Written by : Barak Weichselbaum, all rights reserved 2000-2009

Site: http://www.komodia.com/
Email : barak@komodia.com

KOMODIA'S REDIRECTOR DLL FRAMWORK
Programmer's Manual

Introduction

This manual covers the programming aspects of Komodia's Redirector product when writing a DLL to extend the Redirector's functionality. Another option is to modify the source code directly. More on this option in the PCProxy direct programming manual, at:
http://www.komodia.com/KRDR_manual.pdf

Overview

You can download the DLL framework from: http://www.komodia.com/PCProxyDLL.zip
Inside there are two directories:

1. PCProxyDLL\ This directory contains an empty skeleton.
2. PCProxySample\ This directory contains a suggested DLL with code inside that can perform filtering and data peeking.

The file to open in either directory is PCProxyDLL.dsw, which opens the project in VS.

Parental control functions

Komodia's Redirector have an optional Parental control module which frees you from the need to parse and decode HTTP requests and replies (chunked transfer encoding and Gzip encoding are transparent to you), in case the product you have purchased/evaluating has the parental control module the DLL will support two extra functions which are described later in this document.

DLL Functions

The DLL exposes seven functions for PCProxy to use. PCProxy performs a sanity check on the DLL when loading it. If one of the functions is missing, PCProxy will not load the DLL.

PCInitialize

```
bool _stdcall PCInitialize();
```

This function is called after the DLL is loaded and passes the sanity check. All user-specific initializations, such as creating threads and global variables, should be here.

If the return value is false, the DLL will be unloaded and PCProxy will not use it.

PCUninitialize

```
void _stdcall PCUninitialize();
```

This function is called before PCProxy unloads the DLL. It is not called if Initialize returned a false value. All uninitializations should be performed here.

Deallocate

```
void _stdcall Deallocate(char* pData);
```

This function is called to deallocate data that the DLL has allocated. The reason for this function is that different types of compilers and languages use different types of allocation and deallocation commands. Thus, PCProxy will deallocate data allocated by the DLL using this method.

In the sample we use a C++ array delete:

```
delete [] pData;
```

NewConnection

```
bool _stdcall NewConnection(DWORD& rContext,
                            bool bFromEncryption,
                            unsigned long usListeningPort,
                            unsigned long& rIP,
                            unsigned short& rPort,
                            const ProcessInformation& pPInformation,
                            bool& rEnableSSL,
                            bool& rSSLSwitch,
                            bool& bFreezeReceive,
                            const char* pPeekData,
                            unsigned long ulPeekDataSize,
                            unsigned long& rWaitMoreData,
                            ProxyInformation& rProxyInformation)
```

This function is called whenever a new connection was made to PCProxy.

The parameters are:

rContext – A DWORD to be used across all functions that are called for this connection. The best practice is to allocate a struct with all necessary connection-specific information, and store its address inside this variable.

bFromEncryption – In case the data was originally encrypted with SSL but decrypted with the SSL decryptors, this flag will be true.

usListeningPort – The port of the socket that accepted the connection inside the Redirector (currently it will be 12344 or 12346).

rIP – The destination IP that this session is trying to connect to. The programmer can change this value and the session will connect to the new IP.

rPort – The destination port that this session is trying to connect to. The programmer can change this value and the session will connect to the new port.

pPInformation – Contains information about the calling process, the structure definition is:

```
typedef struct _ProcessInformation
{
      unsigned long           ulPID;
      const unsigned short*   pProcessName;
      const unsigned short*   pDomain;
      const unsigned short*   pUsername;
} ProcessInformation;
```

The Process name, Username and Domain are stored in Unicode, the following code snipest converts them to std::wstring:

```
std::wstring sUsername((wchar_t*) pPInformation.pUsername);
```

rEnableSSL – Outputs this stream as SSL data (the port will usually be changed to 443).

rSSLSwitch – Set to true to notify that you might change this connection to SSL at a later stage, but not now. Useful for proxies.

