NETWITNESS® SDK DOCUMENTATION

Version 9

Table of Contents

Introduction

The NetWitness® SDK consists of a "C" API that allows for read only access to query, search, and render NetWitness® local and remote collections. The API is available for both Windows and Linux operating systems.

All functions return TRUE (1) on success and FALSE (0) on failure. If a function fails, the calling thread can use NwLastError to retrieve the last error message. All functions work against a unique handle which is provided by the NwOpen function and maps to a specific collection. The majority of functions operate against object id ranges and return results in an array of NwFields.

The NwField object contains a string based name and a variant value. The variant can be either a number or a string value.

SDK Syntax

Many of the functions in the SDK use a SQL like language string including NwValues, NwQuery, and NwSearch. The syntax of each are as follows:

Select Clause

The select clause is a comma seperated list of language field names.

Syntax: select <field1>,<field2>,etc…

Example select clause
"select sessionid,time,service"

The above example would select three fields from the database—*sessionid, time, and service.*

Where Clause

The where clause is a comma seperated list of language field values and ranges and is used by NwValues function.

Syntax: where <field1 [field operator] value1,value2,value3-value4> <logic operator> <field2> etc…
Fields: See appendex
Field operator: See below (=, !=, exists, !exists, contains, regex, length, begins, ends)
Values: See appendex for Variant types, plus l (lower bound) and u (upper bound)
Ranges: x-y (dash between two values creates inclusive range)
Logical Operations: || (or) , && (and)
Precedence operators: (,)

Example where clause

"where service=25,80,60000-u && (filename exists || username exists)"

The above example would select service values 25, 80, and 60000 and above where either a filename or username exists. You can specify *l* or *u* within a range for the lower and upper values of a range. Multiple conditions can be specified using logic operators *&&* and ||. Precedence operators *(* and *)* can be used as well.

Numeric and string values support the following operators:

Field Operator	**Explanation**
=	Equals, returns results where the field is equal to any provided value. For example, "tcp.dstport = 21-25,110" will return sessions having TCP destination port of 21, 22, 23, 24, 25, or 110.
!=	Not equal, returns results for field that do not match the values specified. For example, "eth.type != 0x0800" would only return sessions outside of hex value 0x0800 (decimal value of 2048) which would be all non-IP based protocols.
exists	If the field value exists, regardless of value, the operation will evaluate to true.
!exists	If the field value does not exist, the operation will evaluate to true.
length	Evaluates the length of the field. For example, "username length 20-u" would return usernames that were of length 20 or more charactors.

String values (Text or binary) support the following additional operators to search for partial values:

Field Operator	**Explanation**
contains	Searches a text or binary value for partial match.
regex	Performs a regular expression search against text or binary values.
Begins	Checks for value at the beginning of text or binary field
Ends	Checks for value at the end of text or binary field.

Query Clause

A query clause specifies both the select and where clause together and is used by the NwQuery function. It is valid to provide a select clause without a where clause or a where clause without a select clause. If no query clause is provided the query engine assumes "select *" returning all values.

Syntax: <select clause> <where clause>

Example query clauses:
"select *" - returns all objects in the database within the specified range

"where time='2007-Jul-24 12:00:00'-'2007-Jul-24 12:15:00'" – returns all sessions between 12:00 and 12:15 on July 24, 2007. Notice the use of single quotes to group values that contain spaces. Double quotes can also be used as well as the backslash \ to escape characters quoted characters.

"select sessionid,filename where username = 'todd' && service=110" – returns all session ids and filenames where username 'todd' exists and the service is POP3 (110).

Search Clause

The search string specifies the type of content search and the patterns to find.

Syntax: select (fields) where (searchtype1=value1 (options1)) && (searchtype2=value2 (options2)) && etc…
Fields: hit, pretext, posttext
Search Types: keyword | regex
Options: ci | cs | sm | nsm | sp | nsp | ds | nds
Default Options: cs | sp | nsm | nds
Logical Operations: && only
Precedence Operations: None (evaluated left to right)
Search value delimter: '', "", (), {}, []

The hit field gives the exact match, pretext gives text before the match, and posttext gives text after the match. These values can be used to create a search engine result display. The search types are keyword and regex (regular expression). These search types can be intermixed using the logical && operation. For performance reasons keyword search should precede regex searched.

