concludes that something needs to be logged, there are several logging choices.

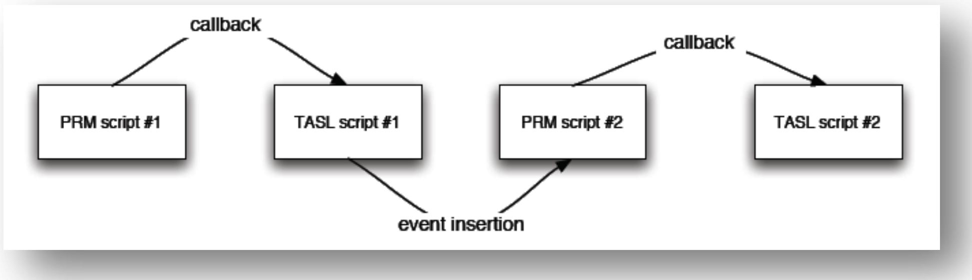
To simply add new events right back into the LCE, a new message can be created and sent via the **`log()`** API. In order for the LCE to understand the message sent by the TASL script, a corresponding **`.prm`** file will be needed to parse the output of the TASL script. In the example TASL scripts later in this document, each one will have their own **`.prm`** plugin.

TASL scripts may also invoke local Linux commands, and therefore can use tools available to the "lce" user on the host (such as mail programs and syslog programs) to send alerts to users or other systems when a desired condition is met. The **`/opt/lce/tools/`** directory provides some of these programs.



As previously stated, TASL scripts cannot directly write to the LCE database. All TASL scripts need a corresponding set of LCE **`.prm`** libraries to parse the events again. This design is elegant and makes sure that a TASL script will not corrupt a database by writing directly to it or by sending malformed events types or names.

Additionally, this design allows the chaining of several TASL scripts. Consider the following architecture:



TASL can be used to write extremely sophisticated algorithms and keep state across several million correlated events. However, it is even more powerful to realize that multiple TASL scripts can be chained together to produce even more intelligent results. Here are some basic examples:

- Counting large numbers of events from specific IP addresses can be optimized by having smaller scripts that count occurrences of 100 events, and then other scripts that count the number of “100 Event” events. Consider a “login failure event” that had a TASL script that produced a “100 Login Failure Event”. A second TASL script that performed additional logic on these “100 Login Failure Event” events would be invoked less often.

- There may be multiple TASL scripts that use different algorithms to produce the same event name. Consider the process of detecting SPAM email. Multiple TASL scripts could be used to all produce various alerts about an IP address that is sending or relaying SPAM and a final master TASL script may look at the occurrence of all these other TASL events and use a heuristic or fuzzy logic to ultimately decide SPAM is occurring.

- Event re-writing can be used to draw attention to the output of other TASL scripts. For example, a very interesting TASL may conclude that worm behavior is occurring on a particular host and generate a “Worm-Outbreak” alert. A second TASL may consume this alert and see that the attack is indeed coming from a well-known network address, such as a server farm and produce a new alert named “Worm-Outbreak-ServerFarm”.

- For extremely high-volume events, chained TASL scripts can be used to “turn on” collection and monitoring for specific IP addresses. For example, a LCE server may be collecting and logging large amounts of “netflow” data. It may be impractical to launch a TASL script for each network connection, however, possibly after observing a set of IDS events, a second TASL script may temporarily subscribe to any “netflow” event for the host of interest for a short period of time.

In the “LCE Log Normalization Guide” we learned that each normalized event gets a unique name as well as a more general type. Many of Tenable’s TASL scripts take advantage of this fact and subscribe to all events of a certain type. This technique is more flexible than having to hard-code specific event IDs into TASL scripts.

TASL Syntax

The syntax, operators and control operators in TASL are extremely similar to C. However, some aspects are similar to Perl and the function arguments work slightly differently. For example, the “#” character can be used anywhere to indicate a comment.

Variables

Unlike in C, variables do not need to be declared. TASL is a loosely typed language that recognizes internally only three types of variables:

1. strings
2. integers
3. objects

Strings can be converted to integers or integers to strings using the **`string()`** and **`int()`** functions. The variable names do not have to contain any special characters. In particular, the dollar sign ($) imposed by Perl is not necessary.

Example:

```
a = 42;
b = "foo";
c = string(a, b); # "c" now equals to "42foo"
```

Operators

The following operators are recognized in TASL:

`+, +=`	: addition, string concatenation
`--, ++`	: variable incrementing (as in “i ++”)
`-, -=`	: subtraction, string removal
`*, *=`	: multiplication
`/, /=`	: division
`%, %=`	: modulo
`**`	: power
`><`	: substring

The “**><**” operator is TASL specific and returns true if a substring is in another string.

Example:

```
root_present = 0;
a = "root";
b = "login from user root";
if ( a >< b ) root_present = 1;
```

Control Operators

The following control operators are recognized in TASL:

- `for ( init ; condition ; post_action )`
- `while ( condition )`
- `foreach item (array) { }`

A First TASL Script

Starting with LCE 3.6.1, by default the LCE runs all of the Tenable-provided TASLs. If these TASLs are modified, subsequent plugin updates will overwrite the changes. Instead of modifying the Tenable-provided TASLs, disable the existing TASL by adding a reference to the TASL file name in the “TASL and Plugins” section of the LCE GUI , make a copy of the TASL, edit it as specified in this document and then restart the LCE daemon so that it uses the new changes.

The first script we will write detects if the remote host is infected by an SSH worm. For this example, a rampant SSH worm is going around that scans hosts and attempts to log into them by performing between 20 and 2000 brute force login attempts. Therefore, we need a plugin that:

- Monitors SSH login failures events
- For every failed login attempt, counts how many times the remote host attempted to log into the system during the last 10 minutes
- Injects an event in the LCE to notify that a SSH worm is running

This script will require a companion **`.prm`** file that parses the injected event and stores it in the LCE database.

TASL scripts have a **`.tasl`** extension. By default, they are stored in the **`/opt/lce/daemons/plugins`** directory.

Our first TASL script looks like this:

```
function myCallback(obj)
     {
     local_var count, last5mn;
     last5mn = unixtime() - 60*5;
     count = obj.event.count.get( id:BAD_LOGIN1,
       src:obj.event.src(),
       newer_than:last5mn);
     count += obj.event.count.get(id:BAD_LOGIN2,
                                  src:obj.event.src(),
                                  newer_than:last5mn);
     count += obj.event.count.get(id:BAD_LOGIN3,
                                  src:obj.event.src(),
                                  newer_than:last5mn);
     if ( count >= 10 )
          {
          obj.log(src:obj.event.src(),
                  dst:obj.event.dst(),
                  protocol:IPPROTO_TCP,
                  sport:obj.event.sport(),
                  dport:obj.event.dport(),
                  msg:'SSH_Worm - The remote host is
                  infected by an SSH worm');
          # Don't ever report this host again
          obj.filter.ip.src.remove(obj.event.src());
          }
     }

# This is our "main"
BAD_LOGIN2 = 1806;
BAD_LOGIN1 = 1815;
BAD_LOGIN3 = 1817;
obj = object();
obj.filter.ip.src.add("0.0.0.0/0");
obj.callback.add("myCallback");
obj.filter.event.id.add(BAD_LOGIN1, BAD_LOGIN2, BAD_LOGIN3);
```

As you can see, our script contains a function called "**`myCallback`**" and a bunch of statements in what we call the "**`main`**" section ("**`main`**" really means "outside of any function").

The way TASL works is that a script will register the objects it needs, then exits. The LCE will call the relevant callbacks when needed, based on the subscribed events. This means that a TASL script does not really have a main loop. It simply subscribes and then exits. This concept is very important, because it is the backbone of the whole TASL architecture.

Let's look at our main function first:

```
BAD_LOGIN2 = 1806;
BAD_LOGIN1 = 1815;
BAD_LOGIN3 = 1817;
```

TASL, like any modern language, supports the use of variables. For the sake of code readability, we define what we call "bad logins" based on the **`.prm`** plugins #1815, #1806 and #1817. These plugins detect SSH login failures on Linux servers and are present in the **`ssh_openssh.prm`** plugin library.

For a concise list of event names, types, and IDs to which TASLs can subscribe see **`prm_map.prm`** in the LCE plugins directory (**`/opt/lce/daemons/plugins/`** by default). This file lists the PRM files and their associated Plugin IDs, event name, and event type.

These are the events we are interested in. It is also worth noting that each TASL script lives in its own world – two TASL scripts will not explicitly share any variable, so there is no risk of collision between the global variables of several scripts.

Global variables can be called from within the TASL callback functions. This allows the script-writer to maintain a state between two callbacks.

The following statement creates the object "obj":

```
obj = object();
```

Unlike other variables in TASL, objects need to be explicitly created with the function named "**`object()`**". This function takes no arguments and simply creates a memory structure for the object. It registers this object within the LCE. Every time a new event occurs, the LCE will walk the list of registered objects and call the relevant callback functions. Therefore, creating an object must be considered a costly operation; every time an object is created, the LCE will spend more CPU cycles going through the list. Ideally, objects should only be created in the main section of a script, although it is legal to create one from within a callback function.

```
obj.filter.ip.src.add("0.0.0.0/0");
```

Each object is associated with a set of filters. In particular, the filters are the source IP addresses we are interested in, the destination IP addresses we are interested in (as well as those we are **not** interested in) and the events we want or do not want to subscribe to. The statement above explicitly states that we are interested in every IP address. The behavior of the IP filter is described in more detail in the section about **`obj.filter.ip.*`** later on.

```
obj.callback.add("myCallback");
```

The statement above tells **`lced`** to call the function "**`myCallback()`**" when all the filters of this object match.

```
obj.filter.event.id.add(BAD_LOGIN1, BAD_LOGIN2, BAD_LOGIN3);
```

Our object now subscribes the events BAD_LOGIN1, BAD_LOGIN2 and BAD_LOGIN3. The method **`filter.event.id.add()`**, accepts any number of arguments.