bFreezeReceive – Set to true to tell the Redirector not to send you any client data for processing until you are ready. Useful when negotiating with a 3rd party proxy.

pPeekData – This buffer contains data "peeked" from the client, which means for example that in a web request this can contain the actual request before making any connection, this is useful when wanting to make connecting decisions based on the traffic type.

rWaitMoreData – This flag sets the timeout in milliseconds to wait before invoking this function again, more information on this in a few paragraphs below.

rProxyInformation – Contains the proxy information, in case there's a global proxy defined, the structure will hold this data, it is also possible to modify the struct to disable the proxy, enable a proxy, and of course set the proxy settings, this setting is per connection.

The structure definition:

```
typedef enum _DLLProxyType
{
      dptHTTP,
      dptHTTPConnect,
      dptHTTPConnectSSL,
      dptHTTPHybrid,
      dptHTTPHybridSSL,
      dptSocks4,
      dptSocks5
} DLLProxyType;


typedef struct _ProxyInformation
{
      bool              bUsingProxy;
      DLLProxyType      aProxyType;
      char*             pUsername;
      char*             pPassword;
      unsigned long     ulIP;
      unsigned short    usPort;
} ProxyInformation;
```

Return value:
True – Allow this session
False – Reject this session

Sample code:

```
//Allow only port 80
if (usPort!=80)
    return false;
else
    return true;
```

This code allows only port 80 sessions through. Keep in mind that it only blocks sessions that are intercepted by the LSP according to the rules you set.

Sample code for redirecting all port 80 sessions to SSL:

```
//Is it HTTP port?
if (rPort==80)
{
      //Change port to HTTPS
      rPort=443;

      //Enable the SSL
      rEnableSSL=true;
}
```

Sample code for saving session specific information inside the context variable:

```
//Defined at global scope
typedef struct _DLLData
{
      DWORD             dwIP;
      Unsigned short    usPort;
} DLLData;

//This part goes into the function
DLLData* pDLLData=new DLLData;
pDLLData->dwIP=rIP;
pDLLData->usPort=rPort;

//Store the struct at context level
rContext=(DWORD)pDLLData;

//Some code

//Done
return true;
```

Inside the sample DLL (the one with the code), this function contains code that detects whether the session is HTTP and if so, it waits until it has the complete session.

HandleProxyHTTPConnect

```
bool _stdcall HandleProxyHTTPConnect(const char* pConnectString,
                                     char** ppNewString,
                                     DWORD dwContext)
```

This function is called after the Redirector creates the HTTP Connect clause, it allows you to modify the content of it, and usually it is done to add custom authentication parameters.

pConnectString – Original connection clause (null terminated)

ppNewString – If the string is changed, the new string (null terminated) will be allocated into this variable (by you)

dwContext – The DLL-specific context that was created at the NewConnection function

Return value:
True – Clause was modified
False – Clause wasn't modified

DataBeforeSend

```
bool _stdcall DataBeforeSend(const char* pData,
                             DWORD dwDataSize,
                             char** pNewBuffer,
                             DWORD& rNewDataSize,
                             bool& rTerminateSession,
                             bool& rNewBufferToReceive,
                             OutgoingDataManipulations& rManipulations
                             DWORD dwContext);
```

This function is called before sending the data to the final destination. For example, HTTP requests are seen at this stage, keep in mind that TCP is stream based which means that data comes in chunks, therefore HTTP requests can be chunked across multiple calls, there are number of ways to make sure you receive a complete web request:

1. Use the code inside NewConnection (under the DLL sample) which waits until it has the complete web request, and then you will receive the complete request here.
2. Use parental control add-on module, which does this for you transparently, you, can see how the parental control add-on module works later in this manual.
3. In case you want to do perform header manipulation such as removing/modifying/adding fields, you don't need the complete request, you can just use the built in filtering solution that does that for you (sample code is inside the DLL sample)

