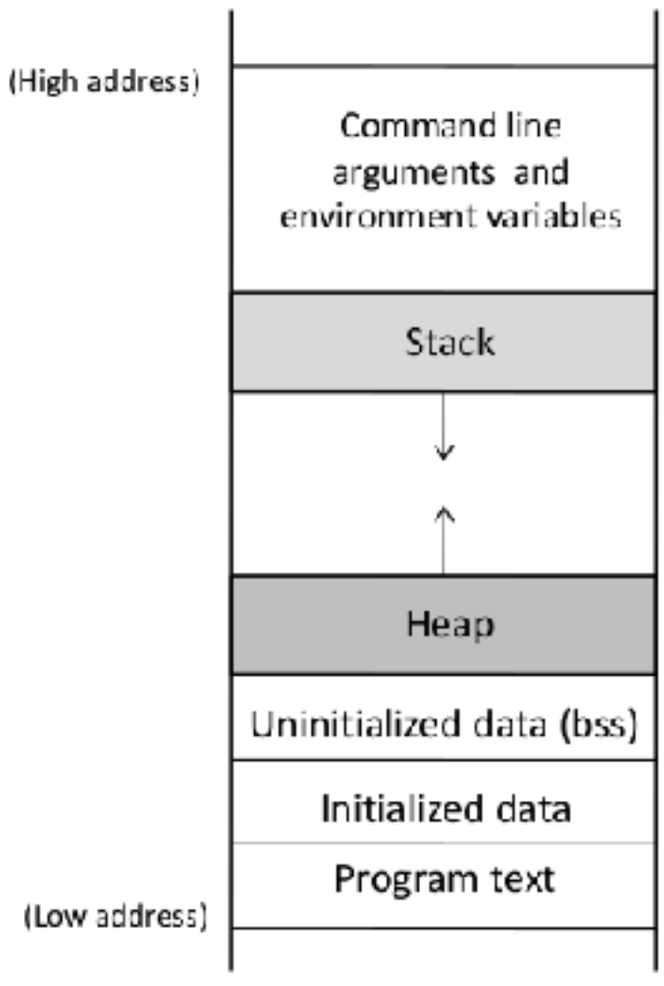
Buffer Overflow Attack

Buffer Overflow Attack and Countermeasures

Buffer Overflow - History

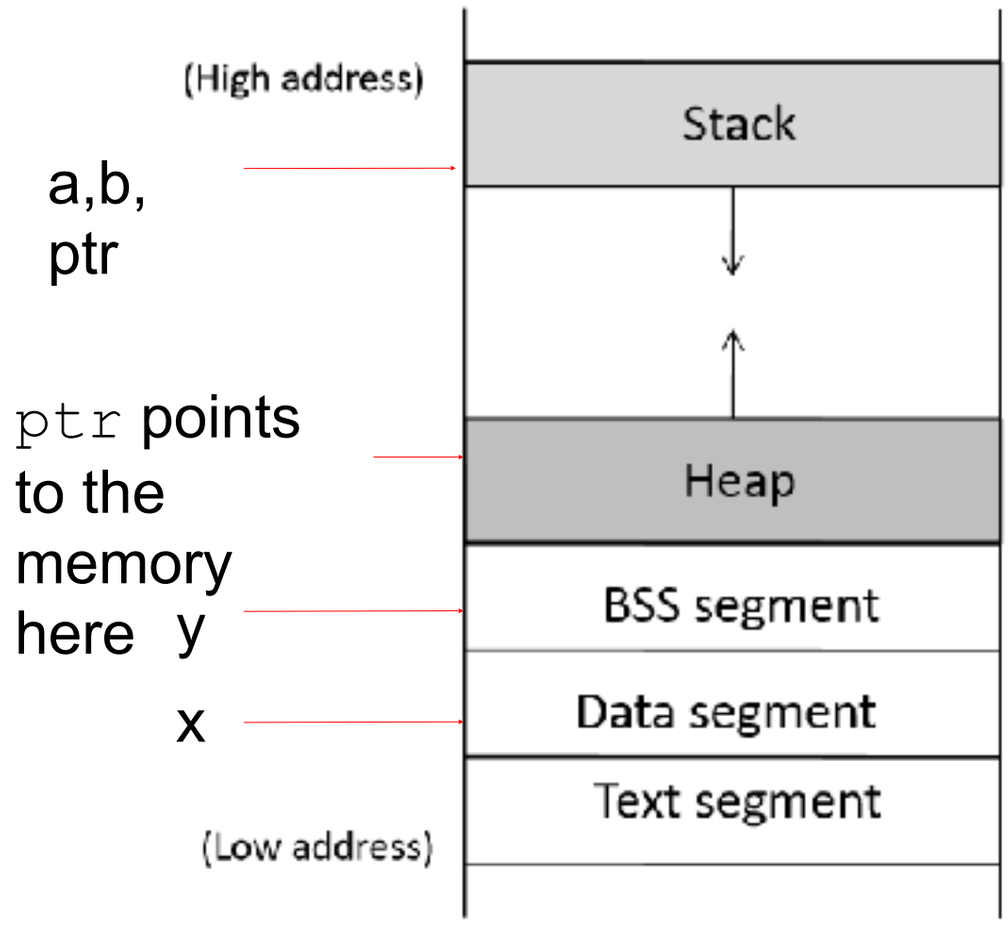
- Documented and Understood since 1972
- First known attack (1988) Morris Worm
    - exploited a Unix service: finger
- 2001: “Code Red Worm” exploited a vulnerability in Microsoft IIS server
- 2003: SQL Slammer compromised servers running Microsoft SQL Server 2000
- 2015: Stagefright attack against Android phones
- ….