The options are case insensitive (ci), case sensitive (cs), search meta (sm), no search meta (nsm), search packets (sp), no search packets (nsp), decode session (ds), and no decode session (nsd). Case insensitive would mean "HELLO" and "Hello" would be the same value, and case sensitive would mean the values were different. Search meta will content search meta field values of the session. Search packets scans packet payloads of the session. Decode session will render native types such as Mail and Web pages (decompress/uudecode) and then perform the search. By default the options will be case sensitive, no search meta, search packets, and no decode sessions.

Example search clause:
"select pretext, hit, posttext where keyword='call me' ci && regex='\b1? ?\(?\d\d\d(\(|\(|\-|\.)\d\d\d[-\.]\d\d\d\d\b'"

The above example does a case insensitive keyword search for 'call me' and if found then performs a regex search for a phone number pattern. The last search type specified is the result given so in the above example only the regex hits would be returned for phone numbers.

SDK Functions

NwOpen

```
nwuint32 NwOpen(const char *            url,
                nwuint32 *              collectionHandle,
                const char *            cacheDir,
                NwLoginChallengeHandler loginChallengeHandler,
                void *                  userData)
```

NwOpen opens a NetWitness® Collection that can be in a local directory or on a remote NetWitness® Server. A unique handle is always returned which represents the collection and is used for the majority of SDK functions. If a collection fails to open the returned handle can be used to get the last error.

The local cache directory is optional, but for best performance should be the same directory for a given URL. If NULL is passed, a temporary directory will be used and cleaned up when NwClose is called.

The URL can be a local directory such as "file:///C:/netwitness/collection1" or a remote server such as "nws://admin:password@myserver" or "nw://todd:mypass@recorder:50001/". The *file* specification requires three slashes followed by the local directory. The *nws* specification opens a secure encrypted connection to the remote server, and *nw* specifies a non-encrypted connection. Username, password, and server are required for remote connections and port number is optional and defaults to 50005.

The loginChallengeHandle and userData are for use when connecting to remote servers configured with external authentication schemes that require additional information at login. These parameters should be NULL for default NetWitness authentication. See NwLoginChallengeHandler in the SDK Callback Functions.

NwOpenRemote

```
nwuint32 NwOpenRemote(nwuint32                handle,
                      const char *            url,
                      nwuint32 *              collectionHandle,
                      const char *            cacheDir,
                      NwLoginChallengeHandler loginChallengeHandler,
                      void *                  userData)
```

Opens a NetWitness Collection on a remote server thru an existing authenticated session. (nws://username:password@service:port/ nw://username:password@service:port/) Service must be an existing node name under /services, on the service handle refers too. cacheDir is optional, but for best performance should be the same directory for a given URL. If NULL is passed, a temporary dir will be used and cleaned up when NwClose is called. loginChallengeHandler can be NULL for NetWitness authentication, but is

required for some external authentication systems like RSA. userData will be passed, as is, to the challenge handler.

NwClose

```
nwuint32 NwClose(nwuint32 handle)
```

NwClose will close a valid opened handle. All operations against the handle will fail after closing. Even if NwOpen fails you should still call close on the handle provided.

NwCreatePipeByPath

```
nwuint32 NwCreatePipeByPath(nwuint32     collectionHandle,
                            const char * targetPath,
                            nwuint32 *   pipeHandle)
```

For remote collections only, creates a pipe to the target node on the service.

NwCreatePipeByHandle

```
nwuint32 NwCreatePipeByHandle(nwuint32   collectionHandle,
                              nwuint32   targetPath,
                              nwuint32 * pipeHandle)
```

For remote collections only, creates a pipe to the target node on the service.

NwPipeResponseHandler

```
nwuint32 NwCreatePipeByHandle(nwuint32          pipeHandle,
                              NwResponseHandler handler,
                              void *            userData)
```

Adds a response handler to the pipe, which will be called when a response arrives for that pipe. Replaces any previous handler. Set to NULL to remove. The handler will be invoked on the pipe's background message handling thread.

NwSendMessage

```
nwuint32 NwSendMessage(nwuint32     pipeHandle,
                       const char * messageName,
                       const char * parameters)
```

Sends a asynchronous message to the pipe's target node. messageName is the name of the message (8 chars max, cannot be NULL) parameters are any inputs to the message. Can be NULL, a string, or name/value pairs (<Name>=<Value> or <Name>="<Value>", whitespace between pairs).