The parameters are:

pData – The data to process.

dwDataSize – The size of data to process.

pNewBuffer – If the data is modified, this will hold the modified data. Must be allocated by the programmer, since the proxy will delete it.

rNewDataSize – If the data is modified, this will hold the modified data size.

rTerminateSession – Set to true to terminate the session when this function exits.

rNewBufferToReceive – Takes the contents of the new data and sends it to the application that initiated the session (used mostly for redirects).

rManipulations – This structure contains the manipulations you want to employ on a HTTP header, of course this is optional. You don't use this structure directly, instead you use a helper class called CHTTPHelper which is described later in this document.

dwContext – The DLL-specific context that was created at the NewConnection function.

Return value:
True – Data was modified.
False – Data wasn't modified.

The flag rTerminateSession is not affected by the return value. You can terminate the session without changing the data that you received inside pData.

Sample code to redirect the web traffic into a different site:

```
	//Check that we are port 80
	//Get socket data from the context
	DLLData* pDLLData=(DLLData*)dwContext;

	//Set terminate to false, just in case
	rTerminateSession=false;

	//Check we are port 80
	if (pDLLData->usPort==80 &&
	    pDLLData->dwIP!=inet_addr("64.118.87.10")) //This is our web
site address
	{
		//Our quote
		std::string sQuote;
		sQuote='"';

		//Build a new string
		std::string sTmp;
		sTmp+="<html><head><META HTTP-
EQUIV="+sQuote+"Refresh"+sQuote+" CONTENT="+sQuote+"0;URL=";
		sTmp+="http://www.komodia.com";
		sTmp+=sQuote+"></head></html>";
```

```
                //Convert the size
                char aTmp[11];
                itoa(sTmp.size(),
                       aTmp,
                       10);

                //HTML header
                std::string sHeader;
                sHeader="HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n";
                sHeader+="Content-Encoding: text/html\r\n";
                sHeader+="Content-Length: ";
                sHeader+=aTmp;
                sHeader+="\r\n\r\n";
                sHeader+=sTmp;

                //Our size
                rNewDataSize=sHeader.size();

                //Allocate new buffer
                *pNewBuffer=new char[rNewDataSize];
                memcpy(*pNewBuffer,
                         sHeader.c_str(),
                         sHeader.size());

                //Send to client
                rNewBufferToReceive=true;

                //Terminate the session
                rTerminateSession=true;

                //Set to change
                return true;
        }
        else
                return false;
```

In this code we get the session information, and then redirect every session that will connect to port 80 to Komodia's website using HTML redirection. The code compares the destination IP of the current session to the IP of Komodia's website and if the addresses matches there will be no redirection, this to avoid redirection loops.

Inside the sample DLL this functions contains code that modifies the header, in this case, it modifies the user-agent field, adds a new fields called "X-Redirector" and deletes the field "refer":

```
        //Do we have a context?
        ContextData* pContext;
        pContext=(ContextData*)dwContext;

        //This code adds HTTP manipulations
        //This code also make sure to set the filter once per request,
because a request can come in many times
        //Because it is fragmented
        //Is this HTTP?
        if (pContext &&
                pContext->bNewRequest && //This checks that we are inside a
new request, and not inside an existing request
                IsHTTP(pData,
```

```
                    dwDataSize))
      {
            //The request can fragment and we don't want to be here per
fragment, but per request
            pContext->bNewRequest=false;

            //Here we build the filter
            //These fileds are just for the sake for demonstration
            //Please change them to reflect your own fields
            CHTTPHelper aHelper;

            //Each method can be called multiple times
            //Fields to change
            //Uncomment the next code, keep in mind that it may cause
problems with
            //Sites that are based on your browser
            aHelper.ChangeHTTPHeader("user-agent","Blank agent");

            //Fields to add
            aHelper.InsertSpecialField("X-Redirector","Just a value");

            //Fields to remove
            aHelper.FilterHTTPHeader("Refer");

            //Build the struct
            //Don't call this function unless you want to have a filter
            //Or have a filter override
            rManipulations=aHelper.GetFinalStruct();

            //Done
            return false;
      }

      //Done
      return false;
```

This is the sample code taken from the DLL, the class used here CHTTPHelper is documented later in this chapter.