Memory Layout



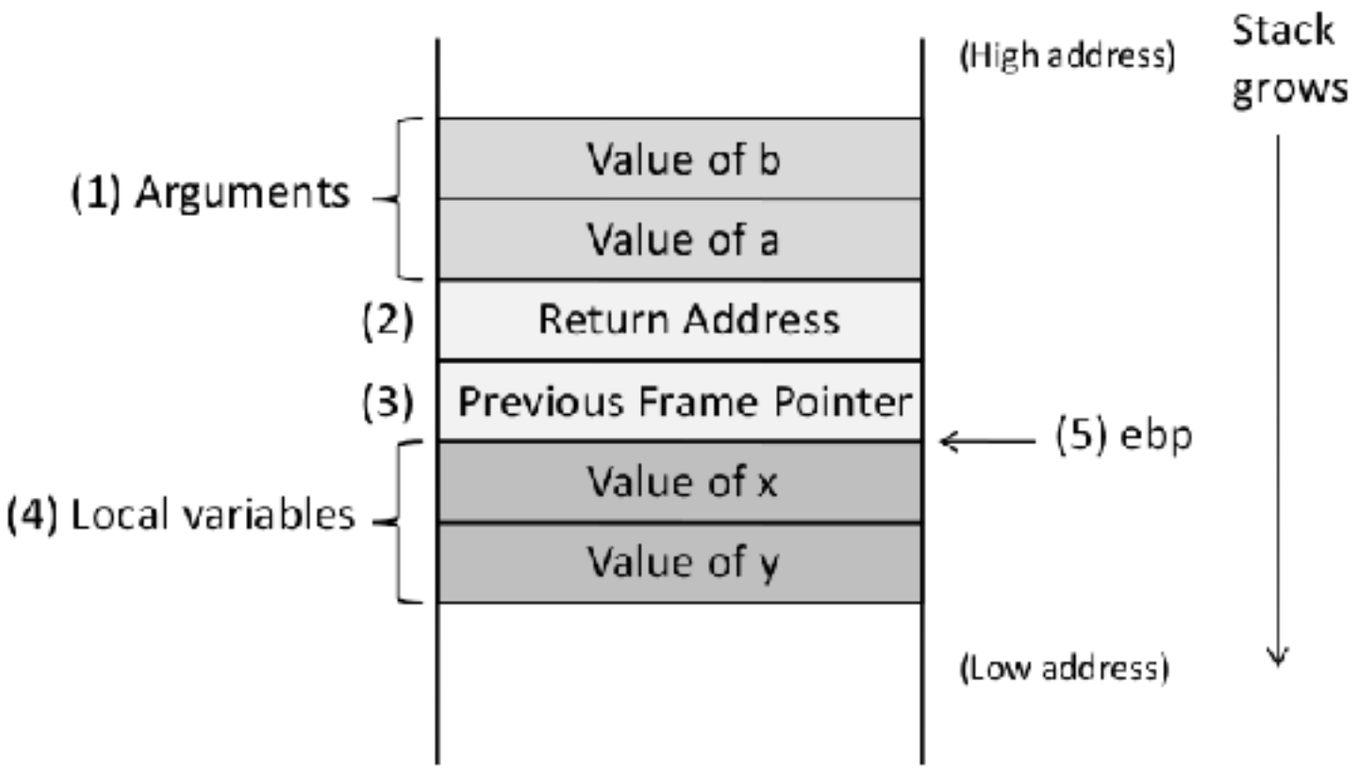
Program Memory Stack

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



Order of the function arguments in stack

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

    x = a + b;
    y = a - b;
}
```



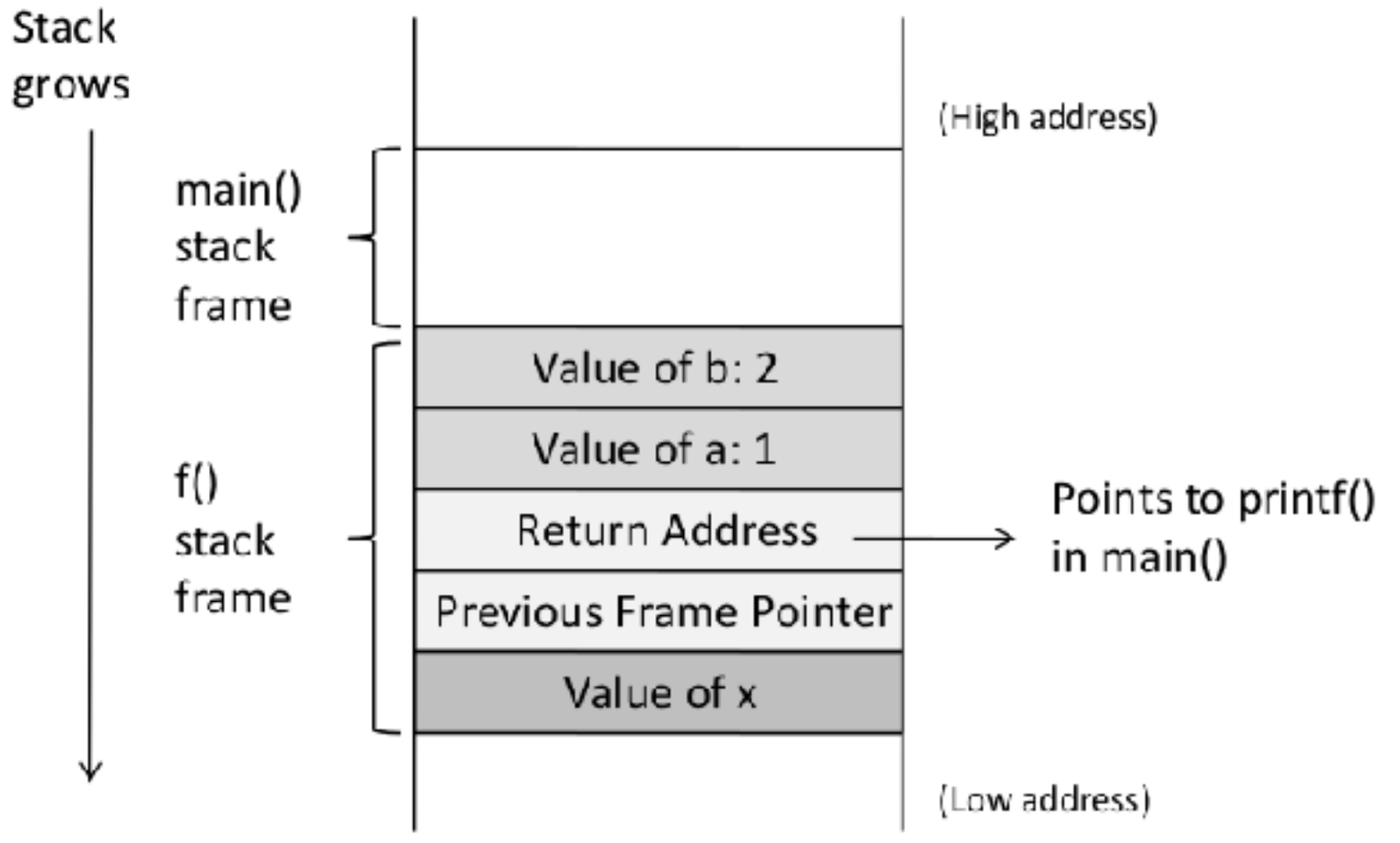