NwSendBinaryMessage

```
nwuint32 NwSendMessageBinary(nwuint32     pipeHandle,
                             const char * messageName,
                             const void * blob,
                             nwuint32     blobLength)
```

Sends binary data as input to the message.

NwWait

```
nwuint32 NwWait(nwuint32 pipeHandle,
                nwuint32 seconds)
```

Waits for a response to arrive for the specified seconds, or forever if zero.

NwStopWait

```
nwuint32 NwStopWait(nwuint32 pipeHandle)
```

Forces the thread waiting in NwWait on a response from pipeHandle to just return, regardless if a response has arrived or not.

NwResponseCount

```
nwuint32 NwResponseCount(nwuint32   pipeHandle,
                         nwuint32 * count)
```

Returns the number of response messages waiting.

NwResponseFlags

```
nwuint32 NwResponseFlags(nwuint32   pipeHandle,
                         nwuint32 * flags)
```

Returns the flags bitmask for the first response message.

NwResponseSize

```
nwuint32 NwResponseSize(nwuint32  pipeHandle,
                        nwuint32  index,
                        nwint32 * size)
```

Returns the size of the buffer needed for the front response of the pipe. Index is used for responses that contain arrays of data. If used, it will return the size needed at that index. If index exceeds the number of items in the array, size will be returned as -1. If the response does not contain an array, then index must be set to zero.

NwResponseData

```
nwuint32 NwResponseData(nwuint32  pipeHandle,
                        nwuint32  index,
                        void *        data,
                        nwint32 * size)
```

Returns the actual data of the front response. index is only used for response data types that are arrays, otherwise it must be zero.

NwResponseQueryResults

```
nwuint32 NwResponseQueryResults(nwuint32         pipeHandle,
                                nwuint64 *       field1,
                                nwuint64 *       field2,
                                struct NwField * result,
                                nwuint32 *       resultSize)
```

Returns the first waiting response as query results.

NwPopFrontResponse

```
nwuint32 NwPopFrontResponse(nwuint32 pipeHandle)
```

When finished processing the front response, this should be called to remove it from the queue.

NwInfo

```
nwuint32 NwInfo(nwuint32 collectionHandle,
                char *   info,
                nwuint32 infoSize)
```

NwInfo provides high level collection information such as time ranges, data sizes of collection, and version and hostname of remote server.

The returned buffer includes name value pairs formatted in the following syntax: *<name>:<value>;<name>:<value>;…*

NwSession

```
nwuint32 NwSession(nwuint32   collectionHandle,
                   nwuint64 * session1,
                   nwuint64 * session2,
                   nwuint64 * field1,
                   nwuint64 * field2);
```

NwSession provides object id ranges for sessions and fields which are used by NwValues, NwQuery, NwContent, and NwSearch functions. If session1 and session2 are zero then the return result will be the the entire range of session and field records. If session1 and session2 are non-zero then the associated field ranges for those sessions are returned.

NwLanguage

```
nwuint32 NwLanguage(nwuint32          collectionHandle,
                    nwuint64 *        lang1,
                    nwuint64 *        lang2,
                    struct NwField *  result,
                    nwuint32 *        resultSize)
```

NwLanguage retrieves the language element names used in functions such as NwValues and NwQuery. Language names are stored in the results array and for the ranges lang1 to lang2. If the result vector is not large enough to hold all language field names then the lang1 and lang2 ranges will be updated to the range remaining.

Each language element will specify the name of the field and format of the variant. The description of the language element is stored in the field variant sval. The flags variable of the Field structure will be set as follows:

```
Flags = 0 - Language key filtered
Flags = 1 - Recorded but not indexed
Flags = 2 - Index of Key Name
Flags = 3 - Index of Key Name & Distinct Values
```

The above settings are configured individually per language key. If set to 0 that means the field is not available and has been filtered out. If set to 1 then data is available but retrieving these field records will be slow for large datasets. If set to 2 then there exists a basic index that will offer slightly better performance on sparse data (ie—Data that is not created for every session). If set to 3 then queries will perform very fast especially if estimates are requested. It is recommended to only call NwValues on fields that are indexed as distinct values..