So far our object has done the following:

- It has been created.
- It explicitly told **`lced`** it wants to know about every host.
- It asked **`lced`** to call the function "**`myCallback()`**" when the filter matches.
- It subscribed to events 1806, 1815, and 1817, which were three different SSH login failure plugins.

Next, have a look at the function "**`myCallback`**" that contains the logic of the software:

```
function myCallback(obj)
```

TASL contains a shell-like syntax to define new functions. It is simply "**`function <funcName>(arg1, [arg2, ...])`**". Callback functions only accept one argument, which is the object that registered the callback. This is useful as a single TASL script creating two objects that can share the same callback function.

```
local_var id, count, last5mn;
```

Functions can have their own local variables that will be freed whenever the execution tree exits from the function. The reserved word "local_var" is equivalent to Perl's "my" reserved word that has been designed for the same purpose.

```
last5mn = unixtime() - 60*5;
```

The function **`unixtime()`** returns the value of time in seconds since epoch (the same as a time(NULL) would do in C). In the statement above, we compute when it was "5 minutes ago".

```
count = obj.event.count.get(id:BAD_LOGIN1, src:obj.event.src(),
                            newer_than:last5mn);
count += obj.event.count.get(id:BAD_LOGIN2, src:obj.event.src(),
  newer_than:last5mn);
count += obj.event.count.get(id:BAD_LOGIN3, src:obj.event.src(),
  newer_than:last5mn);
```

This is where things are starting to become interesting. We are in the function "**`myCallback`**", called by events the object "**`obj`**". Whenever an object is called back, it has access to a set of methods that are stored under **`object.event.*`**. These methods contain information about the current event: its raw text, its source IP, destination IP, protocol, source port, destination port and time.

What we want to do now is to compute the number of bad SSH logins the current host (which generated this event) has made. So we call the method **`obj.event.src()`** to get the source IP of the current host, and use it as a filter for the method **`obj.event.count.get()`**, which returns the number of times an event matching the filter set has occurred. So the following call really means: "return the number of events of type BAD_LOGIN1 that have occurred from the current host during the last five minutes":

```
obj.event.count.get(id:BAD_LOGIN1, src:obj.event.src(),
                    newer_than:last5mn);
```

As the SSH worm actually performs a brute force attack against a single host (and not a sweeping attack) we could narrow down our filter even more:

```
obj.event.count.get(id:BAD_LOGIN1, src:obj.event.src(),
                    dst:obj.event.dst(), newer_than:last5mn);
```

This new call means "return the number of events of type BAD_LOGIN1 that have occurred between the current source host and its current target over the last five minutes". By narrowing down our filter this much, we could easily distinguish the SSH worm from an attacker sweeping the network for a default "root/root" SSH account (this in turn would generate only one bad login per host).

Since there are three SSH bad login events, we perform the same call from BAD_LOGIN2 and BAD_LOGIN3, and we add the results using the "**+=**" operator, which is a shortcut for adding the result of an operation to the current value.

Note that in this particular case, doing three function calls is redundant. We could obtain the same count result by not specifying any ID in our request and calling this method only once.

```
if ( count >= 10 )
     {
     obj.log(src:obj.event.src(),
             dst:obj.event.dst(),
             protocol:IPPROTO_TCP,
             sport:obj.event.sport(),
             dport:obj.event.dport(),
             msg:'SSH_Worm - The remote host is infected
             by an SSH worm');
     }
```

If more than 10 events matched the calls we did above, then we know the remote host is infected by a SSH worm. Therefore, we decide to report on it using the method **`obj.log()`**, with the message "SSH_Worm - The remote host is infected by an SSH worm".

A companion **`.prm`** file will have to parse this message for it to appear in the LCE database. This will be covered shortly.

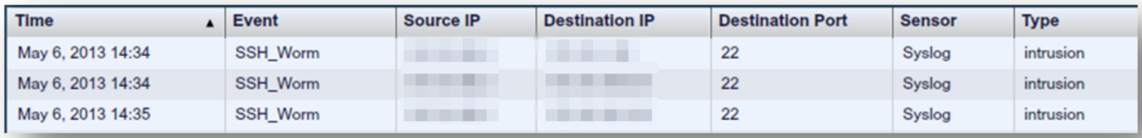
```
obj.filter.ip.src.remove(obj.event.src());
```

Since SSH worms tend to come back often, we do not want our object to report on this attacking host ever again, so we simply call the method **`obj.filter.ip.src.remove()`**. Alternatively, we could block logs from this IP address for a specific period of time.

Once this is done, we need to write a **`.prm`** file that will report on the generated event:

```
id=9902
name=SSH Worm
example=SSH_Worm - The remote host is infected by an SSH worm
match=SSH_Worm - The remote host is infected by an SSH worm
log=event:SSH_Worm type:system
```

Once these two files are written (the TASL script and the **`.prm`** file) copy them into the directory **`/opt/lce/daemons/plugins`** and restart **`lced`**. As events occur, they will be viewable in the report such as:



Time	Event	Source IP	Destination IP	Destination Port	Sensor	Type
May 6, 2013 14:34	SSH_Worm	[redacted]	[redacted]	22	Syslog	intrusion
May 6, 2013 14:34	SSH_Worm	[redacted]	[redacted]	22	Syslog	intrusion
May 6, 2013 14:35	SSH_Worm	[redacted]	[redacted]	22	Syslog	intrusion

The **`lced`** process expects log-parsing plugins to have a file extension of **`.prm`** and TASL scripts to have an extension of **`.tasl`**.

TASL Object Filter Methods

The **`object.filter.*`** set of methods contains functions that tell **`lced`** the conditions that must match in order to call the registered callbacks of an object. There are two subclasses: **`object.filter.event.*`** and **`object.filter.ip.*`**.

Event Filters

`object.filter.event.id.add(<id>[, <id2>, ... <idN>])`
`object.filter.event.id.rm(<id> [, <id2>, ... <idN>])`

These two methods subscribe and unsubscribe to/from a set of plugin IDs. They accept any number of arguments. An object may only subscribe to 250 items in parallel.

Example:

```
      obj.filter.event.id.add(123, 456);
      obj.filter.event.id.add(42);
```

Direction Filters

`obj.filter.event.direction.rm(DIRECTION_INTERNAL);`
`obj.filter.event.direction.rm(DIRECTION_EXTERNAL);`

These two object filters allow for filtering of data by direction. The directions that can be filtered are listed below. By default, all directions are considered unless directionality filters have been used with **`obj.event.direction.rm(direction)`**.

```
DIRECTION_INTERNAL
DIRECTION_INBOUND
DIRECTION_OUTBOUND
DIRECTION_EXTERNAL
```

```
obj = object();
obj.callback.add("mycallback");

# outside of the callback add filters
```

```
obj.filter.event.direction.rm(DIRECTION_INTERNAL);
obj.filter.event.direction.rm(DIRECTION_EXTERNAL);

# in the callback check the other possible direction types
Function mycallback(obj)
{
             if( obj.event.direction() == DIRECTION_OUTBOUND )
              {
                    # handle outbound event
              }
             else
              {
                   # handle inbound event
              }

}
```

IP Filters

`object.filter.ip.src.add(<ip|ip/mask|ip/cidr>, [<ip,...>])`
`object.filter.ip.src.remove(<ip|ip/mask|ip/cidr>, [<...>])`
`object.filter.ip.dst.add(<ip|ip/mask|ip/cidr>, [<...>])`
`object.filter.ip.dst.remove(<ip|ip/mask|ip/cidr>, [<...>])`

These methods are used to filter which hosts we want to hear events from. They take any number of arguments, each argument being under one of the following form:

- IP (e.g., “192.168.0.1”)
- IP/Mask (e.g., “192.168.0.0/255.255.255.0”)
- IP/CIDR (e.g., “192.168.0.0/24”)

If no filter is set, then all events go through (i.e.,“default accept”). When multiple conflicting ranges are entered, the last matching range is the one that decides if an event goes through.

Example:

```
# Monitor 192.168.0.0/24
obj.filter.ip.src.add("192.168.0.0/24");

# Except 192.168.0.1
obj.filter.ip.src.remove("192.168.0.1");

# Or after all, do monitor 192.168.0.1
obj.filter.ip.src.add("192.168.0.1");
```

`object.filter.ip.src.add.until(ip:<ip>, until:<timestamp>)`
`object.filter.ip.src.remove.until(ip:<ip>, until:<timestamp>)`
`object.filter.ip.dst.add.until(ip:<ip>, until:<timestamp>)`
`object.filter.ip.dst.remove.until(ip:<ip>, until:<timestamp>)`

These methods are used to temporarily add or remove a given set of hosts.

The “**`until`**” method has a single argument. It is a Linux timestamp that indicates when the rule will expire.

Example:

```
# Monitor 10.0.0.1 for the next 10 minutes
expiration = unixtime() + 60*10;
obj.filter.ip.src.add.until(ip: "10.0.0.1", until:expiration);
```

Port Filters

`obj.filter.event.sport.add(int:port);`
`obj.filter.event.dport.add(int:port);`
`obj.filter.event.sport.rm(int:port);`
`obj.filter.event.dport.rm(int:port);`

Each TASL script can be filtered to only accept or reject events on specific ports. Ports can be specified for specific sources or destinations. Multiple filters are “ORed” together, not “ANDed”. TASL scripts that wish to subscribe to one port in general (like port “80”), must issue two separate filter statements for both source and destination ports.

Example:

```
# Subscribe to destination port 21 events
object.filter.event.dport.add(21);

# Subscribe to any port 25 events against
# our SMTP server at 192.168.0.5
object.filter.ip.dst.add("192.168.0.5");
object.filter.event.dport.add(25);
object.filter.event.sport.add(25);
```

TASL Object Event Methods

Basic Event Information

The **`object.event.*`** set of methods returns information about the current event (the one which triggered the callback). They must not be called from outside of a callback function.