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

How does the compiler know the address of b?

frame pointer -> ebp register

Function Call Chain

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

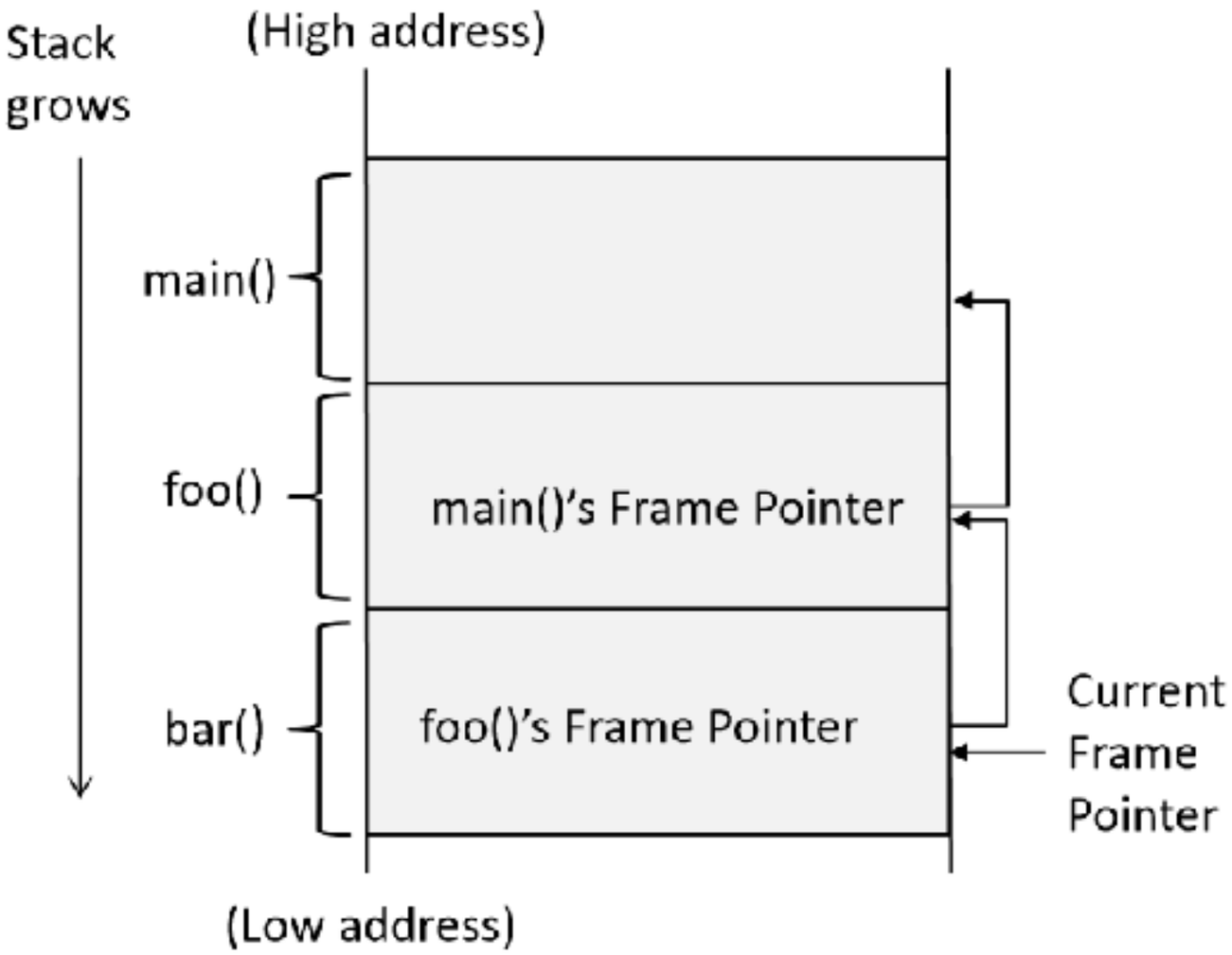


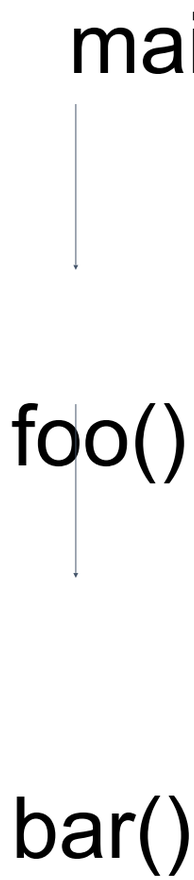
Function Stack Layout