DataBeforeReceive

```
bool _stdcall DataBeforeReceive(const char* pData,
                                DWORD dwDataSize,
                                char** pNewBuffer,
                                DWORD& rNewDataSize,
                                bool& rTerminateSession,
                                bool& rThawReceiveFreeze,
                                bool& rEnableSSL,
                                bool bServerClosed,
                                DWORD dwContext);
```

This function is called before sending the data back to the application that initiated the session. Any answer from a web server will be caught by this function, keep in mind that answers comes in fragmented, if you need HTTP answers to come in complete and/or decoded you might consider using the parental control add-on that is documented later in this chapter.

The parameters are:

pData – The data to process.

dwDataSize – Size of data to process.

pNewBuffer – If the data is modified, this will hold the modified data. It must be allocated by the programmer, since the proxy will delete it.

rNewDataSize – If the data is modified, this will hold the modified data size.

rTerminateSession – Set to true to terminate the session when this function exits.

rThawReceiveFreeze – Set to true to allow Redirector to process data from the client that was held due to the bFreezeReceive flag in the function NewConnection.

rEnableSSL – Set to true to switch to SSL session. This can be set only if the flag rSSLSwitch was set during the NewConnection method call.

bServerClosed – When this flag is set, it means that the remote server closed the connection and at this stage you have last chance to send data to the client side before the Redirector closes the connection.

dwContext – DLL-specific context that was created at the NewConnection function.

Return value:
True – Data was modified.
False – Data was not modified.

The flag rTerminateSession is not affected by the return value. You can terminate the session without changing the data that you received inside pData.

ConnectionClosed

```
void _stdcall ConnectionClosed(bool bError,
                               DWORD dwContext);
```

This function is called when the session has ended.

bError – Indicates whether the session ended because of an error (failed connection attempt), or because the remote side ended the connection.

This is the place to release the dwContext if it was allocated. Sample code:

```
delete (DLLData*)dwContext;
```

Threading model

Calls to NewConnection come from different threads.
Calls to DataBeforeReceive for a specific connection are always from the same thread.
Calls to DataBeforeSend for a specific connection are always from the same thread, but not the same thread as that of DataBeforeReceive.

CHTTPHelper

This class is used to create HTTP filtering, e.g. Adding, modifying and removing HTTP header fields.

Inside the PCProxySample DLL sample there's a usage of this class in the correct context. This section will go over the four main functions of the class.

Overview

When adding parameters to the functions don't add the ':':

Correct: "Connection"
Incorrect: "Connection:"

And never terminate the strings with \r\n

Correct: "Some data"
Incorrect: "Some data\r\n"

Headers are case insensitive, data is case sensitive.

FilterHTTPHeader

```
void FilterHTTPHeader(const std::string& rHeader);
```

Use this function to add name of headers to remove from the request, a sample header would be "If-Modified-Since" removing this field will not allow server to give you cached result.

ChangeHTTPHeader

```
void ChangeHTTPHeader(const std::string& rHeader,
                      const std::string& rNew);
```

Use this function to change data of a specific header, for example:

```
ChangeHTTPHeader("user-agent",
                 "Some agent");
```

The filter will only modify existing headers, if the header isn't in the request it will not be added.

InsertSpecialField

```
void InsertSpecialField(const std::string& rField,
                        const std::string& rData);
```

Use this function to add a field to the request, the field will be added only if it doesn't exist in the original request:

```
InsertSpecialField("X-Redirector",
                   "Some data");
```

If you want to make sure a field always exists with a value you want but you're no sure whether this field always appear, you can set the filter to delete the header field and then insert your field.