`object.event.id()`

Returns the ID of the current event.

`object.event.src()`
`object.event.dst()`

Returns the source IP and destination IP of the current event. The IP is returned as a string.

`object.event.sport()`
`object.event.dport()`
`object.event.protocol()`

Returns the source port, destination port, and protocol of the current event.

`obj.event.direction()`

Returns one of the following directions that have been defined:

```
DIRECTION_INTERNAL
DIRECTION_INBOUND
```

```
DIRECTION_OUTBOUND
DIRECTION_EXTERNAL
```

`object.event.data()`

Returns the raw text (string) of the event. This would be the actual entire SYSLOG message or the entire log as sent by the LCE agent.

`object.event.user()`

Returns the username associated with the log that triggered the event. The **`object.event.user()`** can also pass usernames as filters to other functions like the functions listed below.

```
obj.event.count.get(user:"some_username")
obj.event.time.since.first(user:"some_username")
obj.event.time.since.last(user:"some_username")
obj.event.count.remove(user:"some_username")
```

Event Counting

```
object.event.count.get([id:<id>, src:<src>, dst:<dst>, sport:<sport>, dport:<dport>,
                       protocol:<protocol>, newer_than:<date>, older_than:<date>])
```

Returns the number of times the event matches the conditions above. If **`<id>`** is not set, then all the monitored events are taken in account.

Examples:

```
# Count all subscribed events have ever occurred:
count = obj.event.count.get();

# Count all the events from 1.2.3.4:
count = obj.event.count.get(src: "1.2.3.4");

# Count all the events of id 1234 which have occurred
# during the last 10 seconds :
count = obj.event.count.get(id:1234, newer_than:unixtime() - 10);
```

Removing Event Counts

```
object.event.count.remove([id:<id>, src:<src>, dst:<dst>, sport:<sport>, dport:<dport>,
                          protocol:<protocol>, newer_than:<date>, older_than:<date>])
```

This function resets the counter of the events that match the conditions above. Obviously, it does not affect the LCE databases but the number of occurrences of the event our TASL script is aware of.

Examples:

```
# Delete every event occurrence we know of
obj.event.count.remove();

# Delete every event occurrence which took
# place over 5 minutes ago :
obj.event.count.remove(older_than:unixtime() - 5*60);

# Delete every event coming from host 10.0.0.1 :
obj.event.count.remove(src: "10.0.0.1");
```

Determine the Amount of Time Since a Previous Event

```
obj.event.time.since.last(id:<id>, src:<src>, dst:<dst>, sport:<sport>, dport:<dport>,
                          protocol:<protocol>);
obj.event.time.since.first(id:<id>, src:<src>, dst:<dst>, sport:<sport>, dport:<dport>,
                           protocol:<protocol>);
```

Detecting interesting security events is often a function of how often they occur. TASL has the ability to quickly provide the number of seconds since a particular event has occurred. Do not confuse this feature with the previous API that can count the number of matching events in a given time. This event returns the specific amount of time, not a count of events.

For example, consider the two Passive Vulnerability Scanner IDS events “new host detected” and “internal port scanning”. It would be useful to alert if any “new host” has begun to port scan. A TASL script that subscribed to both of these events for processing could easily be written. It would take action any time an “internal port scanning” alert was received. The action taken would be to test for the presence of a “new host detected” event. If it occurred within a few minutes, there may be a correlation.

If the time does not match, it will return a value of 0 or -1. Any TASL script logic can account for this, otherwise unexpected results may occur. For instance, consider a test to see if an event has occurred in less than five minutes. If the event never occurred, both 0 and -1 seconds are less than five minutes.

The **`obj.event.time.since.first()`** function returns the amount of time since the absolute first occurrence (from the LCE’s point of view) of a particular event. This may be useful for finding out the start of a worm outbreak, a vulnerability scan or other types of activity.

TASL Miscellaneous Object Methods

This category contains two methods to add and remove callback functions:

```
object.callback.add(<name>, [<name2>, ...])
object.callback.remove(<name>, [<name2>, ...])
```

These methods add or remove a callback function from an object.

```
      object.log()
```

This method injects data in the LCE directly:

```
object.log(msg:<text>, [src:<ip>, dst:<ip>, sport:<port>, dport:<dport>,
           protocol:<protocol>])
```

Protocol must be one of IPPROTO_TCP, IPPROTO_UDP or IPPROTO_ICMP.

String Manipulation Functions

Several built-in functions are included in TASL:

Function	Description
`string(var1 [, var2, var3...varN])`	The **`string()`** function adds two strings together to form a new string. If one of the arguments is an integer, it will be treated as a string. Any number of variables can be used to create a new string. Example:

	`a = 42;`
	`b = "foo";`
	`c = string(a, b); # "c" now equals to "42foo"`

`raw_string(var1 [, var2, var3...varN])`	This is similar to the **`string()`** function except that non-strings are handled as ASCII characters. A simple example is shown below. Our example TASL script, **`example.tasl`**, is printed out and then invoked with the **`tasl`** command line testing tool.
	
	`[root@linux tasl]# cat example.tasl`
	`number1 = 1;`
	`number2 = 2;`
	`string1 = raw_string(number1,number2);`
	`display(string1);`
	`display("\n");`
	`string2 = string(number1,number2);`
	`display(string2);`
	`display("\n");`
	`string3 = hexstr(string1);`
	`display(string3);`
	`display("\n");`
	`[root@linux tasl]# ./tasl example.tasl`
	`..`
	`12`
	`0102`
	`[root@linux tasl]#`
	
	In the above example, notice that we have two number variables, number1 and number2. When they are combined with the **`raw_string()`** function, the new string is the actual bytes of these variables merged into one long string. The actual byte values are #1 and #2. For comparison, the **`string()`** and **`hexstr()`** functions are used on the resulting data. Any binary output is printed as a dot by the **`display()`** function.

`strcat(var1 [, var2, var3...varN])`	The **`strcat()`** function appends one or more strings to an initial string.
	
	Example:
	`a = 42;`
	`b = "foo";`
	`a = strcat(a, b); # "a" now equals "42foo"`

`display(var1 [, var2, var3...varN])`	This causes the TASL script to print out a message to STDOUT. When the **`lced`** process is running, messages with the **`display()`** function will be printed out to the shell invoked to run the LCE. TASL writers who wish to display debug messages should also consider random file access, or the **`syslog()`** function. Example: `display("This is a test message\n");`

`ord(<str>)`	This function returns the numeric value of the first byte of the string.
	
	Example:
	`[root@linux tasl]# cat example.tasl`
	`display(ord("Test String"),"\n");`
	`[root@linux tasl]# ./tasl example.tasl`
	`84`

```
[root@linux tasl]#
```

The number 84 is the ASCII code for “T”, which was the first character in the string used by the **`ord()`** function.

`hex(<str>)`

The **`hex()`** function returns a string of the hexadecimal value of the number specified in the argument.

```
[root@linux tasl]# cat example.tasl
value = hex(10);
display(value, "\n");
[root@linux tasl]# ./tasl example.tasl
0x0a
[root@linux tasl]#
```

`hexstr(<str>)`

This function accepts a string and returns a new string containing the hexadecimal value of every byte in the string.

```
[root@linux tasl]# cat example.tasl
value = hexstr("10");
display(value,"\n");
[root@linux tasl]# ./tasl example.tasl
3130
[root@linux tasl]#
```

In the above example, the number “3130” is based on the hexadecimal value of the character “1” and the character “0”.

`strstr(<str>, <substr>)`

The **`strstr()`** function searches for a specific substring inside another string. If a match is found, a new string is returned that consists of all the characters at the point of the match until the remainder of the string. If a match is not found, an empty string is returned. Here is an example:

```
[root@linux tasl]# cat example.tasl
needle1 = "fox";
needle2 = "the";
needle3 = "nevo";
haystack = "see the brown fox run";
match = strstr(haystack, needle1);
display(match,"\n");
display(strstr(haystack, needle2),"\n");
display("test",strstr(haystack, needle3),"\n");

[root@linux tasl]# ./tasl example.tasl
fox run
the brown fox run
test
[root@linux tasl]#
```

In our first search, the variable “match” has a hit with the content of “fox”. The second search also has a hit with the content “the”. Notice that more of the original string is returned in our second search since the match occurred earlier in our target string. Finally, in the third match, the **`strstr()`** function returns an empty string because the content searched for, “nevo”, is not present.

`ereg(string:<str>, pattern:<regex>, [multiline:<FALSE|TRUE>, icase:<FALSE|TRUE>])`	The **`ereg()`** function tests a string for a regular expression match and returns a “1” if the match succeeds or a “0” if the match fails. Here is an example:
	
	`[root@linux tasl]#` **`cat example.tasl`**
	`string1 = "test string 100";`
	`string2 = "test larry 100";`
	`string3 = "string 100";`
	
	`result1 = ereg(string:string1,pattern:"test .* 100$");`
	`display(result1,"\n");`
	`result2 = ereg(string:string2,pattern:"test .* 100$");`
	`display(result2,"\n");`
	`result3 = ereg(string:string3,pattern:"test .* 100$");`
	`display(result3,"\n");`
	
	`[root@linux tasl]#` **`./tasl example.tasl`**
	`1`
	`1`
	`0`
	`[root@linux tasl]#`
	
	In our example we have three strings or varying content. We test each string against a regular expression statement that asks if the string starts with “test” and ends with “100”. Notice the dollar sign ($) is used to indicate the end of a string, which is a common practice in regular expression syntax. Of the three strings, the first two match our regular expression and return a value of “1” and the last one does not and returns a value of “0”.