```
void func(char *a1, int b)
{
   int x,y ;
   printf(“address of a1 is %u”, (unsigned int) a1);
   printf(“address of b is %u”, (unsigned int) b);
      …
      g();
      …

}


void main(char *a1, int b)
{
   func(“Hello”, 5);
}
```

What happens if func() calls g()?

Stack Layout for Function Call Chain

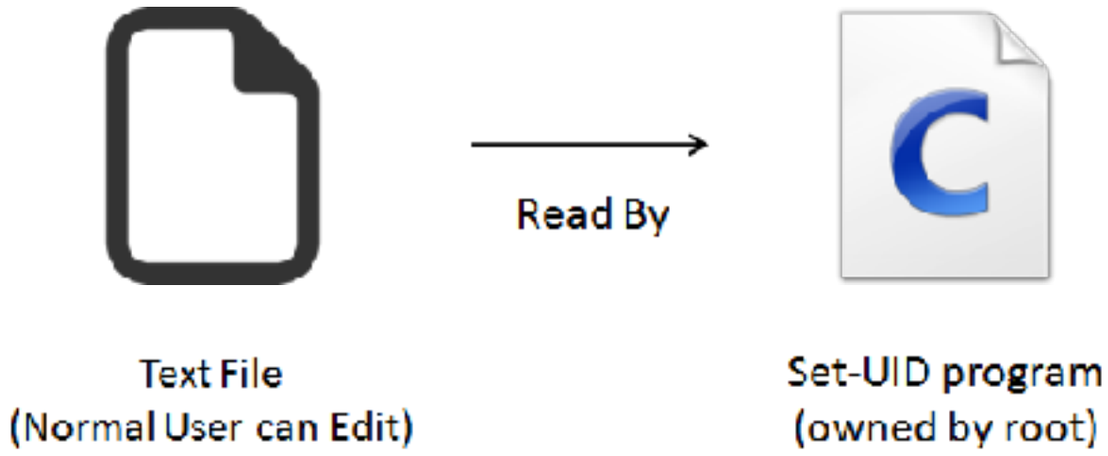




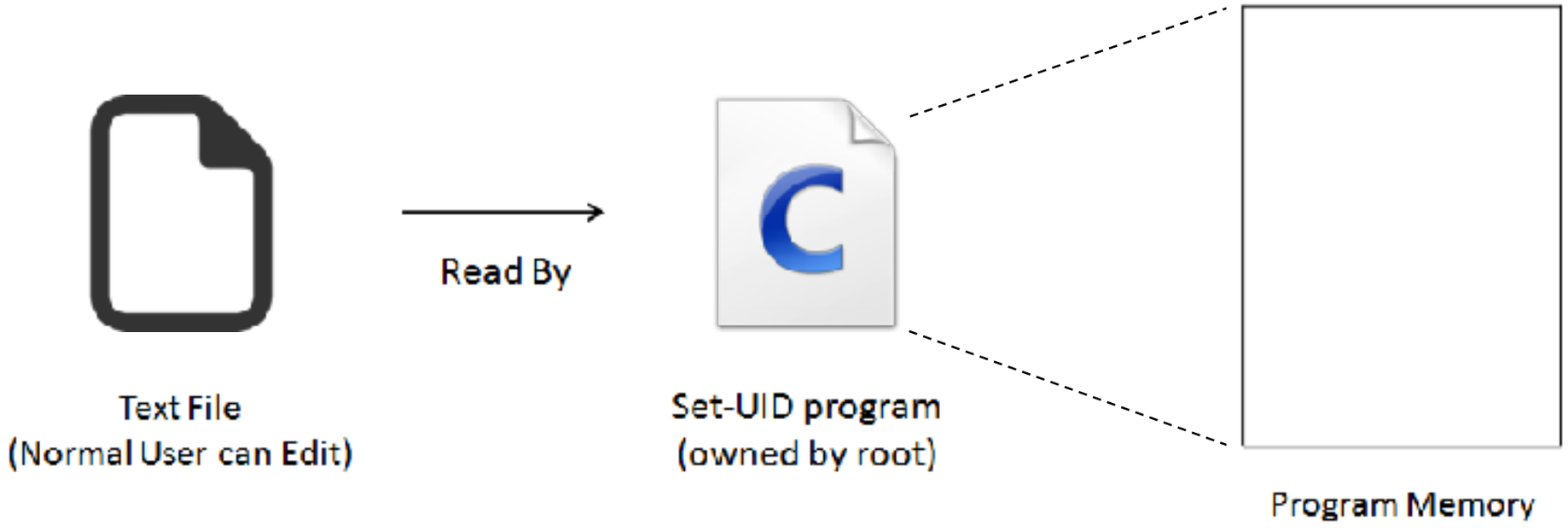
High Level Picture

Set-UID program
(owned by root)

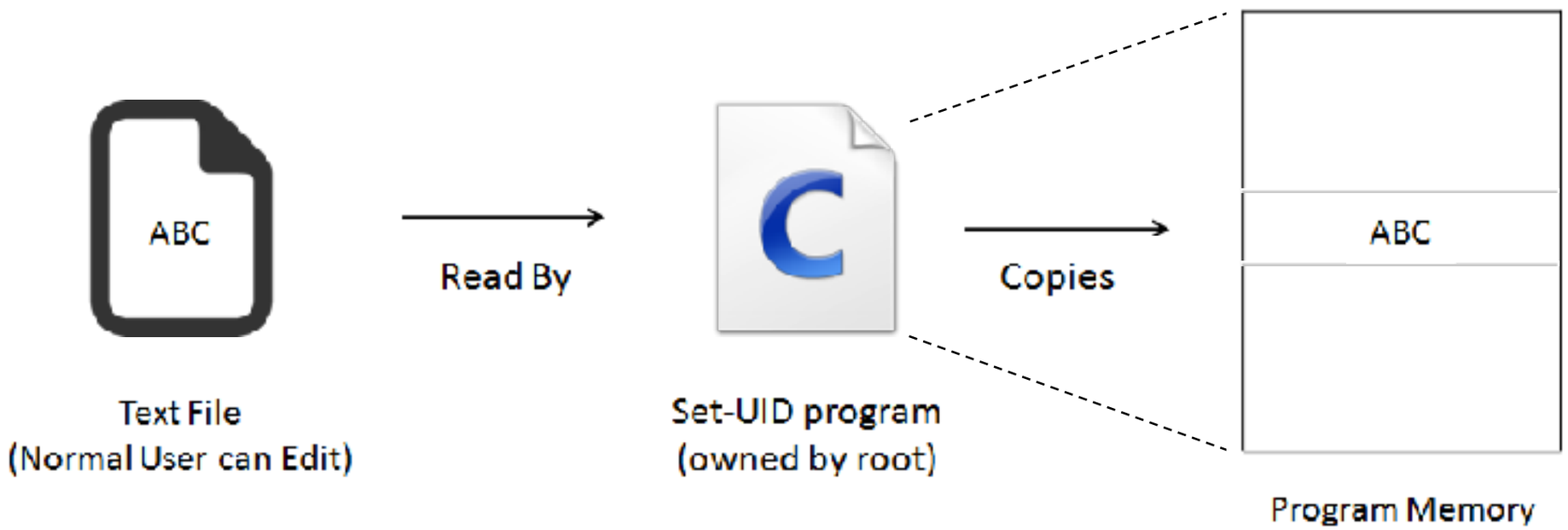
High Level Picture



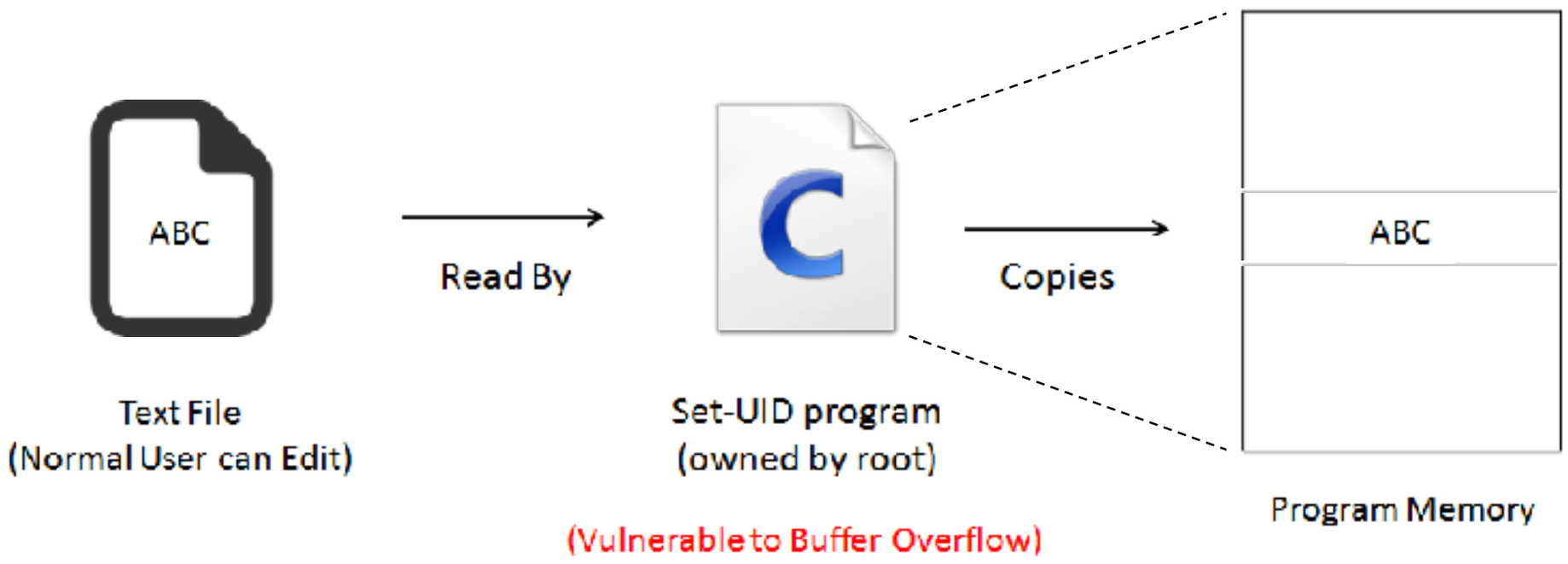
High Level Picture



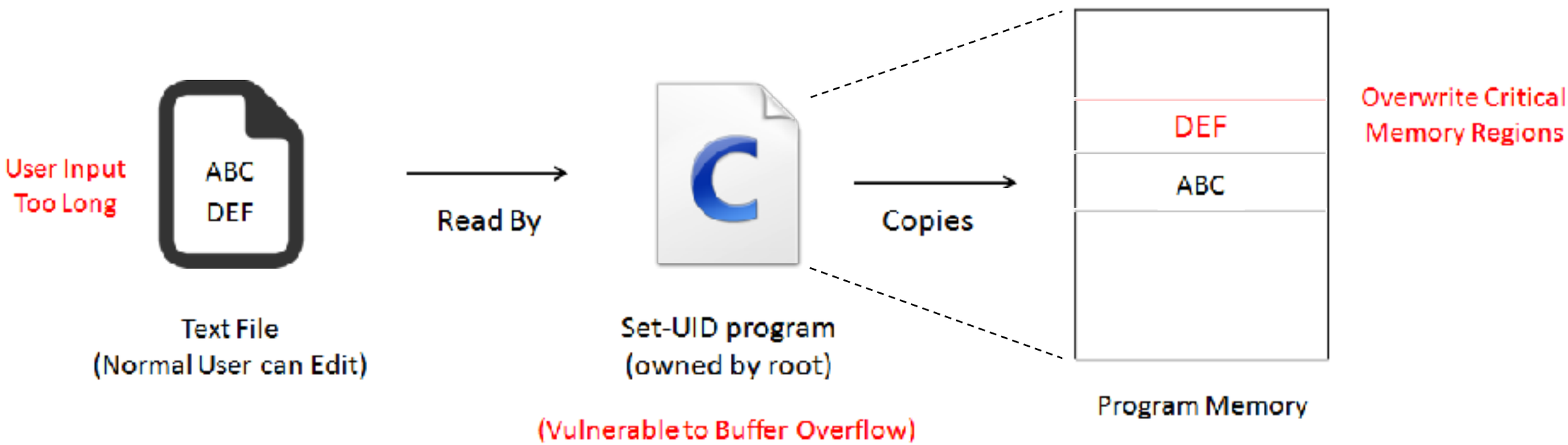
High Level Picture



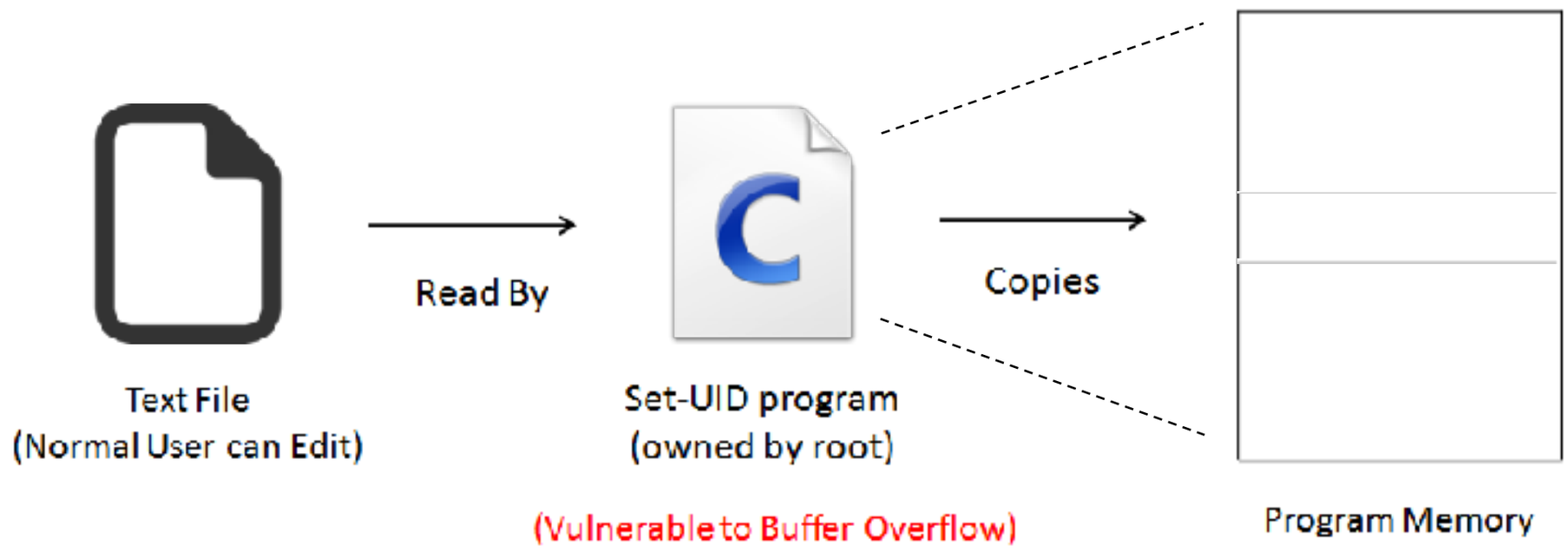
