Software Security

Ethical Hacking

Alessandro Brighente
Eleonora Losiouk

Master Degree on Cybersecurity

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

Outline

- Memory layout and stack
- Buffer overflow vulnerability
- Exploitation of buffer overflow vulnerability
- Countermeasures

Process memory

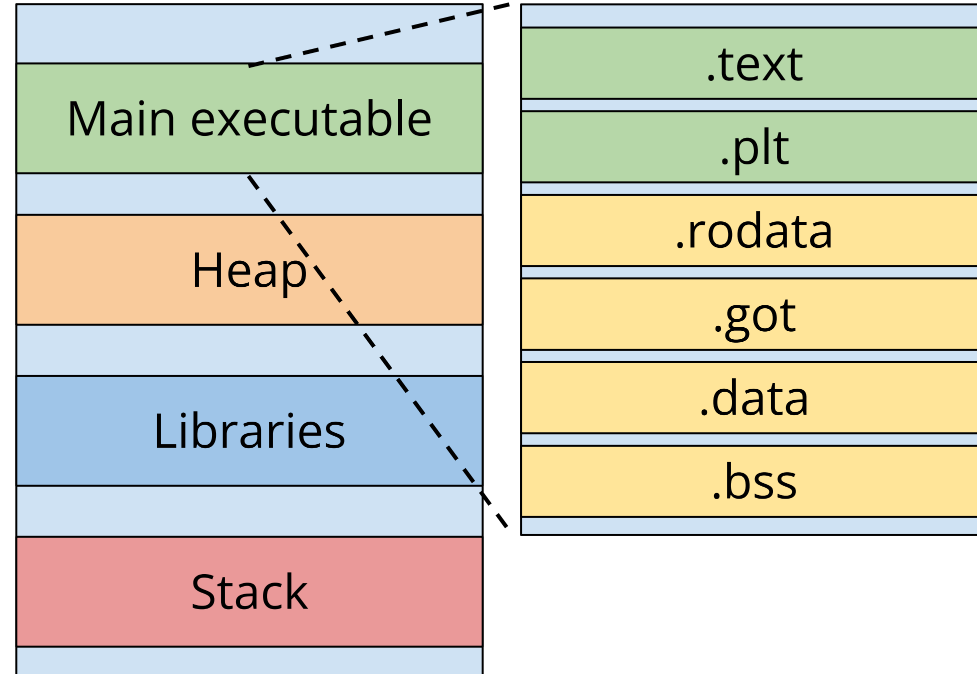
0x0000...

Process memory
Main executable
Heap
Libraries
Stack

Section	Description
.text	Code
.plt	Imports stuff
.rodata	Read-only data
.got	Imports stuff
.data	Read/write data
.bss	Zero-init'ed data

Function Call Stack

```c
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;
}
```



Function Call Stack

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

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

Function Call Stack

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

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

Low addrs

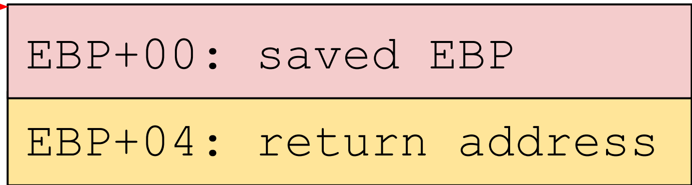
EBP-04: local vars
EBP+00: saved EBP
EBP+04: return address
EBP+08: arguments