`ereg_replace(pattern:<regex>, string:<str>, replace:<str>, [icase:<FALSE|TRUE>])`	The **`ereg_replace()`** function can be used to extract content out of other strings with regular expressions. This allows for a flexible way to extract usernames, IP addresses and other types of content from log files. Below is an example:
	
	`[root@linux tasl]#` **`cat example.tasl`**
	`string1 = "our mac address is AA:BB:EE:FF:GG:HH";`
	`mac1 = ereg_replace(pattern:"our mac address is (.*)$",`
	`string:string1, replace:"\1");`
	`display(mac1,"\n");`
	`string2 = "mac address FF:BB:EE:FF:GG:FF logged in";`
	`mac2 = ereg_replace(pattern:".*address (.*) logged in$",`
	`string:string2, replace:"\1");`
	`display(mac2,"\n");`
	
	`[root@linux tasl]#` **`./tasl ./example.tasl`**
	`AA:BB:EE:FF:GG:HH`
	`FF:BB:EE:FF:GG:FF`
	`[root@linux tasl]#`
	
	This particular example TASL script shows two different fictional messages that deal with Ethernet addresses. In both cases, slightly different regular expressions are used to extract the Ethernet address that is located at different points in the string.

`egrep(string:<str>, pattern:<regex>, [icase:<FALSE|TRUE>])`	This function is similar to the **`ereg()`** function, but is based on the syntax of the Linux **`grep`** statement. The function returns “1” or “0” if the match occurs. Here is an example:

```
[root@linux tasl]# cat example.tasl
string1 = "our mac address is AA:BB:EE:FF:GG:HH";

if (egrep(pattern:"^Server: Dune/0\.([0-5]\.|6\.[0-
7]$)", string:string1) )
     {
     display("There is a match for Dune\n");
     }

if (egrep(pattern:"^our mac address is .*$",
string:string1) )
     {
     display("Had a match for the mac address
message\n");
     }

if (egrep(pattern:"address", string:string1) )
     {
     display("Had a match for the address message\n");
     }
[root@linux tasl]# ./tasl example.tasl
Had a match for the mac address message
Had a match for the address message
[root@linux tasl]#
```

The TASL script does three pattern match attempts. The first test looks for some sort of “Dune” web server banner and has no chance in matching our sample string. The second and third matches though do match. The second match is an example of a more complex regular expression and the third match statement just looks for the occurrence of one word.

eregmatch(string:<str>, pattern:<regex>, [icase:<FALSE|TRUE>])

This function uses regular expressions to find a specific pattern and returns an array of new strings if a match occurs. Here is an example:

```
[root@linux tasl]# cat example.tasl
string = "the username is root";
pattern = "the username is (.*)$";
results = eregmatch(pattern:pattern, string:string);
display(results[0], "\n");
display(results[1], "\n");

string2 = "user jsmith login from 192.168.0.1";
pattern2 = "user (.*) login from (.*)$";
results =  eregmatch(pattern:pattern2, string:string2);
display("username: ",results[1], "\n");
display("address: ",results[2], "\n");
[root@linux tasl]# ./tasl example.tasl
the username is root
root
username: jsmith
address: 192.168.0.1
[root@linux tasl]#
```

In the first example, it can be seen that the “results” array has both an original copy of the string, as well as the matched portion of the regular expression. In the second example, a more complex regular expression is used to pull specific values for a username and a source address from the string.

Please note the use of the trailing dollar sign ($) in both matches. This indicates the end of the string. If we had a longer message or other data after our address, we may have logic like this instead:

```
[root@linux tasl]# cat example.tasl
string2 = "user jsmith login from 192.168.0.1 via SSH";
pattern2 = "user (.*) login from (.*) via";
results = eregmatch(pattern:pattern2, string:string2);
display("username: ",results[1], "\n");
display("address: ",results[2], "\n");
[root@linux tasl]# ./tasl example.tasl
username: jsmith
address: 192.168.0.1
[root@linux tasl]#
```

Notice in this case that since our second desired variable is followed by more text, our regular expression can use the start of that text to anchor itself.

`match(string:<str>, pattern:<regex>, [icase:<FALSE|TRUE>])`

The **`match()`** function evaluates a string with a specific regular expression and returns true or false if the match occurs. The flavor of the match can be case sensitive or in-sensitive. Here is an example below:

```
[root@linux tasl]# cat match-example.tasl
string = "foo BAR ron";
if (match(string: string, pattern: "*bar*", icase:
FALSE))
     {
     display("We have a case sensitive match for
'bar'\n");
     }
if (match(string: string, pattern: "*bar*", icase:
TRUE))
     {
     display("We have a case in-sensitive match for
'bar'\n");
     }
[root@linux tasl]# ./tasl match-example.tasl
We have a case in-sensitive match for 'bar'
[root@linux tasl]#
```

In our example, our target string clearly has its middle word in uppercase. Our first **`match()`** statement looks for the word "bar" and is case sensitive. It does not match. The second **`match()`** statement is exactly the same, but is case insensitive and does match.

Unlike the earlier regular expression examples, the **`match()`** API can be used for simpler pattern matches. It does not support complex "regex" searches, but does support "*" operator.

`substr(string, start, end)`

This function can be used to extract parts of a string when their exact position is known. An example is shown below:

```
[root@linux tasl]# cat example.tasl
```

```
string = "123456789abcdefghi";
offset1 = substr(string,0);
offset2 = substr(string,3);
offset3 = substr(string,3,5);
display(offset1,"\n");
display(offset2,"\n");
display(offset3,"\n");
[root@linux tasl]# ./tasl example.tasl
123456789abcdefghi
456789abcdefghi
456
[root@linux tasl]#
```

Our string is conveniently regular so that we can easily see where the extractions are occurring. Our first **`substr()`** function call asks for a copy of the string, starting from offset zero. This copies the entire string right until the end. The second function call says to go to spot number three and extract that string until the end. The last function call says to go to spot number three, but to only copy until the 5th element of the original string, counting with zero as the first element.

`insstr(string, ins, start, end)`

This function inserts a new string into an existing string. The position of the new string within the original string is controlled by the third argument. The third and fourth arguments together identify the chunk of the original string that is being removed. Here is an example:

```
[root@linux tasl]# more example.tasl
string = "abcdefghijkl";
for (i = 3; i<12; i++)
     {
     result = insstr(string, "AAAA", 3, i);
     display(i," ",result,"\n");
     }
[root@linux tasl]# ./tasl example.tasl
3 abcAAAAefghijkl
4 abcAAAAfghijkl
5 abcAAAAghijkl
6 abcAAAAhijkl
7 abcAAAAijkl
8 abcAAAAjkl
9 abcAAAAkl
10 abcAAAAl
11 abcAAAA
[root@linux tasl]#
```

The example uses the **`insstr()`** function in loop that starts out simply by removing the character “d”, then “de”, then “def” and so on. Each time it is replaced by the string “AAAA”. Eventually, the remainder of the string is completely removed.

`tolower(str)`

This function takes a string and converts any letter to its lower case variant.

```
[root@linux tasl]# cat example.tasl
string = "aBcDeFgHiJkL";
display(tolower(string),"\n");
[root@linux tasl]# ./tasl example.tasl
abcdefghijkl
```

	[root@linux tasl]#
toupper(str)	This function takes a string and converts any letter to its upper case variant. [root@linux tasl]# **cat example.tasl** string = "abcdefghijkl"; display(toupper(string),"\n"); [root@linux tasl]# **./tasl example.tasl** ABCDEFGHIJKL [root@linux tasl]#

crap(length | length:<length>, data:<data>)	This function fills a string variable with a certain length of content. If no content is specified, the letter "X" is used.
	
	[root@linux tasl]# **cat example.tasl**
	string = crap(10);
	display(string,"\n");
	string1 = crap(10,data:'c');
	display(string1,"\n");
	[root@linux tasl]# **./tasl example.tasl**
	XXXXXXXXXX
	cccccccccc
	[root@linux tasl]#

strlen(str)	This function returns the length of a string. Here is an example: [root@linux tasl]# **cat example.tasl** string = "tenable network security"; display(strlen(string),"\n"); [root@linux tasl]# **./tasl example.tasl** 24 [root@linux tasl]#

split(str, [keep:<TRUE|FALSE>, sep:<str>)	This function allows you to create an array of strings, based on splitting up of one string. The example below splits the IP address 192.168.0.1 into each octet by splitting based on the period.
	
	[root@linux tasl]# **cat example.tasl**
	ip = "192.168.0.1";
	display(ip,"\n");
	now = split(ip, sep:".", keep:0);
	display(now[0],"\n");
	display(now[1],"\n");
	display(now[2],"\n");
	display(now[3],"\n");
	[root@linux tasl]# **./tasl example.tasl**
	192.168.0.1
	192
	168
	0
	1
	[root@linux tasl]#

chomp(str)	This function can be used to remove newlines from strings. Below is a slightly convoluted example: [root@linux tasl]# **cat example.tasl**

	`string = "0123456789";`
	`display(string,"\n");`
	`newline = raw_string(10);`
	`display(newline);`
	`string2 = insstr(string,newline,9,9);`
	`display(string2);`
	`string = chomp(string2);`
	`display(string,"\n");`
	`display("here\n");`
	`[root@linux tasl]# ./tasl example.tasl`
	`0123456789`
	
	`012345678`
	`012345678`
	`here`
	`[root@linux tasl]#`
	
	We need to go through some effort to create an example string with a newline in it. First, we create a string containing numbers and a second string containing a newline using the **`raw_string()`** function. We then insert the string with the single newline into the last character of our initial string effectively turning the number nine into our newline. We then display this string and do not use a newline (“\n”) in our **`display()`** statement. Notice that the string prints out as well as the newline. We then use the **`chomp()`** function to remove the newline and print it out. This time, we need to add in a “\n” to our **`display()`** function to get the output to print to a new line. Lastly, we print out a simple “here” message to show that there were not two newlines printed.

`int(str)`	This function returns an integer value based on the contents of the single string argument.
	