NwTimeline

```
nwuint32 NwTimeline(nwuint32          collectionHandle,
                    nwint64 *         time1,
                    nwint64 *         time2,
                    nwint32           timezone,
                    const char *      whereStr,
                    nwuint32          flags,
                    struct NwField *  results,
                    nwuint32 *        resultSize,
                    NwProgressHandler progressHandler,
                    void *            userData)
```

Retrieve counts for session/size/packets across a time range, results are given in discrete time intervals. time1 and time2 are specified as seconds since 1970 in GMT (POSIX

time). timezone is the timezone hour offset [-12 to 12] of the local computer from GMT, all time returned will be converted to local time. Zero means no conversion from GMT. Daylight savings time could extend the range to [-13 to 13]. whereStr can be used to filter the results (do NOT filter by time), it can also be NULL for no filtering. The flags field can specify the count type returned including session count, packet count, and session size in bytes. progressHandler and userData can both be NULL, otherwise the handler will be called for progress updates during the query.

NwValues

```
nwuint32 NwValues(nwuint32          collectionHandle,
                  nwuint64 *        field1,
                  nwuint64 *        field2,
                  const char *      fieldName,
                  const char *      whereStr,
                  nwuint32          flags,
                  struct NwField *  result,
                  nwuint32 *        resultSize,
                  NwProgressHandler progressHandler,
                  void *            userData)
```

NwValues retrieve distinct values for a specific language field (fieldName) across range numbers field1-field2 given an optional where clause up to the total number of results passed in. The flags field specifies the type of count given (session count, session size, field count, or packet count), sorting and ordering, and if a faster estimate should be used. If an estimate is asked for then the count will be usually be larger than the real count but the operation will complete much faster. Combine the count type, sorting, ordering, and estimate into a single bitmask.

```
Flags = 0x0001          - Total session count
Flags = 0x0002          - Total session size in bytes
Flags = 0x0004          - Total packet count
Flags = 0x0008          - Total field count

Flags = 0x0100          - Sort by Total
Flags = 0x0200          - Sort by Value

Flags = 0x0400          - Order Ascending
Flags = 0x0800          - Order Descending

Flags = 0x8000          - Estimate Total
```

NwQuery

```
nwuint32 NwQuery(nwuint32          collectionHandle,
                 nwuint64 *        field1,
                 nwuint64 *        field2,
                 const char *      queryStr,
                 struct NwField *  result,
                 nwuint32 *        resultSize,
                 NwProgressHandler progressHandler,
                 void *            userData)
```

NwQuery executes a query against objects starting at field1 up to object number field2. Fields are stored in the result vector that match the query up to the result vector size. If the array is filled before the scan completes it returns giving you the updated range that still needs to be scanned (field1 to field2). If field1 is set to 0 it will be defaulted to the first item in the database. If field2 is 0 it will be defaulted to the last item in the database. The resultSize variable specifies the total size of the array and is updated with the number of items used on return.

NwSearch

```
nwuint32 NwSearch(nwuint32          collectionHandle,
                  nwuint64          session,
                  const char *      searchStr,
                  struct NwField *  result,
                  nwuint32 *        resultSize)
```

NwSearch will content search a single session for keywords or regex patterns and returns any hits found in that session. The searchStr can be written similiarly to NwQuery, where the fields in the select clause are: “hit”, “pretext”, and “posttext”. The where clause can be used to specify either a keyword search, regex search, or combinations of both using the && operation. Order is important. If you specify a keyword search followed by a regex search and the first keyword search does not hit, then this avoids performing the slower regex based search. If keyword search does hit, then the next search condition is applied, and so on. Results are only returned on the last search condition specified. The where cause looks like “where keyword=’USA’ ci && regex=’united states|america’ ci”. The options following each search can be “ci” for case insensitive or “cs” for case sensitive.

NwContent

```
nwuint32 NwContent(nwuint32     collectionHandle,
                   nwuint64     session,
                   nwuint32 *   renderType,
                   nwuint32     renderFlags,
                   nwuint32     maxSize,
                   const char * options,
                   const char * filename)
```