GetFinalStruct

```
OutgoingDataManipulations GetFinalStruct(bool bOverideProxy=false,
                                         bool bOveridGlobal=false)const;
```

This is the function that converts all the data you entered into a struct the Redirector can use.

Parameters:

bOverideProxy - Set to true to disable HTTP proxy and HTTP hybrid proxy filtering (which adjusts the request from normal request to a proxy request)

bOveridGlobal – Set to true to disable global HTTP filtering (which you set via the console)

Parental control add-on module

Integration with existing DLL functions

When working with parental controls, you may still want to handle non HTTP data therefore all functions that part of the regular DLL extension are still active when using parental control, you may choose to use them or not, it's up to you.

Architecture

After a connection is being made (after NewConnection was called) the Redirector assembles the outgoing request (e.g. GET / HTTP/1.1) and when it's complete (incase of a POST request, when it contains all the POST data) the Redirector calls the function HTTPRequestBeforeSend so it can perform analysis on the outgoing request, keep in mind that during the assembly of the request the method DataBeforeSend is called per chunk, but if you choose to work with the parental control feature you can just ignore it (ignoring=not performing any operation at that level)

At this stage you have number of options available to you:

- Do nothing (dhrNothing) – Request will be sent normally.
- Don't perform parental on this request (dhrDontDoParental) – Incase you know that you have no need to further parse this specific request, the answer received will not be parsed with the parental control add-on, if there will be a request made after this one in the same socket connection, the process will start again (the don't perform parental control is good for one request only)
- Block the connection (dhrBlock) – The request is not sent and the connection is closed.
- Redirect the connection to a different URL (dhrRedirect) – You specify a URL you want the browser to be redirected to, e.g. if you specify http://www.komodia.com the session will be redirected to Komodia's web site.
- Modify (dhrModify) – You modify the header to contain custom data.
- Return HTML (dhrReturnHTML) – The Redirector will take your HTML, wrap it with HTTP header and will send it back to the browser, while discarding the original request, this is good for custom block pages.
- Return HTML and header (dhrReturnHTMLAndHeader) – You specify the HTML and HTML header to return back to the browser, while discarding the original request, this is good for block pages.

Parsing of the header is done via a helper class called CHTTPHeader.

CHTTPHeader

This class is used to help you work with HTTP headers, although the Redirector gives you complete requests/replies, you still want to do some inspection on them, this class helps you by parsing the requests/replies and give you methods to easily get and modify the data you want.

Feed / ForceFeed

This two functions are used to give the class the raw data, the difference between Feed and ForceFeed is that Feed is sequential, which means that you can give it sequential fragmented chunks of the header, unlike ForceFeed that clears previous content and start from the beginning, our case we can use either methods, return value is true for indication we have a complete header and false for not having a complete header, usage example:

```
CHTTPHeader aHeader;
if (aHeader.Feed(pHeader,
                 dwHeaderSize))
{
      //Do something
}
```

GetHeaderString / GetHeaderStringCase

After we got a parsed header we want to start inspecting the contents, this two functions get the data of the specific HTTP headers (the only difference is that GetHeaderStringCase preserves the case of the data), for example, lets look at this simple HTTP request:

```
GET /index.php HTTP/1.1
User-agent: Mozilla Firefox
Connection: Keep-alive
Host: www.google.com
```

Now lets get the value of connection (without preserving case), keep in mind that all requests are made in lower case (we are assuming we continue from the last example)

```
std::string sConnection(aHeader.GetHeaderString("connection"));
```

IsRequest

```
bool IsRequest()const;
```

This function will return true if the parsed header was a HTTP request and false if it was a HTTP reply.