	`[root@linux tasl]# more example.tasl`
	`string = "12";`
	`num1 = int(string);`
	`num2 = 5;`
	`num3 = num1 + num2;`
	`display(num3,"\n");`
	`[root@linux tasl]# ./tasl example.tasl`
	`17`
	`[root@linux tasl]#`

`stridx(str, substr)`	This function returns the byte at which a specific substring has been found. Here is a short example:
	
	`[root@linux tasl]# more example.tasl`
	`string = "abcdefghijklmnop";`
	`i = stridx(string,"def");`
	`display(i,"\n");`
	`[root@linux tasl]# ./tasl example.tasl`
	`3`
	`[root@linux tasl]#`
	
	In this case, our search string “def”, starts at character number three, counting from zero.

`str_replace(find:<str>, replace:<str>, string:<str> [,count:<int>])`	This function simply finds a specific sub-string within a string and replaces it with a new sub-string.
	
	`[root@linux tasl]#` **`more example.tasl`**
	`string = "abcdefghijklmnop";`
	`string2 =`
	`str_replace(find:"fgh",replace:"XXoXX",string:string);`
	`display(string2,"\n");`
	
	`[root@linux tasl]#` **`./tasl example.tasl`**
	`abcdeXXoXXijklmnop`
	`[root@linux tasl]#`
	
	Notice that the inserted sub-string can be longer than the “found” sub-string being replaced.

List Related Functions

Function	Description

`make_list(var1 [,var2, ...])`	This function can be used to create an array of strings that can easily be referenced in a script. For example, while conducting log analysis, it may be useful to create a list of keywords and then search a matched log with a loop. Here is a short example of establishing a list:
	
	`[root@linux tasl]#` **`cat example.tasl`**
	`stuff = make_list("this","that","which");`
	`display(stuff[2],"\n");`
	`[root@linux tasl]#` **`./tasl example.tasl`**
	`which`
	`[root@linux tasl]#`

`make_array(var1 [,var2, ...])`	This function allows a TASL script to maintain an array of unique keys and their values. Once an array has been constructed, its elements can be referenced using a key value as an index. If a key value does not exist, a NULL string is returned. Here is an example below:
	
	`[root@linux tasl]#` **`more example.tasl`**
	`stuff = make_array("bill",1, "joe",1, "mike",9, "bob",1,`
	`                   "chuck",5, "matt",2, "joe",1,`
	`"larry",1);`
	`display(stuff["larry"],"\n");`
	`display(stuff["chuck"],"\n");`
	`display(stuff["jim"],"\n");`
	`display(isnull(stuff["jim"]),"\n");`
	`[root@linux tasl]#` **`./tasl example.tasl`**
	`1`
	`5`
	
	`1`
	`[root@linux tasl]#`
	
	This TASL script invokes an array of eight elements. Each has a unique name and a value associated with the name. Displaying the value for “larry”

	is accomplished by referencing the array with the value of “larry”. This prints out the value for “larry”, which is “1”. Similarly, we do this for “chuck” and print out a value of “5”. However, when we reference a key of “jim”, nothing prints out because this key was not instantiated in our array. Furthermore, we can verify this or test for this by using the **`isnull()`** function that evaluates if a string is of zero length. In this case, the string is empty and the function returns a value of “1”.

`keys(<array>)`	This function allows easy walking of an array’s key values. Here is a quick example:
	
	`[root@linux tasl]# ` **`cat example.tasl`**
	`colors =`
	`make_array("red",0,"blue",10,"green",10,"yellow",10);`
	`display(colors["green"],"\n");`
	`display(colors["yellow"],"\n");`
	`new_list = make_list();`
	`foreach i (keys(colors))`
	`     {`
	`     new_list[i] = colors[i];`
	`     display(i," ",new_list[i],"\n");`
	`     }`
	`[root@linux tasl]# ` **`./tasl example.tasl`**
	`10`
	`10`
	`blue 10`
	`green 10`
	`red 0`
	`yellow 10`
	`[root@linux tasl]#`
	
	The script uses the **`keys()`** function to loop through each element of the array and assign it to a new list. It starts out by displaying some of the array values for keys of “green” and “yellow”, both of which are set to “10”. Then the **`make_list()`** function is called to create a new, but empty, list. Finally, the **`keys()`** function is called to return a list of each of the key values for the array. A **`foreach { }`** construct is used to loop through each of the keys returned by the **`keys()`** function, assign the key to a new list and then also print it out.

`max_index(<list>)`	This function counts the number of elements in a list. Here is an example:
	
	`[root@linux tasl]# cat ` **`example.tasl`**
	`colors =`
	`make_list("red",0,"blue",10,"green",10,"yellow",10);`
	`n = max_index(colors);`
	`display(n,"\n");`
	`[root@linux tasl]# ` **`./tasl example.tasl`**
	`8`
	`[root@linux tasl]#`

`sort(<array>)`	The **`sort()`** function performs an alpha-numeric sort of each element in the array. Below is an example:
	
	`[root@linux tasl]# ` **`cat example.tasl`**
	`list = make_list("ron", "bob", "nancy", 100, 200, 10);`
	`list = sort(list);`

	```
	for (i=0;i<max_index(list);i++)
	     {
	     display(list[i],"\n");
	     }
	[root@linux tasl]# ./tasl example.tasl
	10
	100
	200
	bob
	nancy
	ron
	[root@linux tasl]#
	```
	Notice that the list returned places the numbers at the front of the list, and words at the end.

Time Related Functions

Function	Description
`unixtime()`	This function returns the number of seconds that have elasped since January 1, 1970 (the start of time tracking according to the Unix calendar). See the example in **`localtime()`** for an example of the **`unixtime()`** function.
`gettimeofday()`	Similar to the **`unixtime()`** function, but also returns the number of current microseconds. See the example in **`localtime()`** for an example of the **`gettimeofday()`** function.

`localtime([utc:<TRUE|FALSE>])`	This function returns a string that shows the current hour, date, number of minutes and other time tracking elements in a string. Here is an example below:
	```
	[root@linux tasl]# cat example.tasl
	time = unixtime();
	display(time,"\n");
	gtime = gettimeofday();
	display(gtime,"\n");
	ltime = localtime();
	display(ltime,"\n");
	ltime = localtime(utc:TRUE);
	display(ltime,"\n");

	[root@linux tasl]# ./tasl example.tasl
	1119674887
	1119674887.885182
	[ mon: 6, mday: 25, wday: 6, min: 48, sec: 7, yday: 176,
	isdst: 1, year: 2005, hour: 0 ]
	[ mon: 6, mday: 25, wday: 6, min: 48, sec: 7, yday: 176,
	isdst: 0, year: 2005, hour: 4 ]
	[root@linux tasl]#
	```

`mktime(year:<int>, mon:<int>, mday: <int>, hour:<int>, min:<int>, sec:<int> [,isdst:<TRUE|FALSE>])`	This function can be used to create the number of seconds based on a specific combination of year, month, day of the month, hour, minute and second. Here is an example for June 2nd, 2005, at 10:01:55:

	`[root@linux tasl]#` **`cat example.tasl`**
	`newtime = mktime(year:2005, mon:6, mday:2, hour:10, min:1, sec:55);`
	`display(newtime,"\n");`
	`[root@linux tasl]#` **`./tasl example.tasl`**
	
	`1117720915`
	`[root@linux tasl]#`

Local Command Functions

Function	Description

`pread(argv:<list>, cmd:<str>, [cd:<str>, nice:<int>])`	This function can be used to invoke a Linux command and read from its output. All output is read into an array. Below is an example:
	
	`[root@linux tasl]#` **`cat example.tasl`**
	`i = 0;`
	`argv[i++] = "/bin/ls";`
	`argv[i++] = "-lt";`
	`argv[i++] = "/root";`
	`results = pread(cmd:"/bin/ls",argv:argv);`
	`foreach line (split(results))`
	`     {`
	`     line = chomp(line);`
	`     display(line,"\n");`
	`     }`
	`[root@linux tasl]#` **`./tasl example.tasl`**
	`total 36`
	`-rw-r--r--  1 root  root   1229 Oct  7  2004 anaconda-ks.cfg`
	`-rw-r--r--  1 root  root  23471 Oct  7  2004 install.log`
	`-rw-r--r--  1 root  root   3928 Oct  7  2004 install.log.syslog`
	`[root@linux tasl]#`
	
	This script builds up an argv list of "**`/bin/ls -lt /root`**" and invokes it using the **`pread()`** function. Notice that the command executed is in the argument list as well as specified with the "**`cmd:`**" element of the **`pread()`** function. The output is returned as a long string and separated with the **`split()`** function. The **`chomp()`** function is used to string the newline from the output, although we could have simply omitted the "\n" from our **`display()`** function to achieve the same effect.
	
	Keep in mind that the **`lced`** process runs as user "lce". This means that any execution rights need to be available to the "lce" user.

`find_in_path(<str>)`	When invoking Linux commands, this function can be used to test if a desired function is in the default path. In other words, this can be used to see if a program can be executed without specifying an exact path.

	When executing programs, it is more secure to invoke them with a known path.
	
	Here is an example script:
	
	`[root@linux tasl]# cat example.tasl`
	`string = find_in_path("gzip");`
	`display(string,"\n");`
	`string = find_in_path("notexist");`
	`display(string,"\n);`
	
	`[root@linux tasl]# ./tasl example.tasl`
	`1`
	`0`
	`[root@linux tasl]#`
	
	The **`gzip`** program was found in the path and returned a “1” whereas the **`notexist`** program was not found and returned a “0”.

`fread(<path>)`	This function returns the entire contents of a file into an array. Here is an example:
	
	`[root@linux tasl]# cat example.tasl`
	`array = fread("/etc/modules.conf");`
	`foreach line (split(array))`
	`     {`
	`     display(line);`
	`     }`
	
	`[root@linux tasl]# ./tasl example.tasl`
	`alias eth0 pcnet32`
	`alias scsi_hostadapter BusLogic`
	`alias sound-slot-0 es1371`
	`alias usb-controller usb-uhci`
	`[root@linux tasl]#`
	
	This script opens the **`/etc/modules.conf`** file and prints it out line by line.
	