High addrs

Function Call Stack

Low addrs

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

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



High addrs

Function Call Stack

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

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

Low addrs

EBP+00: saved EBP
EBP+04: return address

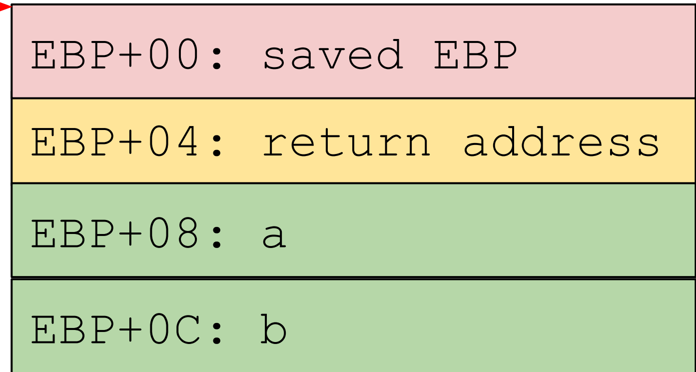
EBP+0C: b

High addrs

Function Call Stack

Low addrs

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

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

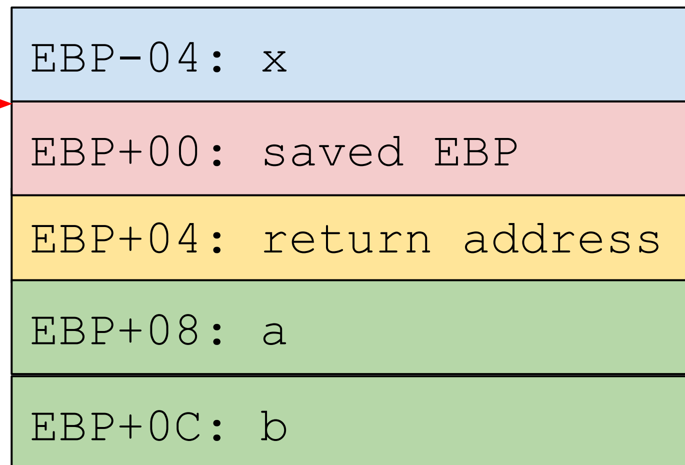


High addrs

Function Call Stack

Low addrs

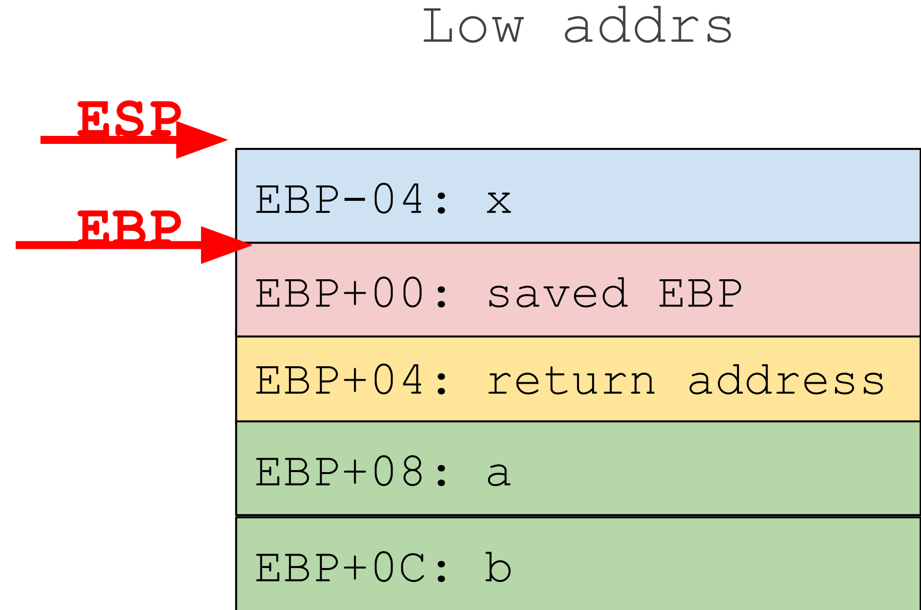
```
void func(int a, int b)
{
    int x, y;

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



High addrs

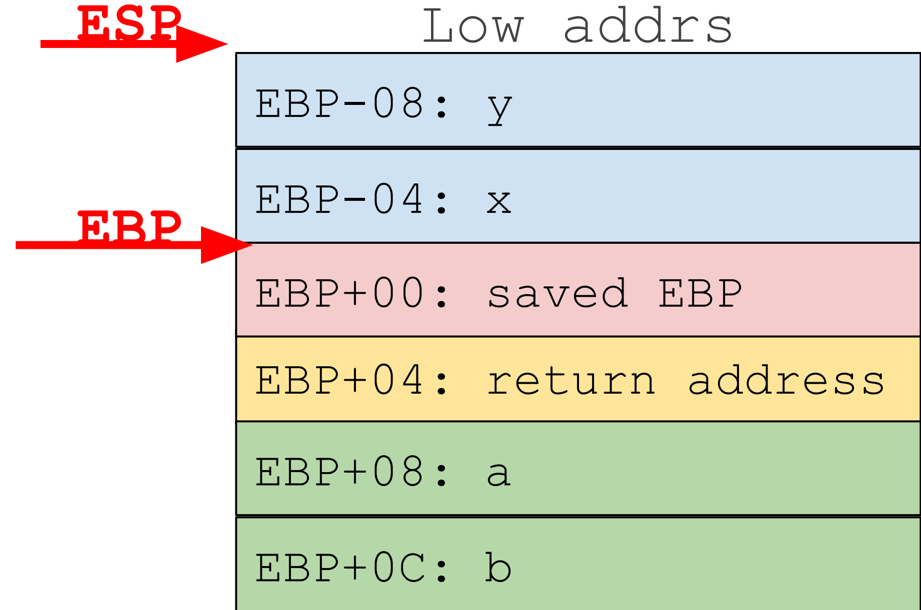
Function Call Stack

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

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



Function Call Stack

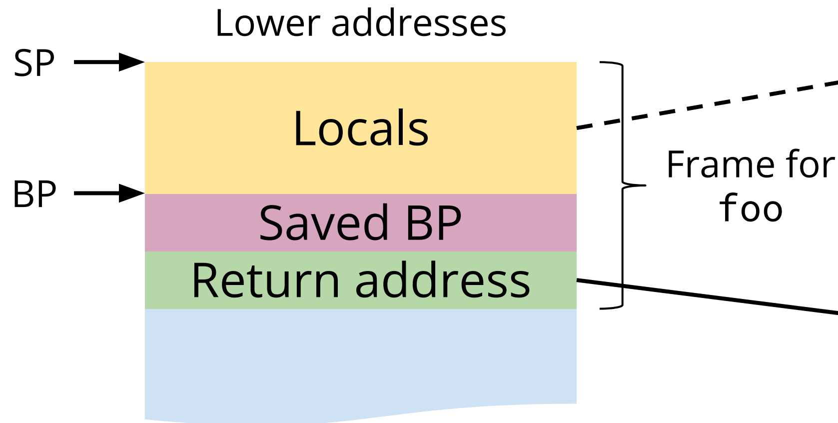
```
void func(int a, int b)
{
   int x, y;