GetHost

```
std::string GetHost()const;
```

This function will give us the host of the request (relevant to requests only)

GetResponseCode

```
unsigned short GetResponseCode()const;
```

This function returns the HTTP response code (relevant to replies only)

GetURL

```
const std::string& GetURL()const;
```

This function returns the URL of the request, in our example it would be /index.php

GetFullAddress

```
const std::string& GetFullAddress()const;
```

This function returns the full address of the request, in our example it would be: http://www.google.com/index.php

SetHeader

```
void SetHeader(const std::string& rHeader,
               const std::string& rData);
```

This function is used to modify the header, in case the header given exists it will be changed, if the header do not exist, it will be added with the specified value, for example changing our header connection type (rHeader is case insensitive):

```
AHeader.SetHeader("connection","close");
```

DeleteHeader

```
void DeleteHeader(const std::string& rHeader);
```

This function will delete a header from the request/reply (if it exists, rHeader is case insensitive)

GetHeaderData

```
DataVector GetHeaderData(DWORD dwStart=0)const;
```

Get the data of the header as a vector of chars, this data will include all the changes we have performed on the header.

HTTPRequestBeforeSend

```
DLLHTTPResult _stdcall HTTPRequestBeforeSend(const char* pHeader,
                                             DWORD dwHeaderSize,
                                             char** ppNewHeader,
                                             DWORD& rNewHeaderSize,
                                             DWORD dwContext);
```

The returned enum definition:

```
typedef enum _DLLHTTPResult
{
      dhrNothing=0,
      dhrBlock,
      dhrRedirect,
      dhrModify,
      dhrReturnHTML,
      dhrReturnHTMLAndHeader,
      dhrDontDoParental
} DLLHTTPResult;
```

This function is called when an outgoing request is ready to be inspected.

The parameters are:

pHeader – The pointer containing the header data

dwHeaderSize – Size of the header in bytes.

ppNewHeader – If you need to return data to the Redirector, in case of modify or redirect, you need to allocate the data and give it to this pointer.

rNewHeaderSize – Size of the new data or redirect address.

dwContext – DLL-specific context that was created at the NewConnection function.

Return value:

Action to take with this request, you can look what each value means under the Architecture section.

Sample code taken from the DLL (you can download the DLL sample from: http://www.komodia.com/PCProxyDLL.zip)

```
//Take the context
ContextData* pData;
pData=(ContextData*)dwContext;

//At this stage we have a complete header
//It's easiest to work with the supplied class
//For header parsing
//Give it to the header
CHTTPHeader aHeader;
```

```
if (aHeader.Feed(pHeader,
                 dwHeaderSize))
{
      //We can get some usefull information at this stage
      //Host
      std::string sHost(aHeader.GetHost());

      //URI
      std::string sURI(aHeader.GetURL());

      //The full address
      std::string sFullAddress(aHeader.GetFullAddress());

      //Save the host in the session data
      if (pData &&
          pData->sHost.empty())
            pData->sHost=sHost;
```

This code snipest parses the header and extracts basic data we can use, now lets perform a redirect:

```
//Lets say we want to build a redirect based on the request
//We need to make sure that we are not redirecting over and over
//The site we are redirecting to
//This code performs it
if (!strstr(sHost.c_str(),
            "komodia.com") &&
    !strstr(sHost.c_str(),
            "google"))
{
      //Redirect to address, always in the form of http://website
      std::string sRedirectTo;
      sRedirectTo="http://www.komodia.com";

      //Save the data
      *ppNewHeader=new char[sRedirectTo.size()+1];
      memcpy(*ppNewHeader,
             sRedirectTo.c_str(),
             sRedirectTo.size()+1);

      //Set size
      rNewHeaderSize=sRedirectTo.size()+1;

      //Let Redirector know that this is a redirect
      return dhrRedirect;
}
```

HTTPRequestBeforeReceive

```
DLLHTTPResult _stdcall HTTPRequestBeforeReceive(const char* pHeader,
                                                DWORD dwHeaderSize,
                                                char** ppNewHeader,
                                                DWORD& rNewHeaderSize,
                                                const char* pData,
                                                DWORD dwDataSize,
                                                char** ppNewData,
                                                DWORD& rNewDataSize,
                                                bool bHeaderCheck,
                                                DWORD dwContext);
```

This function is called when an incoming request is ready to be inspected so you can decide if you want to do something with it right now (instead of waiting for the download to complete), it is then called again when it contain the full HTTP data (data is in the buffers, the browser didn't receive it yet), the data is parsed, even if the original was Gzip encoding of chunked transfer encoding, you still receive plain text and the returned headers are modified automatically to accommodate for this change.