	Keep in mind that the **`lced`** process runs as user “lce”. This means that any file reading rights need to be available to the “lce” user.

`file_write(data:<str>, fp:<path>)`	The **`file_write()`** function is used to store data to disk. Below is an example TASL script that uses the **`file_open()`**, **`file_write()`** and **`file_close()`** functions to store some data to disk.
	
	`[root@linux tasl]# cat example.tasl`
	`string = "see the brown fox" + '\n' + "run to the hill"`
	`+ '\n' + "and sleep." + '\n';`
	
	`fp = file_open(name:"/tmp/example.txt", mode:"w");`
	`file_write(fp:fp, data:string);`
	`file_close(fp);`

```
array = fread("/tmp/example.txt");
foreach line (split(array))
     {
     display(line);
     }
unlink("/tmp/example.txt");

if (file_stat("/tmp/example.txt") == 0)
     {
     display("File was deleted\n");
     }
     else
     {
     display("File was NOT deleted\n");
     }

[root@linux tasl]# ./tasl example.tasl
see the brown fox
run to the hill
and sleep.
File was deleted
[root@linux tasl]#
```

This example starts off by creating an example string that contains new line characters. The string is actually the addition of several substrings, through the use of the "+" operator. Notice that the newlines are added by specifying them inside single quotes. A file handle is then opened for **`/tmp/example.txt`** with write access. The data is written out and the file is closed. To show that the content is indeed there, a **`fread()`** statement is used to print out the contents that were just written.

This example was then further extended to illustrate the use of the **`unlink()`** and **`file_stat()`** functions. Once the file is written, it is then unlinked. After that, a test to see if the file still exists or not is attempted and a display message based on the results is printed out.

Keep in mind that the **`lced`** process runs as user "lce". This means that any file write or delete rights need to be available to the "lce" user.

`unlink(<path>)`	This function deletes the target file. The example shown in the **`file_write()`** section illustrates how to use the **`unlink()`** function.
`file_stat(<path>)`	This function determines if the target file exists or does not exist. The example shown in the **`file_write()`** section illustrates how to use the **`file_stat()`** function.
`file_open(mode:<str>, name:<path>)`	This function opens a target file and returns a file handle. The example shown in the **`file_write()`** section illustrates how to use the **`file_open()`** function.
`file_close(<int>)`	This function closes a target file based on the current file handle associated with that file. The example shown in the **`file_write()`** section illustrates how to use the **`file_close()`** function.

`file_read(fp:<int>, length:<int>)`

This function reads a specified amount of bytes from an open file descriptor. An example is shown below:

```
[root@linux tasl]# cat /tmp/test.txt
1234567890
abcdefghij
.o.o.o.o.o
[root@linux tasl]# cat example.tasl
fp = file_open(name:"/tmp/test.txt", mode:"r");
string = file_read(fp:fp, length:10);
display("read 1: ", string,"\n");
string = file_read(fp:fp, length:10);
display("read 2: ", string,"\n");
string = file_read(fp:fp, length:10);
display("read 3: ", string,"\n");
string = file_read(fp:fp, length:10);
display("read 4: " “,string,"\n");
file_close(fp);

[root@linux tasl]# ./tasl example.tasl
read 1: 1234567890
read 2:
abcdefghi
read 3: j
.o.o.o.o
read 4: .o
[root@linux tasl]#
```

There are several items illustrated in this example. Our **`test.txt`** file contains three lines, each with 10 characters. However, each line also has an 11th character that is a new line. As the TASL script opens the file, it reads in exactly 10 characters and prints them out. Notice on the first read, there are 10 characters printed out. However, on the second read of 10 characters, the first character is the newline, and only the first nine characters of the second line in the **`test.txt`** file are printed out. This continues on until the entire content of **`test.txt`** is read. Also, notice that the last read of 10 characters only returns the remaining data.

The **`file_read()`** function is better suited for reading in binary files or files with specific data structures. Although this example used a small text file, it would be more appropriate to read in the entire contents of the file with **`fread()`** into an array and manipulate the data with string functions.

`file_seek(fp:<int>, offset:<int>, [whence: <str>])`

The **`file_seek()`** function can be used in conjunction with the **`file_read()`** function to grab a specific section of data from a file. The “whence” optional argument can be specified to apply the offset from the beginning (the default, “SEEK_SET”), the current position (“SEEK_CUR”), or the end of the file (“SEEK_END”). Here is an example script:

```
[root@linux tasl]# cat /tmp/test.txt
1234567890abcdefghijklmnopqrstuvwxyz
[root@linux tasl]# cat ./example.tasl
fp = file_open(name:"/tmp/test.txt", mode:"r");
file_seek(fp:fp,offset:10);
string = file_read(fp:fp,length:3);
display(string,"\n");
```

	`file_close(fp);`
	`[root@linux tasl]#` **`./tasl example.tasl`**
	`abc`
	`[root@linux tasl]#`
	
	In this script, we use a test file with the contents of one line of text. The file is opened, the **`file_seek()`** function is used to jump to character 10 and then **`file_read()`** is used to grab just three bytes.

Logging and Alerting Functions

SecurityCenter 4 and the LCE contain built-in alerting functionality that will suffice for most alerting needs. Please refer to the SecurityCenter User Guide and the LCE Administration and User Guide for more information about these capabilities.

Function	Description
`obj.log(src:<long>, dst:<long>, protocol:<int>, sport:<int>, dport:<int>, msg:<string>)`	The **`obj.log()`** function can be used to feed new events back into the LCE database. There are several examples shown throughout this guide. The content of the log message must be parsed by at least one rule in the existing **`.prm`** files in use by the **`lced`** process. Typically, the values for these fields come from the values of the current event. A TASL script may make use of several preceding events, but when logging this particular event, using the current data from it is convenient. This can be confusing in some respects. Consider a brute force **`telnet`** password guessing attempt. There may be several dozen login failures between an attacker's IP and the target IP. All events will probably have port 23 (**`telnet`**) listed in them, but a different ephemeral port for each login failure. However, if our TASL script was alerting on the 12th login failure and chose to log both the source and destination port of this particular login failure, a user may conclude that all brute force attempts occur on those ports. It would be more accurate to log port 23 (**`telnet`**), and leave the other port empty or zero. Here is another example. Consider a rudimentary TASL script that correlates spam relaying, spyware and peer to peer file sharing. Which port should it use? Much of this is left to the TASL script author. Since it is email, one may want to add it on port 25, however if the actual activity is occurring on a non-standard port, it may be wiser to log it there.

Miscellaneous Functions

Function	Description

`rand()`	This function returns a random number. Below is an example script.
	
	`[root@linux tasl]# cat example.tasl`
	`random = rand();`
	`display(random,"\n");`
	`[root@linux tasl]# ./tasl ./example.tasl`
	`311164913`
	`[root@linux tasl]# ./tasl ./example.tasl`
	`1626298773`
	`[root@linux tasl]# ./tasl ./example.tasl`
	`1071398914`
	`[root@linux tasl]#`

`isnull()`	This function returns "0" if the tested string is not empty and returns "1" if the string is empty.

Testing TASL Scripts

If you want to test the syntax of your TASL script before loading it in the LCE, you can use the **`tasl`** command line utility, which is installed under **`/opt/lce`**.

By calling "**`tasl -L script.tasl`**" you will run the command in "lint" mode, which checks the syntax and the function names.

You may execute your script by doing "**`tasl script.tasl`**". However, since TASL is event-driven, only the "**`main`**" part of the script will be executed, and no callback will ever be called. If you want to test a TASL script, write NASL scripts to generate fake SYSLOG messages. Additionally, you may use the **`display()`** function to display messages on screen.

TASL Script Performance

The following sections detail several highly specific examples of TASL scripts. TASL is an extremely efficient language, however it is not designed to process all LCE events as fast as the LCE can receive them.

Pattern Matching

Although TASL can do a variety of regular expression and keyword searches, it is not as efficient as the LCE's plugin language. If you find yourself in the situation of matching on a specific event type, and then inspecting the content of that event to decide if it is of interest, then write a new **`.prm`** plugin and have your TASL events just from that plugin.

For example, let's say a web site is getting 40,000,000 downloads a day, and there is a LCE plugin that says log "Successful Web Download" each time a 200 code is logged by the web server. If a TASL script was then written to take each of these events, but then further search it for something evil such as "credit_card_file.txt", this would be extremely slow. Instead, a **`.prm`** file with a plugin that looked for a 200 code and reference to the file of interest and also output an event name "Successful Credit Card File Download" can be used. The TASL script would then just subscribe to this single event.

Acting on the Least Frequent Event

When testing for multiple events occurring from one host, there is almost no performance cost for a TASL script to subscribe to these events. However, if any logic is invoked, ordering how things are tested can save tremendous amounts of CPU time.

For example, consider events A, B and C. A occurs several million times a day; B occurs a few times an hour and C sometimes does not occur for days. A TASL script that subscribed to all three events and looked for events A, B and C occurring within five minutes of each other against one host is desired. Such a fictional script might start out looking like this:

```
EVENT_A = 10000;
EVENT_B = 10001;
EVENT_C = 10002;
obj = object();
obj.filter.ip.src.add("0.0.0.0/0");
obj.callback.add("myCallback");
obj.filter.event.id.add(EVENT_A, EVENT_B, EVENT_C);

function myCallback(obj)
     {
     # What do we put here?
     }
```

Choosing what sort of logic to place in the actual **`myCallback()`** function will greatly affect our performance. For example, consider this code fragment:

```
cntA =obj.event.count.get(id:EVENT_A,src:obj.event.src(), newer_than:last30);
cntB =obj.event.count.get(id:EVENT_B,src:obj.event.src(), newer_than:last30);
cntC =obj.event.count.get(id:EVENT_C,src:obj.event.src(), newer_than:last30);
if ( (cntA != 0) && (cntB != 0) && (cntC !=0) )
     {
     # send an alert !!
     }
```

The logic is sound in that we get the correct count of the offending events. However, this code fragment would get invoked several million times a day. Keep in mind event A occurs very often.