Renders a session as either a html file or pcap file based on the specified render type specified. If the render type is not specified (0) it will automatically select the best html render type for display. If using a html render type it will overwrite the existing file. If the render type is pcap file it will append to the file if it already exists. The render flags can specify viewing only stream 1 or 2 of the session, single column view, and displaying only the payload of packets in the hex/packet view. The maxsize will truncate the content retrieved to the specified byte size unless set to 0. The options parameter can be NULL or used to specify the charset to use by providing “charset: ‘encoding name’;”.

```
renderType =  0 - Auto Determine View
renderType =  1 - Field Details View
renderType =  2 - Text View
```

```
renderType =  3 - Hex View
renderType =  4 - Packets View
renderType =  5 - Mail View
renderType =  6 - Web View
renderType =  7 - VOIP (RTP) View
renderType =  8 - Instant Messenger View
renderType =  100 - PCAP File

renderFlags |= 1 - Stream 1 of Session
renderFlags |= 2 - Stream 2 of Session
renderFlags |= 4 - Single Column View (Otherwise Left/Right View)
renderFlags |= 8 - Display Payload of Packets (Hex and Packets view)
```

NwLastError

```
nwuint32 NwLastError(nwuint32   collectionHandle,
                     nwuint32 * errorCode,
                     char *     error,
                     nwuint32   errorSize)
```

Retrieves the last error of the specified collection of the calling thread if any of the functions failed.

NwToString

```
nwuint32 NwToString(struct VariantData * variant,
                    char *               str,
                    nwuint32             strSize)
```

Coverts the variant found in the Field structure into string representation.

NwThreadId

```
nwuint32 NwThreadId(nwuint32 * threadId)
```

Retrieves a thread id associated with the calling thread which can be used by NwCancel from a different thread.

NwCancel

```
nwuint32 NwCancel(nwuint32 collectionHandle,
                  nwuint32 threadId)
```

Attempts to cancel the current operation identified by a previous call to NwThreadId.

NwGetNameValue

```
nwuint32 NwGetNameValue(const char * data,
                        nwuint32     pos,
                        char *       name,
                        nwint32 *    nameSize,
                        char *       value,
                        nwint32 *    valueSize)
```

Retrieve the name/value pair at the zero based pos. Returns -1 for size if no name/value pair exists at pos.

SDK Callback Functions

NwResponseHandler

```
void (*NwResponseHandler)(nwuint32 collectionHandle.
                          nwuint32 pipeHandle,
                          void *   userData)
```

A callback that is invoked when responses for a pipe arrive. See the SDK Function NwPipeResponseHandler.

NwLoginChallengeHandler

```
void (*NwLoginChallengeHandler)(const char * challengePrompt,
                                nwint32 *    challengeType,
                                char *       response,
                                nwuint32     responseBufferSize,
                                void *       userData)
```

This callback is used for some external authentication schemes like RSA when calling NwOpen and NwOpenRemote. The challengePrompt will be the prompt that should be displayed to the end user. The challenge answer should be filled in the response buffer. Do not exceed responseBufferSize, including the NULL character at the end of the string. challengeType indicates the type of challengePrompt (echo input, mask input, info prompt, etc).

NwProgessHandler

```
void (*NwProgessHandler)(nwint32          percentComplete,
                         const char *     extraData,
                         struct NwField * result,
                         nwuint32 *       resultSize,
                         void *           userData)
```

This is used by the NwValues call and can return partial results while a query is ongoing. When no results are present, results and resultSize will be NULL. results and resultSize, when not NULL, will always be the same addresses that were passed to NwValues.

Appendix

A. Variant Structure

The variant structure holds a value of a specified type which could be either numeric or string based. It specifies the format, size, and value it is holding. The max size of a string is 256 bytes.

```
struct VariantData
{
    nwuint16 format;
    nwuint16 size;

    union Value
    {
        nwint8 int8;
        nwuint8 uint8;
        nwint16 int16;
        nwuint16 uint16;
        nwint32 int32;
        nwuint32 uint32;
        nwint64 int64;
        nwuint64 uint64;
        nwuint128 uint128;
        nwfloat32 float32;
        nwfloat64 float64;
        char  sval[VariantMaxSize];
    } value;
};
```

The possible variant formats are as follows:

```
    Undefined  =  0
    Int8       =  1
    UInt8      =  2
    Int16      =  3
    UInt16     =  4
    Int32      =  5
    UInt32     =  6
    Int64      =  7
    UInt64     =  8
    UInt128    =  9
    Float32    =  10
    Float64    =  11
    TimeT      =  32
    DayOfWeek  =  33
    HourOfDay  =  34
    Binary     =  64
    Text       =  65
    IPv4       =  128
    IPv6       =  129
    MAC        =  130
```