The parameters are:

pHeader – The pointer containing the header data

dwHeaderSize – Size of the header in bytes.

ppNewHeader – If you need to return data to the Redirector, in case of modify or redirect, you need to allocate the data and give it to this pointer.

rNewHeaderSize – Size of the new data or redirect address.

pData – The pointer containing the actual data (HTML usually)

dwDataSize – Size of the data in bytes.

ppNewData – If you need to return new data to the Redirector, in case of modify or redirect, you need to allocate the data and give it to this pointer.

rNewDataSize - Size of the new data.

bHeaderCheck – This flag is true when this call is with header only and data is still downloading, keep in mind that if the Redirector managed to get the complete reply (header+data) in one call, this method will be called only once (with this flag false). When the flag is false it means that this is the complete reply.

dwContext – DLL-specific context that was created at the NewConnection function.

Return value:

Action to take with this reply, you can look what each value means under the Architecture section.

Sample code taken from the DLL:

```
//Is this a header check (gives us change to determine what we are
receiving, in case it's a big file
//we might not want to process it
//Header check means that we have a complete header, but not all the
data
if (bHeaderCheck)
{
      //Try to parse the header
      CHTTPHeader aHeader;
      if (aHeader.Feed(pHeader,
                       dwHeaderSize))
      {
            //We can get some usefull information at this stage
            //Response code
            DWORD dwResponseCode(aHeader.GetResponseCode());

            //Try to get the content type
            std::string sContentType(aHeader.GetHeaderString("content-
type"));

            //Now there are some content we don't want to perform
parental control on
            //Of course you can change these settings
            //These settings are just for sample sake
            //For these kind of files (.swf, .zip, .exe) filtering
should be performed at the level of HTTPRequestBeforeSend
            if (strstr(sContentType.c_str(),
                       "image/") ||
                strstr(sContentType.c_str(),
                       "video/") ||
                strstr(sContentType.c_str(),
                       "audio/") ||
                strstr(sContentType.c_str(),
                       "application/"))
                  //Don't do parental
                  return dhrDontDoParental;
            }
      }
```

This code snipest checks the reply while we still haven't got the data, and checks the content-type, if it's media files usually we don't need to hold them in buffer (because they are large files mostly) and in case of parental control if they should be rejected it should be done at the stage of HTTPRequestBeforeSend.

Now lets perform a redirect in this stage (after we got all the data):

```
else
{
      //Take the host from the session data
      std::string sHost;

      //Do we have session data?
      if (pContextData)
```

```
            sHost=pContextData->sHost;

      //Uncomment the next command to disable the redirect sample
      //return dhrNothing;

      //At this stage we have a full request, for those we didn't
disable
      //Lets say we want to perform a filtering here
      //We need to make sure that we are not redirecting over and over
      //The site we are redirecting to
      //This code performs it
      if (!strstr(sHost.c_str(),
                  "komodia.com") &&
          !strstr(sHost.c_str(),
                  "google"))
      {
                  //Redirect to address, always in the form of
http://website
            std::string sRedirectTo;
            sRedirectTo="http://www.komodia.com";

            //Save the data
            *ppNewHeader=new char[sRedirectTo.size()+1];
            memcpy(*ppNewHeader,
                   sRedirectTo.c_str(),
                   sRedirectTo.size()+1);

            //Set size
            rNewHeaderSize=sRedirectTo.size()+1;

            //Let Redirector know that this is a redirect
            return dhrRedirect;
      }
}
```