Since we are only really interested when event A, event B and event C occur, and C is the least frequent event, consider this code fragment:

```
if (obj.event.id() == EVENT_C)
     {
     cntA =obj.event.count.get(id:EVENT_A,
            src:obj.event.src(),    newer_than:last30);
     cntB =obj.event.count.get(id:EVENT_B,
            src:obj.event.src(), newer_than:last30);
     cntC =obj.event.count.get(id:EVENT_C,
            src:obj.event.src(),    newer_than:last30);
     if ( (cntA != 0) && (cntB != 0) && (cntC !=0) )
          {
          # send an alert !!
          }
     }
```

This code is much more efficient because it is hardly executed by the LCE at all. For each of the millions of event "A" types that occur each day, all the TASL has to do is add it to its current list of events and compare it once to see if it was event "C".

To increase performance, we could reduce the above script further by removing the count for event "C" because we know there is at least one, and we are just testing for a non-zero condition.

Use Keys and Hashes, not Loops

The LCE’s TASL scripts can get invoked when particular events occur. If a TASL script requires performing some sort of analysis within a loop, consider redesigning the algorithm to use a hash table. For high event rates and longer loops, an exponential impact on performance can occur.

For example, consider a TASL script that subscribed to “VPN User Login” events and for each event, extracted the username that logged in, compared it to a list of existing users and generated an alert if it was a new username. If the search of the existing list of users was done linearly with a **`for()`** loop this search can be very inefficient, especially if there is an increase in the rate of events or an increase in the number of users. Instead, through the use of keys, the existence of a particular user can be accomplished in one lookup.

Without having all the logic of a complete TASL script, consider this code fragment:

```
if ( GlobalVPNUsers[username] != 1 )
     {
     # code to handle occurrence of a new "username"
     }
```

If we had a script that read in our list of valid users from a file and placed that into a hash named “GlobalVPNUsers”, lookups for potentially new names are then done in one step. When a new user was found, we may want to alert that an invalid user tried to log in. We may also want the script to dynamically add the name to our list or even persist this information to disk.

Filter Out the Bad Guys

Once we know that a certain IP address is potentially a worm source, a spam source, or otherwise a source of large numbers of events, we may want to temporarily remove it from event processing. Consider this following code fragment:

```
obj.filter.ip.src.remove.until(ip:obj.event.src(), until:unixtime () + 24*3600);
```

If this code were placed just after an event was logged to the LCE database, the particular TASL script would not receive events from the particular IP address for 24 hours.

Example 1: Snort IDS Worm Detect

Introduction

This is an example of simple event consolidation. It is observed that when three Snort IDS events occur within 30 seconds of each other from hosts that are infected with a particular SQL worm. These are the Snort IDS event names:

- MS-SQL version overflow attempt
- MS-SQL Worm propagation attempt
- MS-SQL Worm propagation attempt OUTBOUND

In the LCE, these alerts show up as normalized “Snort-UDP_Event” names. What we would like to do is generate a new alert anytime a host sends us these three events within 30 seconds.

TASL Script

Here is a TASL script that looks for more than three Snort-UDP_Event events from one host in less than 30 seconds:

```
function myCallback(obj)
     {
     local_var count, last30;
     last30 = unixtime() - 30;
     count = obj.event.count.get(id:SNORT_UDP_EVENT,
                                 src:obj.event.src(),
                                 newer_than:last30);
     if ( count >= 3 )
          {
          obj.log(src:obj.event.src(),
                  dst:obj.event.dst(),
                  protocol:IPPROTO_UDP,
                  sport:obj.event.sport(),
                  dport:obj.event.dport(),
                  msg:'Worm_Probe - Multiple SQL Worm Probe –
                  Snort Detect');
   # Don't ever report this host again for 24 hours
          obj.filter.ip.src.remove.until(ip:obj.event.src(),
                                    until:unixtime () + 24*3600);
          }
     }

 # This is our .main.
 SNORT_UDP_EVENT = 5102;    #  Snort-UDP_Event
 obj = object();
 obj.filter.ip.src.add("0.0.0.0/0");
 obj.callback.add("myCallback");
 obj.filter.event.id.add(SNORT_UDP_EVENT);
```

This script has been used as an example through this document several times. Below is an example LCE **`.prm`** plugin that can be used to add new events back into the LCE.

```
  id=20002
  name=SQL Ping Snort Detect
  match=Worm_Probe - Multiple SQL Worm Probe - Snort Detect
  log=event:Worm_Probe type:correlated
```

Note that the "type" logged by the plugin is named "correlated". All example TASL scripts from Tenable use this for TASL scripts.

Comments

This script could be further extended several ways. We originally wanted to alert on specific Snort "MS-SQL" events, yet this script is alerting on generic Snort UDP events. Without writing new LCE **`.prm`** rules, one could consider extending the TASL script to only filter on ports that were related to SQL. In addition, new LCE plugins could be written that generated specific "Snort SQL Events".

Example 2: Dragon IDS Scan & Compromise

Introduction

This example will make use of events from the Dragon IDS. In this scenario, there is some sort of worm that is sending UDP packets across the network. When the worm hits a vulnerable host, we get a few Dragon "Compromise" events. Dragon specifically reports thousands of events named "ICMP:SUPERSCAN" followed by a few "COMP:WIN-2000" and "MS-BACKDOOR3" events. When the LCE receives these events, it normalized them to the following names:

- Dragon-ICMP_Event
- Dragon-Compromise_Event

The behavior that we are looking for is an IP address that has performed a large number of scans and as a result has generated a large number of “Dragon-ICMP_Event” instances and then has also encountered the Dragon “Compromise_Event” logs. We want to alert when we see a large scan followed by compromise events.

TASL Script

The logic will be to subscribe to Dragon Compromise and Dragon ICMP events. If we get a Dragon Compromise event, then we will look for the same IP doing Dragon ICMP events. Also keep in mind that when we have a Dragon Compromise event, we have a source and a destination IP.

We want to make sure we look for Dragon ICMP events with source IP addresses that match the destination address of the Dragon Compromise events. Although not shown, the source IP address of the Dragon Compromise events is actually the attacked host and not the attacker.

Many NIDS that look at response traffic for a detection will flip the expected source and destination IP addresses.

Here is the example script:

```
function myCallback(obj)
     {
     local_var count, last30;
     last30 = unixtime() - 30;      # 30 seconds

     if (obj.event.id() == DRAGON_COMP_EVENT)
          {
          count = obj.event.count.get(id:DRAGON_ICMP_EVENT,
                                      src:obj.event.dst(),
                                      newer_than:last30);

          if (count >= 3)
               {
               obj.log(src:obj.event.src(),
                       dst:obj.event.dst(),
                       protocol:obj.event.protocol(),
                       sport:obj.event.sport(),
                       dport:obj.event.dport(),
                       msg:'Compromise_Activity - UDP scan  followed by compromise
      events - Dragon Detect');
                 obj.filter.ip.src.remove.until(ip:obj.event.dst(),
                               until:unixtime () + 24*3600);
               }
          }
     }

 # This is our .main.
 DRAGON_COMP_EVENT = 5006;   #  Dragon-Compromise_Event
 DRAGON_ICMP_EVENT = 5005;   #  Dragon-ICMP_Event
 obj = object();
 obj.filter.ip.src.add("0.0.0.0/0");
 obj.callback.add("myCallback");
```

```
  obj.filter.event.id.add(DRAGON_COMP_EVENT, DRAGON_ICMP_EVENT);
```

Notice that we are putting the word “Compromise_Activity” in the front of the message generated by this alert. We will also use this to name the event in the corresponding **.prm** file.

Here is the corresponding **.prm** plugin:

```
id=20004
name=Dragon worm UDP activity
match=Compromise_Activity - UDP scan followed by compromise
 events - Dragon Detect
log=event:Compromise_Activity type:correlated
```

Comments

This script could have implemented its logic by comparing the current event to the Dragon ICMP event type. Although logically sound, this would have been a slower performing TASL script. The Dragon Compromise event was chosen because it was not occurring as often as the Dragon ICMP event.

Example 3: Uninvited Wireless Guest with Apple AirPort

Introduction

Here is a very simple script that demonstrates how a TASL script can keep state of some items through the use of a global variable and persisting data to disk.

The goal of this script is to send an email message every time someone “different” attempts to log into a wireless network (which is protected by a WPA key, so hopefully there will be a lot of attempts but no successes). We do not want to receive an email every time the same person attempts to log in the network, just when a different workstation is passing by.

This script is equivalent to the Passive Vulnerability Scanner's “new host” functionality, except that it works on MAC addresses sent by an access point in SYSLOG format.

Note that you cannot use the IP address of remote wireless users, but can use their Ethernet address. This can be spoofed, and a sophisticated wireless abuser may attempt to use a valid or previously seen Ethernet address.

Since TASL objects do not offer any facility to store any kind of data, we will use a global variable to store each MAC address we have seen, and send an alert every time a new MAC address is passing by.

For reference, a SYSLOG message from an Apple AirPort access point looks like this:

```
Jun 18 23:37:30 foobar dot11: Associated with station 00:0a:95:f2:6d:61
```

It is recognized and parsed by plugin number 3501 in the **accespoint_airport.prm** library.