High Level Picture



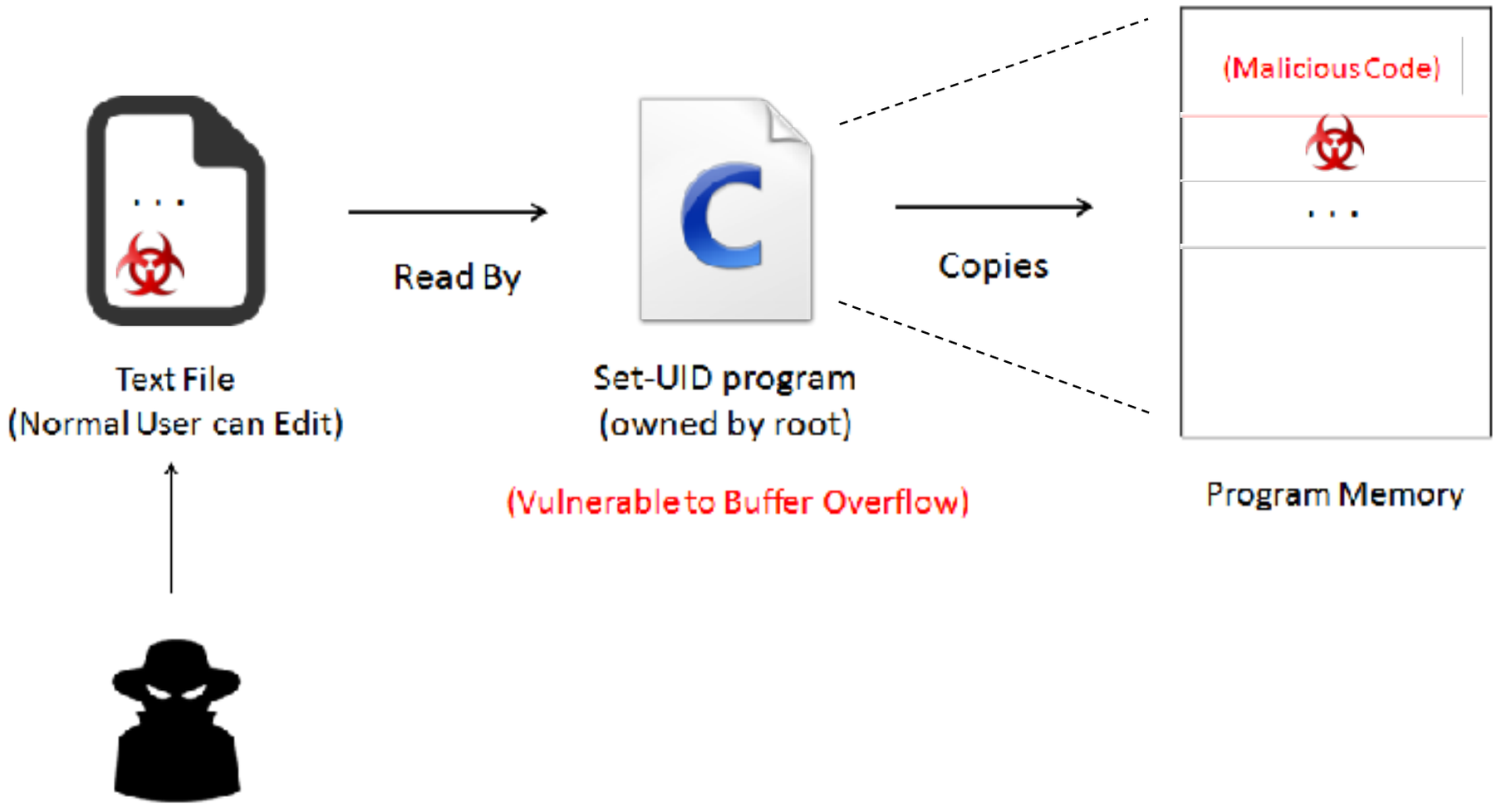
High Level Picture



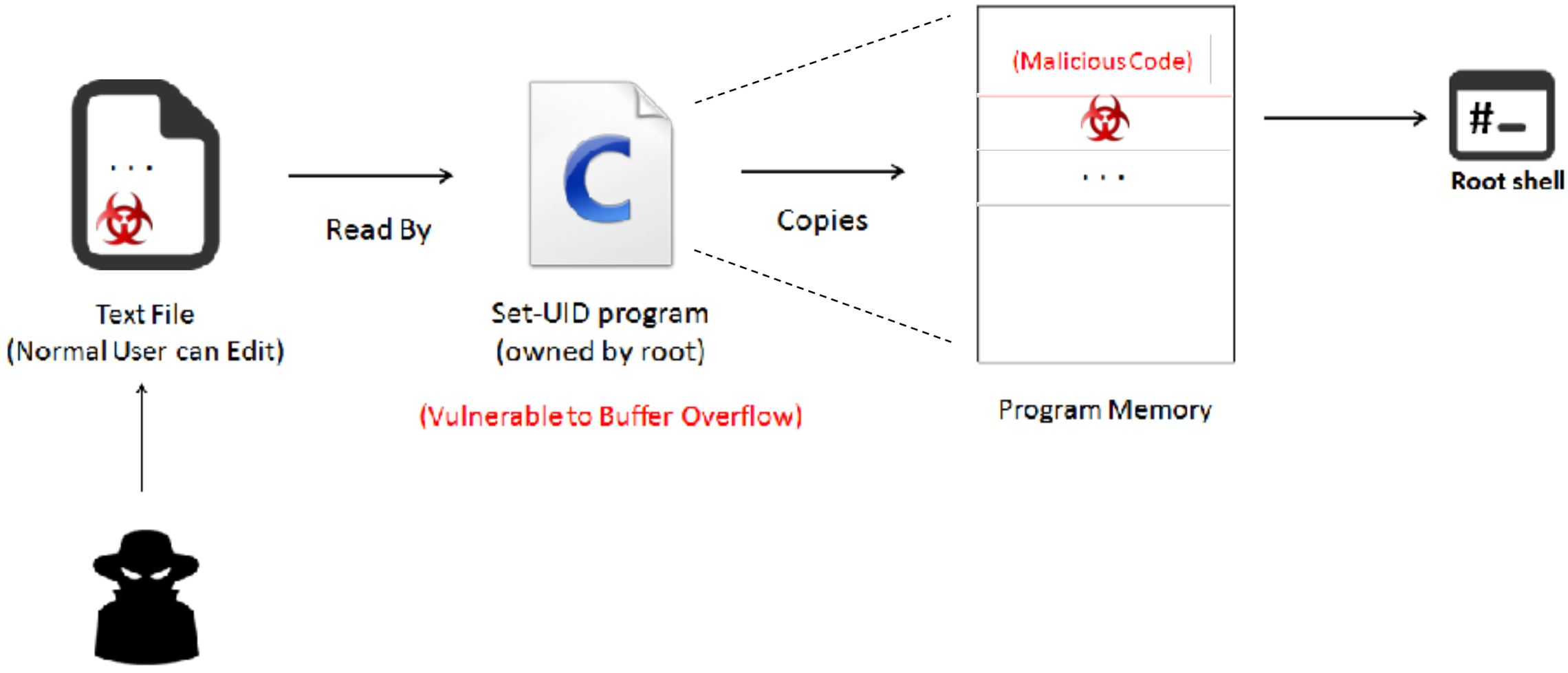
High Level Picture



High Level Picture



High Level Picture



Vulnerable Program

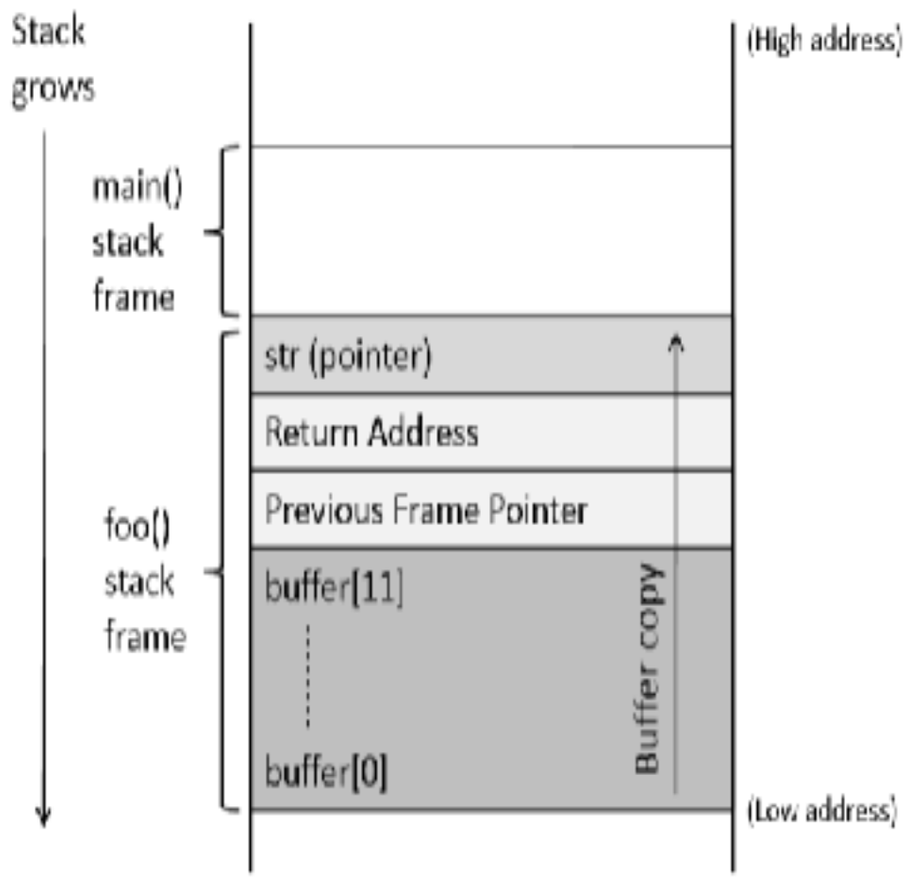
```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

