NwConsole's "sdk content" command

NwConsole is like a NetWitness swiss army knife, there's all kinds of tools buried underneath it's command line interface. NwConsole is multi-platform; executables are available for CentOS (it already ships on appliances), Windows and Mac. One of the really powerful tools is the command "sdk content". It's literally brimming with a crazy ton of options to do just about anything, at least as far as extracting content from the SA Core stack. You can use it to create pcap files, log files or extract files out of network sessions (e.g., grab all the pictures from email sessions). The files can be appended, have a max size assigned before a new file is created, the files can be automatically cleaned up when the directory grows too large. It can run queries in the background to find new sessions. It will break queries up into manageable groups and perform those operations automatically. When the group is exhausted, it will requery to get a fresh set of data to operate on. The list goes on and on.

Because the command has so many options, it's probably best to just to give examples of different commands and what they do. So this article documents recipes for different use cases.

First off, some NwConsole basics:

- To run a set of commands from a file: **NwConsole -f /tmp/somefile.script**
- To pass in a list of commands from the command line: **NwConsole -c <command1> -c <command2> -c <command3>**
    - This isn't necessarily recommended except for very simple scripts. The bash interpreter can make mincemeat out of quoted strings if you don't escape properly. If you are having non-obvious errors passing via command line, switch over to reading from a file to see if that fixes the issues.
- And of course, you can just run **NwConsole** and type the commands in the console window.

Now, before you can run **sdk content**, there are a few commands (like logging into a service) that you need to run first. Here are some examples:

- First connect to a service: **sdk open nw://admin:netwitness@10.10.25.50:50005**
- If you need to connect over SSL, use the nws protocol: **sdk open nws://admin:netwitness@10.10.25.50:56005**
- Keep in mind that you are passing a URL and must URL encode it properly. So if the password is "p@ssword", the URL would look like this: **sdk open nw://admin:p%40ssword@10.10.25.50:50005**
    - This applies to username as well
- Once you are logged in, you can set an output directory for the commands: **sdk output <some pathname>**
- For command line help, just type: **sdk content**

Before we get into some example commands, let's cover the **sessions** parameter first. This parameter is very important and controls how much or how little data you want to grab (the where clause is also important). The **sessions** parameter is either a single session id or a range of session ids. All SA Core services work with session ids, which start at 1 and increase 1 up for every new session added to the service (network or log session). Session ids are 64 bit integers, so they can get quite large. To keep it simple, let's assume we have a Log Decoder that has ingested 1000 logs and parsed them. On the service, you now have 1000 sessions with session ids from 1 to 1000 (session id 0 is never valid). If you wanted to operate over all 1000 sessions, you would pass **sessions=1-1000**. Makes sense right? If you only wanted to operate over the last 100 sessions, you would pass **sessions=901-1000**. Once the command finishes processing session 1000, it will exit back to the console prompt.

Many times however, we don't care about specific session ranges. We just want to run a query over all of them and process the sessions that match a query. There are some shortcuts that simplify this:

- The letter **l** (lowercase L) means lower bound or the lowest session id
- The letter **u** means the highest session id. As a matter of fact, it actually means the highest session id for future sessions as well. In other words, if you pass **sessions=l-u**, this special range means operate over all the current sessions in the system, but also don't quit processing and as new sessions enter the system, process those too. The command will pause and wait for new sessions once it reaches the last session on the service. In a nutshell, the command never exits and goes into "continuous processing" mode. It will run for days, months or years - unless killed.
- If you don't want the command to run forever, you can pass **now** for the upper limit. This will determine the last session id on the service at the time the command is started and process all sessions until it reaches that session id. Once it reaches that session id, the command will exit, regardless of how many sessions may have been added to the service since the command started. So, for our example Log Decoder, **sessions=200-now** would start processing at session 200 and go all the way to session 1000 and quit. Even if another 1000 logs were added to the Log Decoder after the command started, it still exits after processing session 1000.
- The parameter **sessions=now-u** means start at the very last session and continue processing all new sessions that come in. It will not process any existing sessions (except the last one), only new sessions.