Basic Script

The TASL script will:

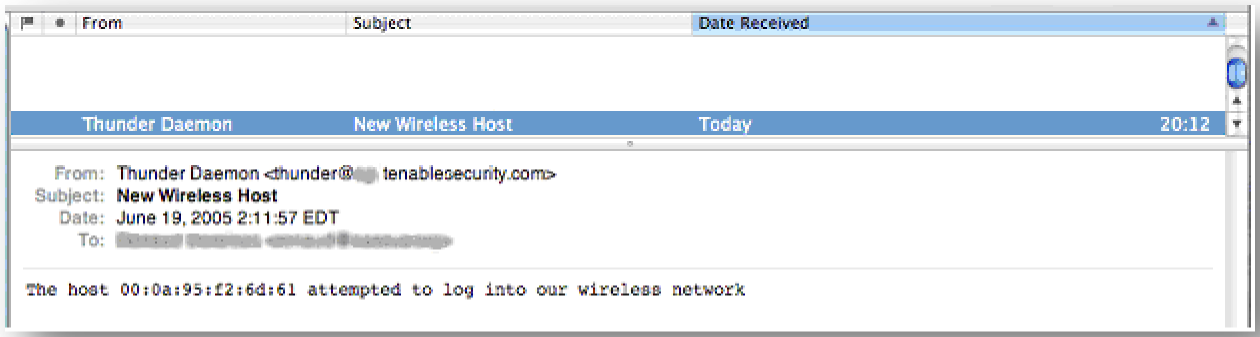
- Subscribe to LCE **.prm** #3501 (in **accesspoint_airport.prm**)
- Extract the MAC address from the message (using the **ereg_replace()** function)

- Use the MAC address as an index to a hash table
- Look up the hash table to see if we flagged this address already
- Alert if the observed MAC address is a “new” address

The script looks like this:

```
# First we define a global variable which is an empty list
GlobalMacArray = make_list();

# Then we initialize our object and subscribe to event
# 3501

obj = object();
obj.callback.add("myCallback");
obj.filter.event.id.add(3501);

# Now we define the callback:

function myCallback(obj)
{
    local_var mac_addr;

    # Extract the MAC address
    # <133>Jun 18 23:37:30 foobar dot11: Associated with station 00:0a:95:f2:6d:61

    mac_addr = ereg_replace(pattern:".*with station (.*)$", string:obj.event.data(),
       replace:"\1");

    # In the example above, mac_address is now equal to
    # "00:0a:95:f2:6d:61.
    # We look at GlobalMacArray["00:0a:95:f2:6d:61"] if a value of 1
    # is set.
    if ( GlobalMacArray[mac_addr] != 1 )
    {
        # Set the index to 1
        GlobalMacArray[mac_addr] = 1;

        # Create a mail script file - redirection and piping
        # not allowed in pread.
        # We will re-use our msmtp.conf, bundled with LCE.
        # !!!Be sure this is configured!!!
        mail_data = "echo " + '"' + "Subject: New Wireless Host Detected"
                    + '\n'
                    + "The host " + mac_addr + " attempted to log into our wireless
       network." + '\n' +
                 "Original log data:" + '\n' + '\n' +
                 obj.event.data() + '"' +
                 " | /opt/lce/tools/msmtp -C /opt/lce/tools/msmtp.conf
       tester@nessus.org";

        fp = file_open(name:"/tmp/exampletasl.sh",mode:"w");
        file_write(fp:fp, data:mail_data);
        file_close(fp);

        # Then send the email by invoking the script we wrote above
        argv[0] = "sh";
        argv[1] = "/tmp/exampletasl.sh";
        r = pread(cmd:"sh",argv:argv);

        # Remove the mail file
        unlink("/tmp/exampletasl.sh");
    }
}
```

Load the script and wait for the flow of incoming emails:



Thunder Daemon   New Wireless Host   Today   20:12

From: Thunder Daemon <thunder@[redacted]tenablesecurity.com>
Subject: New Wireless Host
Date: June 19, 2005 2:11:57 EDT
To: [redacted]

```
The host 00:0a:95:f2:6d:61 attempted to log into our wireless network
```

Saving State on Disk

The script above works fine, however every time the end-user will shut down the LCE and restart it, our state table will be flushed so “old” MAC addresses will show up again. This is annoying, inaccurate and can be compensated for by saving state to disk.

To solve this issue, we can use the **`file_open()`**, **`file_write()`**, and **`fread()`** functions to write down each new MAC address to disk, and reload all of them at startup.

Our script now looks like this:

```
# First we define a global variable which is an empty list
GlobalMacArray = make_list();

# If the file "mac_addrs.txt" exists, read it
if ( file_stat("/opt/lce/db/mac_addrs.txt") > 0 )
{
    # fread() reads the file entirely and returns its content in a
    # buffer. We call split() to turn this buffer into an array, and
    # finally we read each element of the array using
    # the foreach() iterator to fill our global hash table
    all_mac_addr = split(fread("/opt/lce/db/mac_addrs.txt"), sep:'\n', keep:FALSE);
    foreach mac (all_mac_addr)
    GlobalMacArray[mac] = 1;
}

# Then we initialize our object and subscribe to event
# 3501
```

```
obj = object();
obj.callback.add("myCallback");
obj.filter.event.id.add(3501);

# Now we define the callback:

function myCallback(obj)
{
    local_var mac_addr;

    # Extract the MAC address
    # <133>Jun 18 23:37:30 foobar dot11: Associated with station 00:0a:95:f2:6d:61

    mac_addr = ereg_replace(pattern:".*with station (.*)$", string:obj.event.data(),
       replace:"\1");

    # In the example above, mac_address is now equal to
    # "00:0a:95:f2:6d:61.
    # We look at GlobalMacArray["00:0a:95:f2:6d:61"] if a value of 1
    # is set.
    if ( GlobalMacArray[mac_addr] != 1 )
    {
        # Set the index to 1
        GlobalMacArray[mac_addr] = 1;

        # Write down the mac addr to disk
        fp = file_open(name:"/opt/lce/db/mac_addrs.txt", mode:"a");
        file_write(fp:fp, data:mac_addr + '\n');
        file_close(fp);

        # Create a mail script file - redirection and piping
        # not allowed in pread.
        # We will re-use our msmtp.conf, bundled with LCE.
        # !!!Be sure this is configured!!!
        mail_data = "echo " + '"' + "Subject: New Wireless Host Detected"
                    + '\n' +
                "The host " + mac_addr + " attempted to log into our wireless
       network." + '\n' +
                "Original log data:" + '\n' + '\n' +
                obj.event.data() + '"' +
                " | /opt/lce/tools/msmtp -C /opt/lce/tools/msmtp.conf
       tester@nessus.org";

        fp = file_open(name:"/tmp/exampletasl.sh",mode:"w");
        file_write(fp:fp, data:mail_data);
        file_close(fp);

        # Then send the email by invoking the script we wrote above
        argv[0] = "sh";
        argv[1] = "/tmp/exampletasl.sh";
        r = pread(cmd:"sh",argv:argv);

        # Remove the mail file
        unlink("/tmp/exampletasl.sh");
    }
}
```

Next time we restart **`lced`**, we will not be alerted on hosts we have already seen.

For More Information

Tenable has produced a variety of documents detailing the LCE's deployment, configuration, user operation, and overall testing. These documents are listed here:

- Log Correlation Engine 4.2 Architecture Guide – provides a high-level view of LCE architecture and supported platforms/environments.
- Log Correlation Engine 4.4 Administrator and User Guide – describes installation, configuration, and operation of the LCE.
- Log Correlation Engine 4.4 Quick Start Guide – provides basic instructions to quickly install and configure an LCE server. A more detailed description of configuration and management of an LCE server is provided in the "LCE Administration and User Guide" document.
- Log Correlation Engine 4.4 Client Guide – how to configure, operate, and manage the various Linux, Unix, Windows, NetFlow, OPSEC, and other clients.
- LCE 4.4 High Availability Large Scale Deployment Guide – details various configuration methods, architecture examples, and hardware specifications for performance and high availability of large scale deployments of Tenable's Log Correlation Engine (LCE).
- LCE Best Practices – Learn how to best leverage the Log Correlation Engine in your enterprise.
- Tenable Event Correlation – outlines various methods of event correlation provided by Tenable products and describes the type of information leveraged by the correlation, and how this can be used to monitor security and compliance on enterprise networks.
- Tenable Products Plugin Families – provides a description and summary of the plugin families for Nessus, Log Correlation Engine, and the Passive Vulnerability Scanner.
- Log Correlation Engine Log Normalization Guide – explanation of the LCE's log parsing syntax with extensive examples of log parsing and manipulating the LCE's **`.prm`** libraries.
- Log Correlation Engine TASL Reference Guide – explanation of the Tenable Application Scripting Language with extensive examples of a variety of correlation rules.
- Log Correlation Engine 4.0 Statistics Daemon Guide – configuration, operation, and theory of the LCE's statistic daemon used to discover behavioral anomalies.
- Log Correlation Engine 3.6 Large Disk Array Install Guide – configuration, operation, and theory for using the LCE in large disk array environments.
- Example Custom LCE Log Parsing - Minecraft Server Logs – describes how to create a custom log parser using Minecraft as an example.

Documentation is also available for Nessus, the Passive Vulnerability Scanner, and SecurityCenter through the Tenable Support Portal located at https://support.tenable.com/.

There are also some relevant postings at Tenable's blog located at http://www.tenable.com/blog and at the Tenable Discussion Forums located at https://discussions.nessus.org/community/lce.

For further information, please contact Tenable at support@tenable.com, sales@tenable.com, or visit our web site at http://www.tenable.com/.

About Tenable Network Security

Tenable Network Security provides continuous network monitoring to identify vulnerabilities, reduce risk, and ensure compliance. Our family of products includes SecurityCenter Continuous View™, which provides the most comprehensive and integrated view of network health, and Nessus®, the global standard in detecting and assessing network data.

Tenable is relied upon by more than 24,000 organizations, including the entire U.S. Department of Defense and many of the world's largest companies and governments. We offer customers peace of mind thanks to the largest install base, the best expertise, and the ability to identify their biggest threats and enable them to respond quickly.

For more information, please visit tenable.com.