Vulnerable Program

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

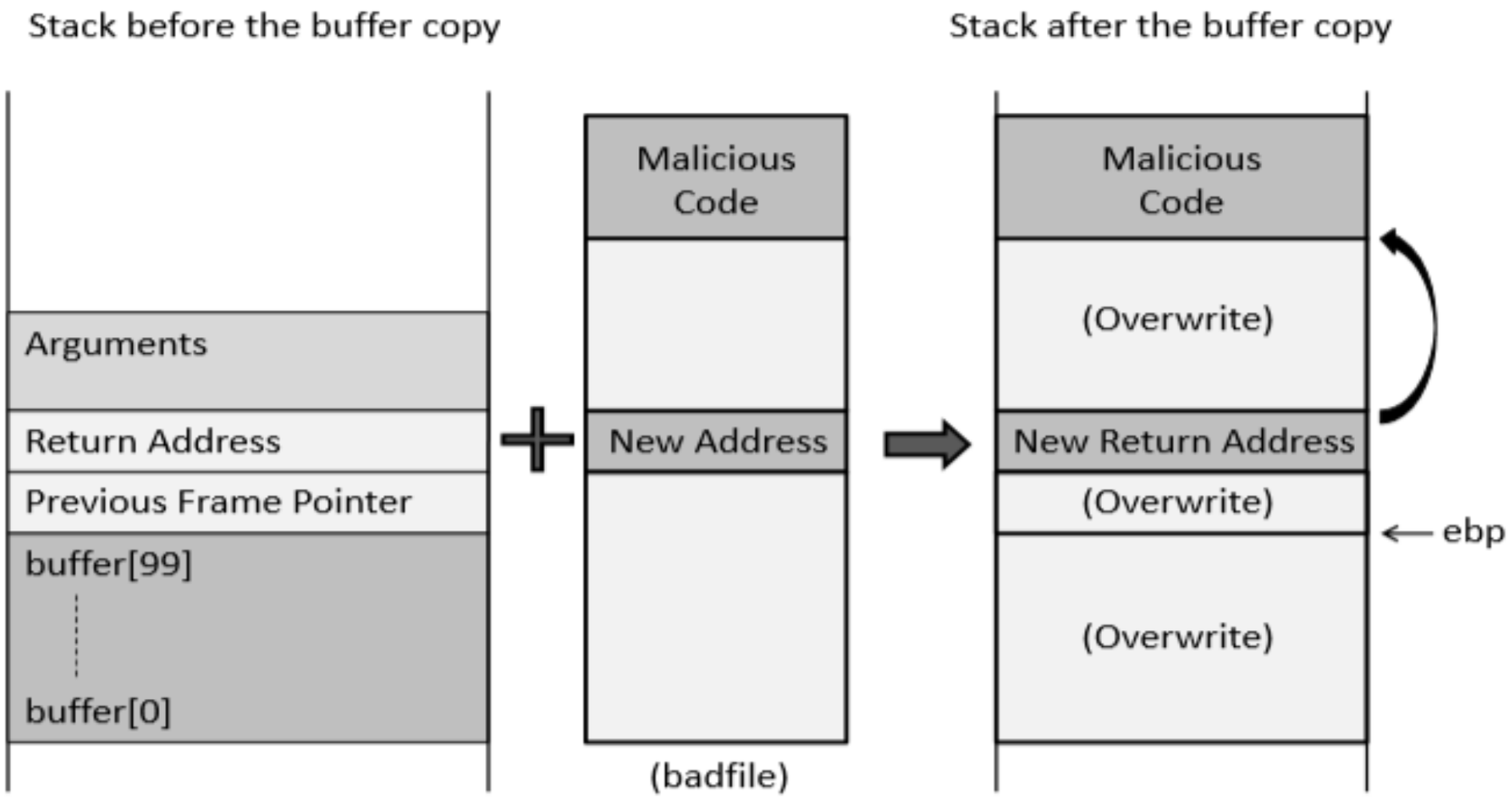


Consequences of Buffer Overflow

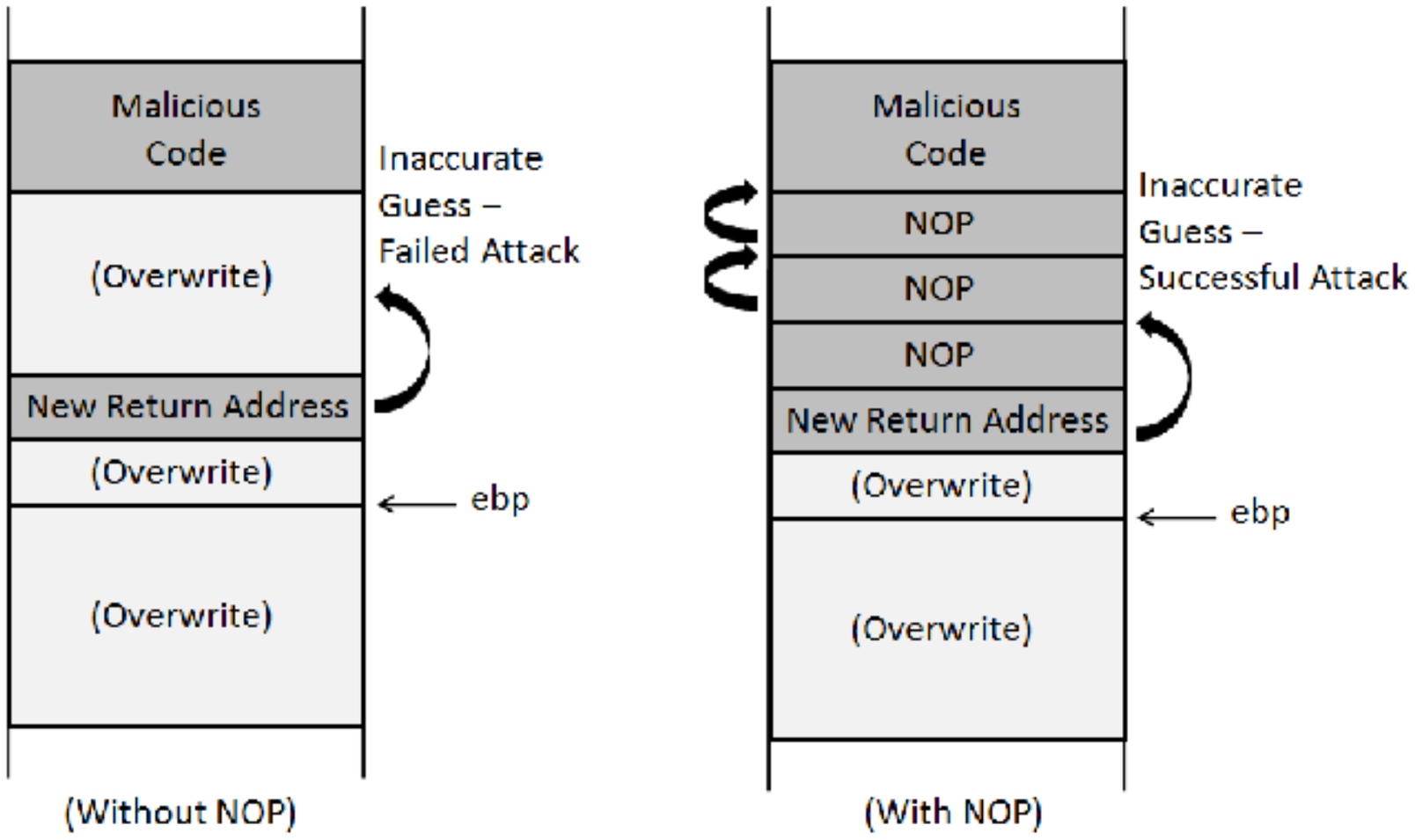
Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- Attacker's code → Malicious code to gain access

How to Run Malicious Code



Use of NOP's


Malicious Code
(Overwrite)
New Return Address
(Overwrite)
(Overwrite)
Inaccurate Guess – Failed Attack
ebp
(Without NOP)
Malicious Code
NOP
NOP
NOP
New Return Address
(Overwrite)
(Overwrite)
Inaccurate Guess – Successful Attack
ebp
(With NOP)

Environment Setup

1. Turn off address randomization (countermeasure)

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

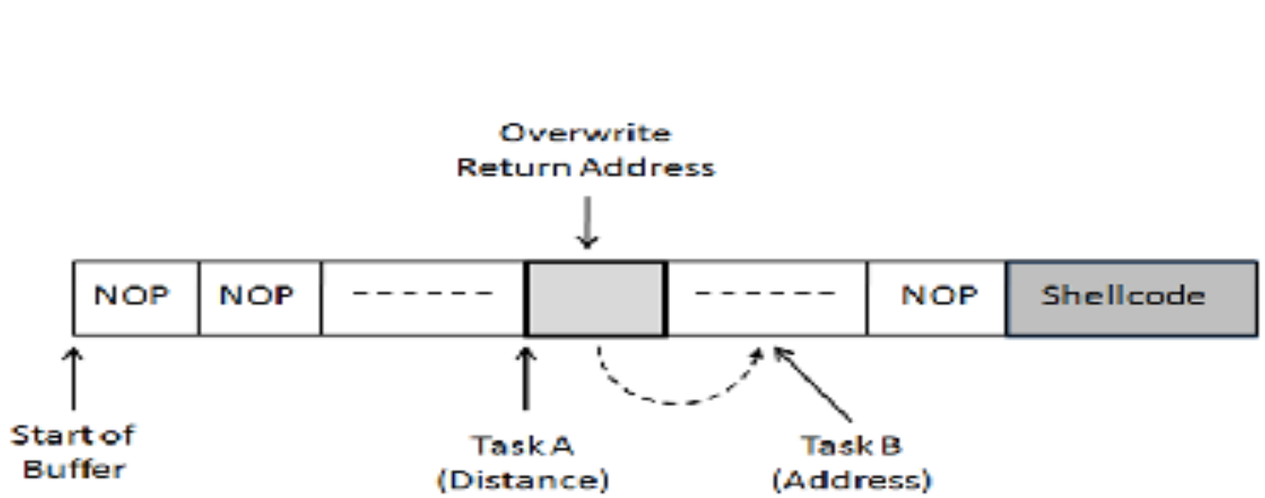
2. Compile set-uid root version of stack.c