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



The X86 stack

```c
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

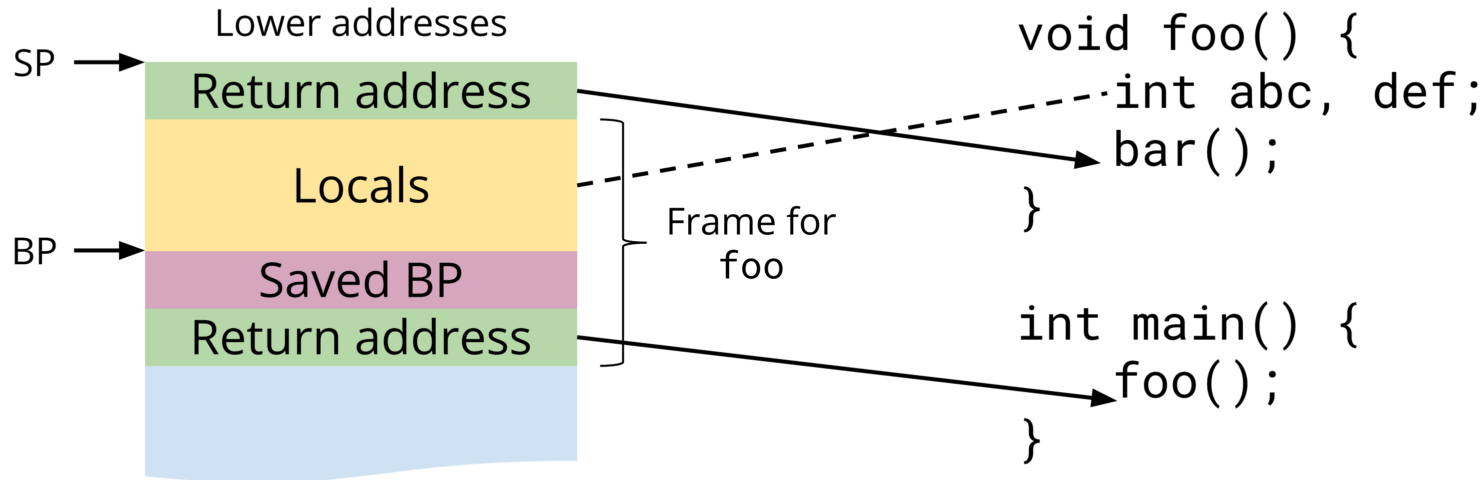
The X86 stack