Recipes

First, let's start with something simple. I'm going to show all the commands once, then just show the sdk content commands. In this first recipe, we are going to create a log file and grab the first 1000 logs out of a Concentrator aggregating from a Log Decoder.

```
sdk open nw:://admin:netwitness@myconcentrator.local:50005

sdk output /tmp
```

```
sdk content sessions=1-1000 render=logs append=mylogs.log fileExt=.log
```

This script will output 1000 logs (assuming sessions 1 thru 1000 exist on the service) to the file /tmp/mylogs.log. The logs will be written in a plain text format. The parameter **fileExt=.log** is necessary to indicate to the command that we want to output a log file.

```
sdk content sessions=1-1000 render=logs append=mylogs.log fileExt=.log includeHeaders=true
separator=","
```

In this command, it will grab the same 1000 logs as above, but will parse the log header and extract the log timestamp, forwarder and other information and put them in a CSV formatted file

Example CSV: `1422401778,10.250.142.64,10.25.50.66,hop04b-LC1,%MSISA-4: 81.136.243.248...`

The timestamp is in Epoch time. The *includeHeaders* and *separator* parameters can only be used on installs 10.4.0.2 and later.

```
sdk content sessions=l-now render=logs append=mylogs.log fileExt=.log includeHeaders=true
separator="," where="risk.info='nw35120'"
```

This command will write a log file across the current session range, but only logs that match **risk.info='nw35120'**. Keep in mind that when you add a where clause, it performs a query in the background to gather the session ids for export. The query should be run on a service with the proper fields indexed (which is typically a Broker or Concentrator). In this case, since you are querying the field *risk.info*, you will need to double-check the service you run the command on to make sure it's indexed at the value level (IndexValues, see index-concentrator.xml for examples). By default, most Decoders only have *time* indexed. If you use any field but *time* in the where clause, you need to move the query from the Decoder to a Concentrator/Broker/Archiver with the proper index levels for the query. More information on indexing and writing queries can be found in the Core Database Tuning Guide.

```
sdk content sessions=l-now render=logs append=mylogs.log fileExt=.log includeHeaders=true
separator="," where="threat.category exists && time='2015-01-05 15:00:00'-'2015-01-05 16:00:00'"
```

Same as above but will only search for matching logs between 3pm and 4pm (UTC) on Jan 5, 2015 that have a meta key *threat.category*. Again, because this query has a field other than time in the where clause (*threat.category)*, it should be run on a service with *threat.category* indexed at least at the *IndexKeys* level (the operators *exists* and *!exists* only require an index at the key level, although values works fine too).

```
sdk content sessions=l-now render=logs append=mylogs fileExt=.log where="event.source begins
'microsoft'" maxFileSize=1gb
```

This command will create multiple log files, each one no larger than 1 GiB in size. The filenames will be prepended with "mylogs" and appended with the date-time of the first packet/log timestamp in the file. Some example filenames: "`mylogs-1-2015-Jan-28T11_08_14.log`", "`mylogs-2-2015-Jan-28T11_40_08.log`" and "`mylogs-3-2015-Jan-28T12_05_47.log`". On versions older than 10.5, the T separator between date and time will be a space.

```
sdk content sessions=l-now render=pcap append=mypackets where="service=80,21 && time='2015-01-28
10:00:00'-'2015-01-28 15:00:00'" splitMinutes=5 fileExt=.pcap
```

Grab all packets in between the 5 hour time period for service types 80 and 21 and write a pcap file. Every 5 minutes, start a new pcap file.

```
sdk content time1="2015-01-28 14:00:00" time2="2015-01-28 14:15:00" render=pcap append=mydecoder
fileExt=.pcap maxFileSize=512mb sessions=l-now
```

Pay attention to this command. Why? It works for both packets and logs and is *extremely fast*. The downside is you get everything between the two time ranges and you can't use a where clause. But again, it starts streaming everything back almost immediately and does not require a query to run first on the backend. Because everything is read using sequential I/O, it can completely saturate the network link between the server and client. It will start creating files prepended with "mydecoder" and split to a new file once it reaches 512 MiBs in size.

```
sdk tailLogs
```

or (the equivalent command):

sdk content render=pcap console=true sessions=now-u

This is a fun little command. It actually uses "sdk content" behind the scenes. The whole point of this command is to spit out all incoming logs on a Log Decoder. That's it. Real simple. As logs come into the Log Decoder (you can run it on a broker or concentrator too), they will be output on

the console screen. It's a great way to see if the LD is capturing and what exactly is coming in. This runs in continuous mode. This should not be used if the LD is capturing at a high ingest rate (there's no way this command can keep up). But it's great for verification or troubleshooting purposes.

```
sdk tailLogs where="device.id='ciscoasa'" pathname=/mydir/anotherdir/mylogs
```

Same as above except it will only output logs that match the where clause and instead of outputting to the console it will write them to a set of log files under /mydir/anotherdir that do not grow larger than 1 GiB. Obviously this can be accomplished with the "sdk content" command as well, but it's a little less typing with this command if you like the default behavior.

```
sdk content sessions=now-u render=pcap where="service=80" append=web-traffic fileExt=.pcap
maxFileSize=2gb maxDirSize=100gb
```