B. Field Structure

The following is the Field structure and description of each element:

```
struct NwField
{
    nwuint64 group;
    union TYPE
    {
        nwuint128 handle;
        char name[16];
    } type;
    struct VariantData variant;

    nwuint64 id1;
    nwuint64 id2;
    nwuint64 count;
    nwuint32 flags;
};
```

Name	Description
Group	The group number specifies a session id for the meta record and is used to group related pieces of meta together. All meta of the same group id are based from the same session.
Type	The type is the field name which can be represented as a 128 bit number or a 16 byte string. The name may not be padded with a null value if the size is 16 bytes.
Variant	The variant data structure contains a specified format, a size, and the value itself.
Id1	The unique meta id for this field. If NwValues was called this will be your starting id for this distinct value.
Id2	The unique meta id for this field. If NwValues was called this will be your ending id for this distinct value.
Count	If NwValues was called this reflects the number times this field was seen inside the inclusive id1-id2 range.
Flags	Future use

C. Field Names and Data Types

The following is a list of currently supported field names. This information can be extracted using the NwLanguage function call.

Category	Element Name	Data Type	Description
Network			
	sessionid	UInt64	Session ID
	time	TimeT	Start Time
	size	UInt32	Size
	eth.src	MAC	Ethernet Source Address
	eth.dst	MAC	Ethernet Target Address
	eth.type	UInt16	Ethernet Protocol
	ip.proto	UInt8	IP Protocol
	ip.src	IPv4	Source IP Address
	ip.dst	IPv4	Destination IP Address
	ipv6.src	IPv6	Source IPv6 Address
	ipv6.dst	IPv6	Target IPv6 Address
	ipv6.proto	IPv6	IPv6 Protocol
	tcp.srcport	UInt16	TCP Source Port
	tcp.dstport	UInt16	TCP Destination Port
	udp.srcport	UInt16	UDP Source Port
	udp.dstport	UInt16	UDP Target Port
Application			
	service	UInt32	Service Type
	action	Text	Action Event (login, ogoff, sendfrom, sendto, get, put, delete, attach, print)
Entities			
	username	Text	User Account
	email	Text	E-mail Address
	filename	Text	Filename resource
	handle	Text	Resource Handle
	database	Text	Database name
	group	Text	Group Channel
Alias Records			
	alias.ip	IPv4	IP Address Alias Record
	alias.host	Text	Hostname Record
Properties			
	content	Text	Content Type
	fullname	Text	Fullname
	nickname	Text	Nickname
	buddy	Text	Buddy Name
	client	Text	Client Application
	server	Text	Server Application
	password	Text	Password
	cookie	Text	Cookie
	response	Text	Response
	referer	Text	Referer

created	Text	Created
modified	Text	Modified
generator	Text	Generated
message	Text	Message
subject	Text	Subject
attachment	Text	Subject
crypto	Text	Crypto Key
org	Text	Organization
orig_ip	Text	Originating IP Address
link	Text	Link
renewal	Text	Renewal
dns	Text	Dns
address	Text	Address
subnet	Text	Subnet
sql	Text	Sql Query
sqlresponse	Text	Sql Response
create	Text	Create
invite	Text	Invite
crc	Text	32bit CRC Hash
md5	Text	MD5 Hash
phone	Text	Phone Number
device	Text	Device Name
signature	Text	Signature
alertid	Text	Alert ID
sourcefile	Text	Source File
found	Text	Found
match	Text	Match
encapsulated	Text	Encapsulated
data_chan	Text	Data Channel
proxy	Text	Proxy Name

D. SDK Examples

All SDK examples assume a valid collection has already been opened with the NwOpen function call and properly closed after the function returns. Supporting functions such as checking error codes and printing fields are included in the code examples below.