```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

The X86 stack

```
void bar() {
    char baz[32];
    /* … */
}
```



The X86 stack

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

Lower addresses

SP, BP →

Saved BP
Return address
Locals
Saved BP
Return address

Frame for foo

The X86 stack

Lower addresses

SP →

BP →

Locals
Saved BP
Return address
Locals
Saved BP
Return address

Frame for bar

Frame for foo

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

The X86 stack

Lower addresses

SP, BP →

Saved BP
Return address
Locals
Saved BP
Return address

Frame for foo

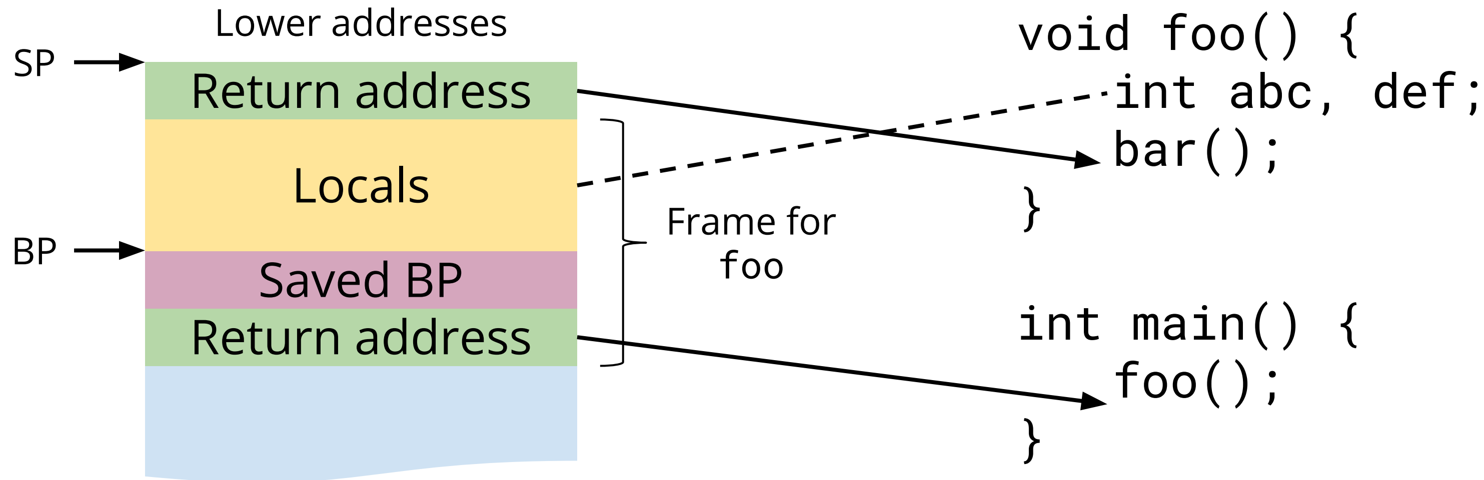
```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

The X86 stack

SPRITZ
Security & Privacy
Research Group

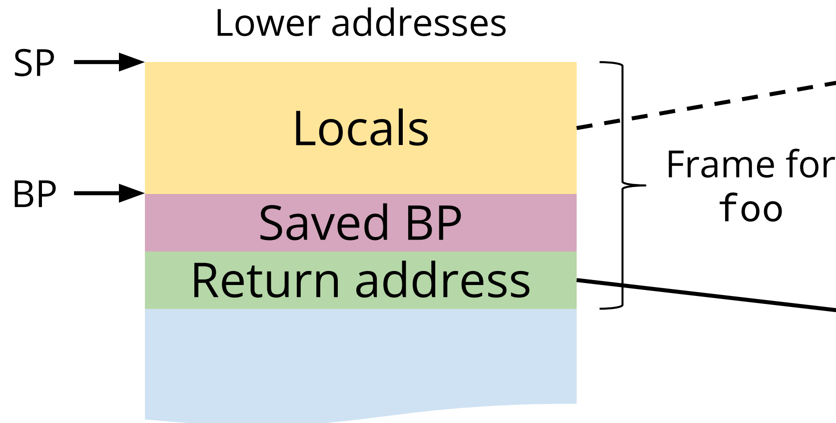
Università
degli Studi
di Padova