Start writing pcaps of all web traffic from the most recent session and all new incoming sessions that match service=80. Write out pcaps no larger than 2 GiBs and if all the pcaps in the directory grow larger than 100 GiBs, then delete the oldest pcaps till the directory is 10% smaller than the max size. Keep in mind that the directory size checking isn't exact and only checks every 15 minutes by default. You can adjust the number of minutes between checks by passing *cacheMinutes* as a parameter, but this only works with 10.5 and later.

```
sdk content sessions=79000-79999 render=nwd append=content-%1%.nwd metaFormatFilename=did
```

This is a poor man's backup command. It will grab 1000 sessions and output the full content (sessions, meta, packets or logs) to the NWD (NetW itness Data Format) format. NWD is special format that can be re-imported to a Packet/Log Decoder without reparsing. So essentially, the original parsed session is imported without change. The timestamp will not change as well, so if it was originally parsed 6 months ago, the timestamp upon import will be retained as 6 months ago. NOTE: Do not expect great performance with this command, especially with packets. Gathering the packets for a session involves a lot of random I/O and can drastically slow down the export. Logs do not suffer as much from this problem (only one log per session), but behind the scenes this command uses the /sdk content API and this is not a performance minded streaming API like /sdk packets. So again, don't expect great performance. One neat trick with this command is the *metaFormatFilename* param eter. If this is run on a concentrator with more than one device, the NWD filenames will be created with the "did" meta for each session (the *%1%* in the *append* parameter is substituted with the value of "did"). Essentially, this means each filename will indicate exactly which Decoder the data came from.

```
sdk content session=l-u where="service=80,139,25,110" render=files maxDirSize=200mb cacheMinutes=10
```

Another fun little command. Works very similiar to our old Visualize product if you pair the output directory with something like Windows Explorer in Icon mode. It will extract files from all web, email and SMB traffic. This includes all kinds of crazy stuff: images, zip files, videos, PDF's, office documents, text files, executables, audio files, etc. If it extracts malware, your virus scanner will flag it. Don't worry, nothing will be executed by the command, so it's not like it can infect the machine (don't be silly and execute it yourself). But it can be useful because if you do find malware, the filename indicates the session id where it was extracted. You can then query that session id and see what host the malware possibly infected and take action. You can filter what gets extracted with the parameters *includeFileTypes* or *excludeFileTypes* (see the command help). For instance, adding **excludeFileTypes=".exe;.dmg;.msi"** will prevent executables and installers from being extracted. This command will just run nonstop extracting files from all existing and any new sessions. After the directory gets littered with more than 200 MiBs of files, it will automatically starting cleaning up the files every 10 minutes. NOTE: This command only makes sense for packet sessions, not logs.

```
sdk content session=1-now where="time='2015-01-27 12:00:00'-'2015-01-27 13:00:00' &&
(service=25,110,80)" subdirFileTypes="audio=.wav;.mp3;.aac; video=.wmv;.flv;.mp4;.mpg;.swf;
documents=.doc;.xls;.pdf;.txt;.htm;.html images=.png;.gif;.jpg;.jpeg;.bmp;.tif;.tiff
archive=.zip;.rar; other=*" renameFileTypes=".download|.octet-stream|.program|.exe;.jpeg|.jpg"
render=files maxDirSize=500mb
```

In this example, we are extracting files from HTTP and email sessions from a one hour period and then grouping the extracted files into directories specified by the *subdirFileTypes* parameter. For instance, any extracted audio file with the extension .wav, .mp3 or .aac will be placed into the subdirectory audio, which will be created under the specified output directory. Same goes for all the other groups specified in that parameter. Some files will also be automatically renamed based on their file extension. This is handled by *renameFileTypes*. Any file with an extension .download, .octet-stream or .program will be renamed to .exe. Files with the extension .jpeg will be renamed .jpg. Once the toplevel directory exceeds 500 MiBs, the oldest files will get cleaned up. This command will stop at the last session at the time the command was started.

```
sdk search session=l-now where="service=80,25,110" search="keyword='party' sp ci"
```

This command will search all packets and logs (the *sp* parameter) for the keyword "party". If party is found anywhere in the packets or logs, it will output the session id along with the text it found and the surrounding text for context. The where clause indicates it will only search web and email traffic. The *ci* parameter means case insensitive search. You can substitute "regex" for "keyword" and it will perform a regex search.

```
sdk search session=l-now search="keyword='checkpoint' sp ci" render=log append=checkpoint-logs.log
fileExt=.log
```

Now this gets interesting. Search all logs (or it could be packets) for the keyword "checkpoint" and if that keyword is seen, extract the log to a file "checkpoint-logs.log". There are all kinds of possibilities with this command. Essentially what it does is when a hit is detected, it hands off the session to the content call. So any parameters you pass to *sdk search* that it doesn't recognize, it will just pass along to the content call. This allows the full capabilities of the *sdk content* call, but only working on those sessions with content search hits. With great power comes great responsiblity!