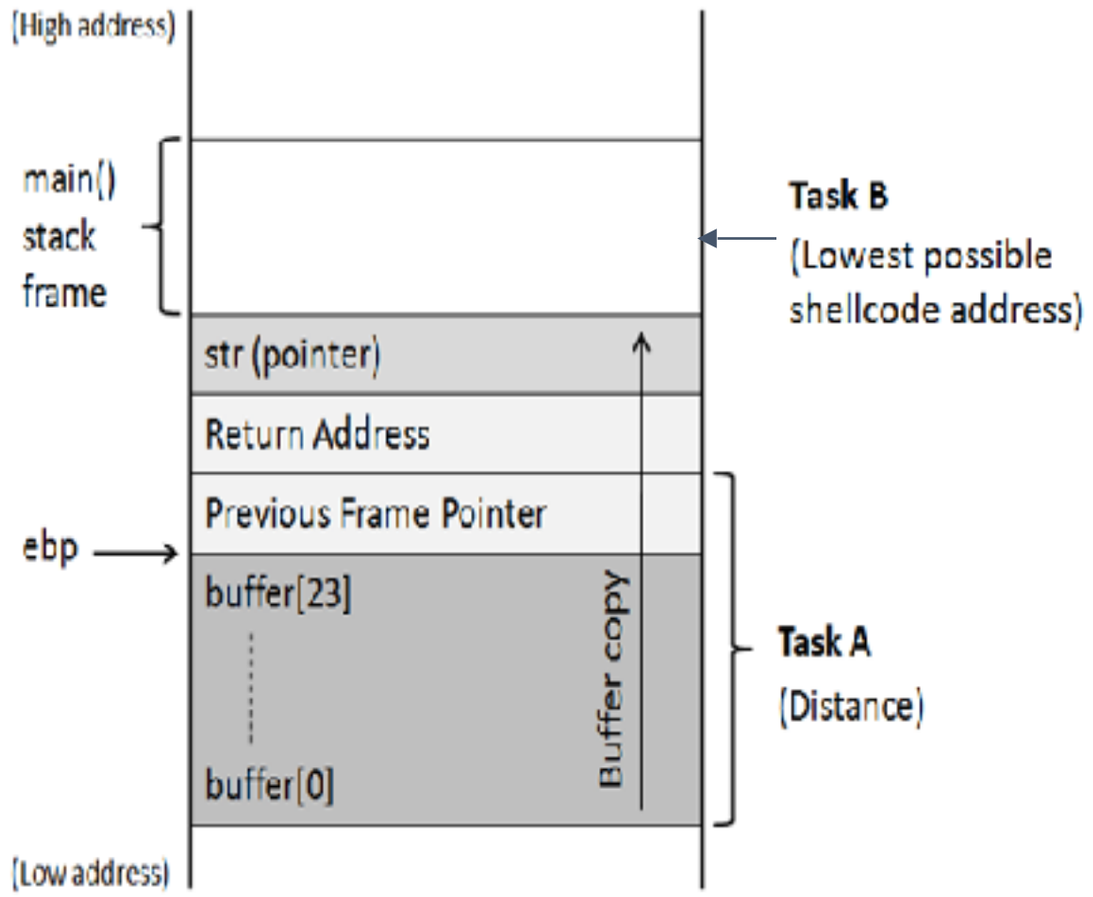
```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.

Task B : Find the address to place the shellcode





Task A : Distance Between Buffer Base Address and Return Address

Using GDB

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 0x02 - 0x01
```

5. Exit (`quit`)

- Breakpoint at vulnerable function using gdb
- Find the base address of buffer
- Find the address of the current frame pointer (ebp)
- Return address is $ebp +4

Task B : Address of Malicious Code

- Investigation using gdb
- Malicious code is written in the badfile which is passed as an argument to the vulnerable function.
- Using gdb, we can find the address of the function argument.

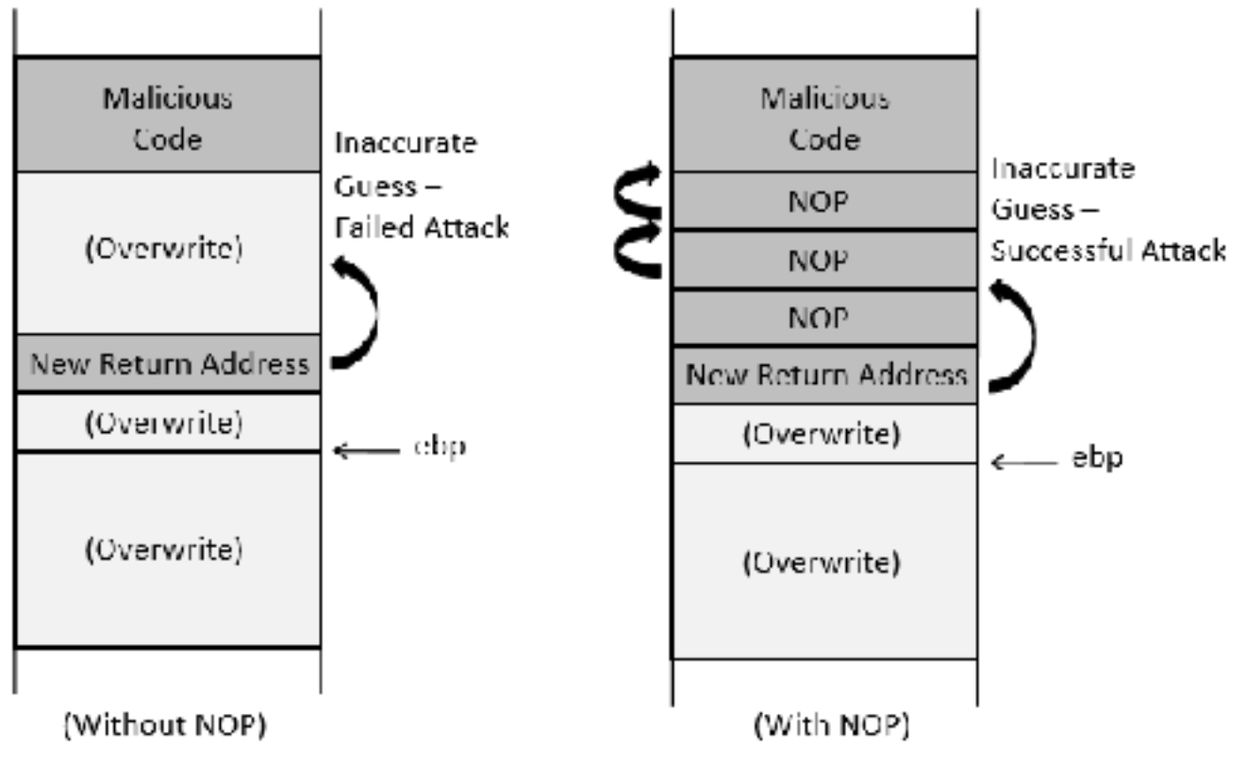
```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

Task B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.



Construct the badfile - exploit.c

```
void main(int argc, char **argv)
{
    // Initialize buffer with 0x90 (NOP instruction)
    memset(&buffer, 0x90, 517);

    // From tasks A and B
    *((long *) (buffer + <distance - task A>)) = <address - task B>;

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

}
```

Countermeasures

- ASLR (refer in lab description)
- StackGuard (refer in lab description)
- Non-Executable (NX) Stack (Next Lab)