```
void bar() {
    char baz[32];
    /* … */
}
```



The X86 stack

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

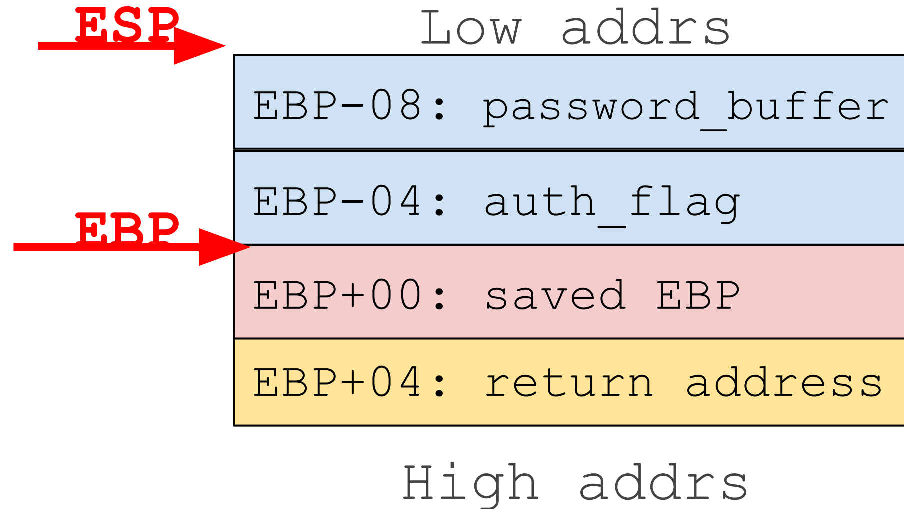


Buffer Overflow Example

```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf("Enter password");
   scanf("%s", password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