1. Print All Records

This code sniplet will retrieve and print all field records.

```
void dbDump(nwuint32 handle)
{
	/* setup local variables */
	nwuint64		field1 = 0;
	nwuint64		field2 = 0;
	nwuint32		size = 100;
	struct NwField	fields[100];
	nwuint32		result = FAILURE;
	unsigned int	i;

	printf("\n\nDumping All Field Records\n");

	/* query all records paging 100 at a time */
	while (field1 <= field2
		&& (result = NwQuery(handle, &field1, &field2, "",
					fields, &size, NULL, NULL))
		&& size > 0)
	{
		/* dump all received fields */
		for (i = 0; i < size; ++i)
		{
			printField(&fields[i]);
		}
	}

	/* output any error info */
	check(result, handle);
}
```

2. Retrieving Top N Lists

This function shows how to retrieve a TOP 10 list of desending session counts for any field variable (ex: service). Changing the size value and fields array to match will give you any top N you choose.

```
/* get top 10 sessions counts of a specified field */
void dbValues(nwuint32 handle, const char* field)
{
	/* setup local variables */
	nwuint64		field1 = 0;
	nwuint64		field2 = 0;
	nwuint32		size = 10;
	struct NwField	fields[10];
	nwuint32		result = FAILURE;
	unsigned int	i;
	char			name[17];
	char			value[257];

	/* clear name and value fields */
	memset(name, 0, sizeof name);
	memset(value, 0, sizeof value);

	printf("\nTop 10 Values for %s\n",field);

	/* retrieve the top 10 report */
	if ((result = NwValues(handle, &field1, &field2, field, "", 0,
				0, fields, &size, NULL, NULL))
		&& size > 0)
	{
		/* cycle through all returned results */
		for (i = 0; i < size; ++i)
		{
			/* print results to screen */
			printField(&fields[i]);
		}
	}

	/* output any error info */
	check(result, handle);
}
```

3. Content Search and Rendering

This code sniplet will find all FTP sessions, content search them for “todd”, and render the first session that hits to a html file.

```
void dbSearch(nwuint32 handle)
{
      /* setup local variables */
      nwuint64          field1 = 0;
      nwuint64          field2 = 0;
      nwuint32          size = 100;
      struct NwField    fields[100];
      nwuint32          result = FAILURE;
      unsigned int      i;

      /* query all records paging 100 at a time */
      while (field1 <= field2
             && (result = NwQuery(handle, &field1, &field2,
                                  "select sessionid where service=21",
                                  fields, &size, NULL, NULL))
             && size > 0)
      {
            /* cycle through all returned sessions */
            for (i = 0; i < size; ++i)
            {
                  /* perform content search for keyword "todd" */
                  struct NwField    search;
                  nwuint32          searchsize = 1;

                  nwuint64 sessionid = fields[i].variant.value.uint64;
                  if ((result = NwSearch(handle, sessionid,
                                 "select hit where keyword='todd' ci",
                                 &search, &searchsize))
                      && searchsize > 0)
                  {
                        /* auto display content to hit.html and exit */
                        nwuint32 rendertype = 0;
                        result = NwContent(handle, sessionid,
                                           &rendertype, 0, 0, NULL,
                                           "c:\\hit.html");

                        /* print the hit we have found */
                        printField(&search);
                  }
                  /* check if we should stop searching because of
                     failure or hit found */
                  if (result == FAILURE || searchsize > 0)
                  {
                        field1 = field2+1;
                        break;
                  }
            }
      }
      /* output any error info */
      check(result, handle);
}
```

4. Printing Field Records

This function will print out a field record to the screen.

```
void printField(struct NwField *field)
{
	char		name[17];
	char		value[257];

	// clear name and value fields
	memset(name, 0, sizeof name);
	memset(value, 0, sizeof value);

	// copy the name of the field into a safe buffer for printing
	strncpy(name, field->type.name, 16);

	// convert the variant value to a string
	NwToString(&field->variant,value,256);

	// check if id1 and id2 are same
	if (field->id1 == field->id2)
	{
		// output field range to screen
		printf("id=%-5I64u  session=%-5I64u  %16s=%s",
			field->id1, field->group, name, value, field->count);
	}
	else
	{
		// output field range to screen
		printf("ids=%-5I64u - %-5I64u  session=%-5I64u  %16s=%s",
			field->id1, field->id2, field->group, name,
	value);
	}

	// if we have a count print that
	if (field->count)
	{
		printf("  count:%I64u", field->count);
	}

	// new line
	printf("\n");
}
```

5. Checking Error Conditions

This function will check the return result of any SDK function and output if it was successful (ok) or on failure will output the error message.

```
/* check error status */
void check(nwint32 result, nwint32 handle)
{
	char error[256];

	strcpy(error, "ok");

	if (result == FAILURE)
	{
		NwLastError(handle, error, 256);
	}

	printf(" - handle: %i result: %i message: %s\n",
		handle, result, error);
}
```