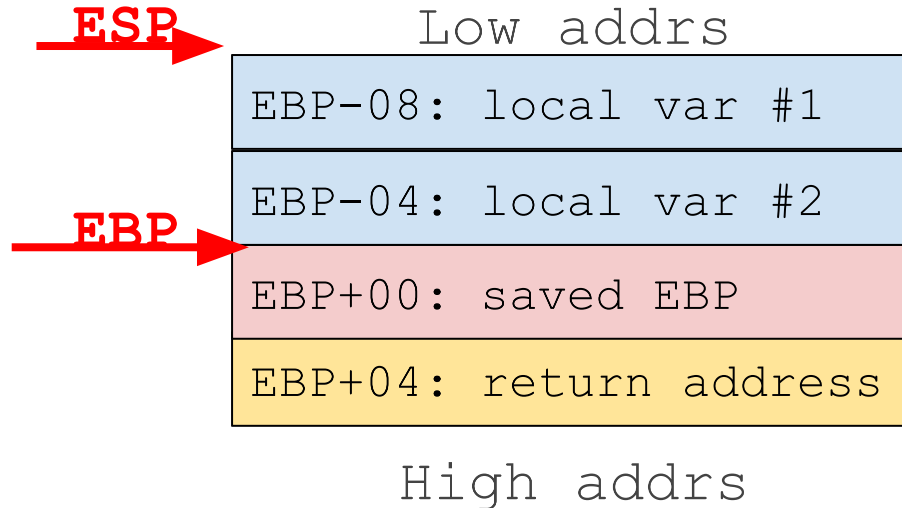
```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf("Enter password");
   scanf("%s", password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

Buffer Overflow Example



Consequences of Buffer Overflow

- Overwriting return address with some random address can point to :
  - Invalid instruction
  - Non-existing address
  - Access violation
  - Attacker's code



How to Run Malicious Code

Stack before the buffer copy

Arguments
Return Address
Previous Frame Pointer
buffer[99]
buffer[0]

\+

Malicious
Code
New Address
(badfile)

Stack after the buffer copy

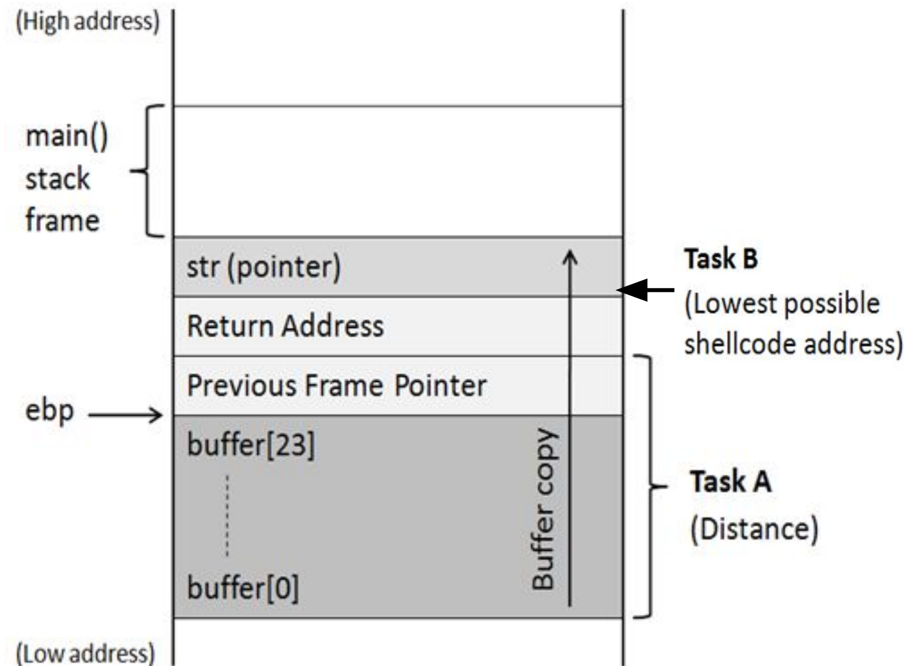
Malicious
Code
(Overwrite)
New Return Address
(Overwrite)
← ebp
(Overwrite)

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

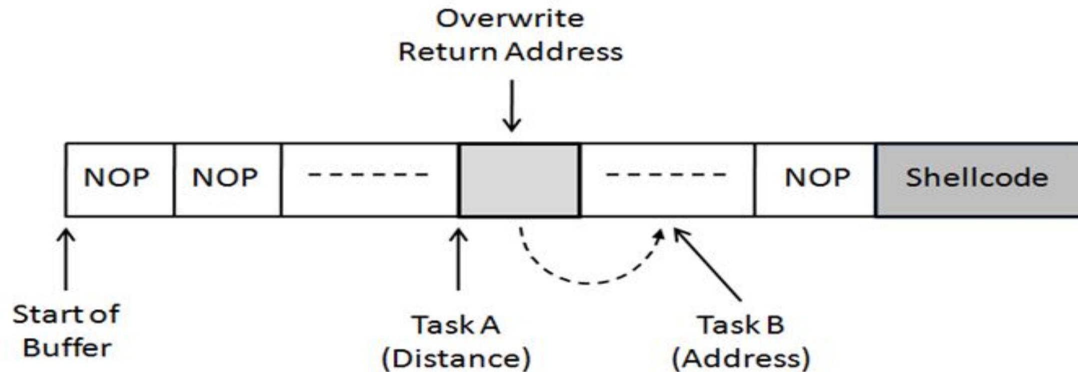
Creation of The Malicious Input (badfile)

- **Task A :** Find the offset distance between the base of the buffer and return address.
- **Task B :** Find the address to place the shellcode



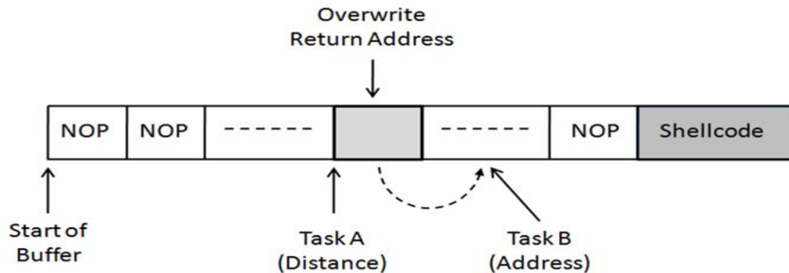
Creation of The Malicious Input (badfile)

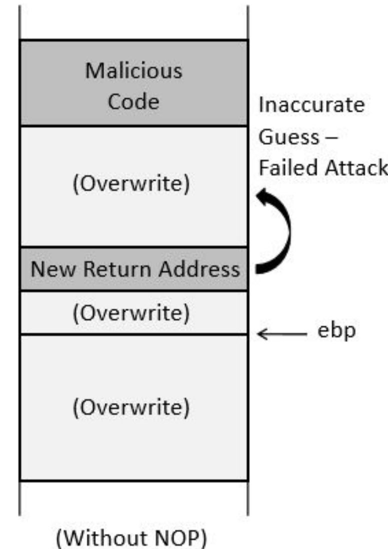
- **Task A :** Find the offset distance between the base of the buffer and return address.
- **Task B :** Find the address to place the shellcode

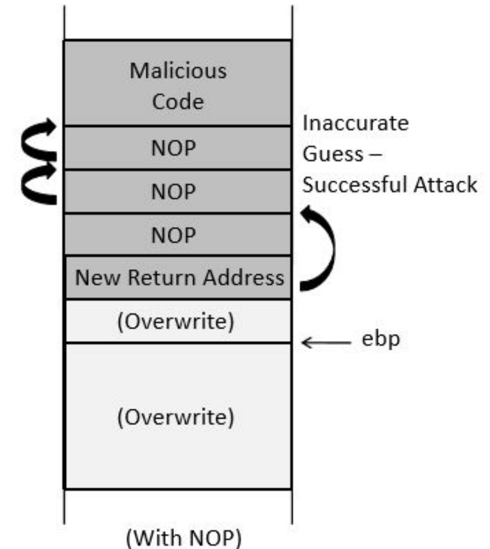


Creation of The Malicious Input (badfile)

- **Task A :** Find the offset distance between the base of the buffer and return address.
- **Task B :** Find the address to place the shellcode







New Address in Return Address

- 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.

Countermeasures Overview

- Developer approach
- OS approach
- Compiler approach

Developer Approaches

- Check the data length
- Use protected functions

```
char *strcpy(char *dest, const char *src)
char *strncpy(char *dest, const char *src, size_t n)

char *strcat(char *dest, const char *src)
char *strncat(char *dest, const char *src, size_t n)

int sprintf(char *str, const char *format, ...)
int snprintf(char *str, size_t size, const char *format, …);

char *gets(char *str)
char *fgets(char *str, int n, FILE *stream)
```

Developer Approaches

- Check the data length
- Use protected functions
- Use safer libraries (e.g., libsafe)

Developer Approaches

- Check the data length
- Use protected functions
- Use safer libraries (e.g., libsafe)
- Use safer languages (e.g., Java)

Compiler Approach: Stack Canaries

The program copies the user's input to a fixed size 32-byte stack buffer

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Canary	11	22	33	44	55	66	77	88
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

The program copies the user's input to a fixed size 32-byte stack buffer

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Canary	11	22	33	44	55	66	77	88
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Input: 54 'A'

41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	00	00

Compiler Approach: Stack Canaries

The program copies the user's input to a fixed size 32-byte stack buffer

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Canary	11	22	33	44	55	66	77	88
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Input: 54 'A'

41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	00	00

Doesn't return to `0x414141414141`

Compiler Approach: Stack Canaries

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

    int guard;
    guard = secret;
    char buffer[12];
    strcpy (buffer, str);

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

Compiler Approach: Stack Canaries

```
// Canary Set Start
movl %gs:20, %teax
movl %eax, -12(%ebp)
xorl %teax, %teax

// Canary Set End
movl -28(%ebp), %eax
movl Seax, 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
```

Bypassing stack canaries

- Problem: we don’t know the secret canary
- The canary is generated randomly at program startup, but it’s constant within a run!
- If we leak the canary at any point in time, we know it for the whole program’s lifetime