Secure Software Design and Engineering
(CY-321)

Buffer Overflow

Dr. Zubair Ahmad

A kind Reminder

- **Attendance?**

  - Active Attendance
  - **Dead Bodies.**
  - **Active Minds**
  - Mobiles in hands -> Mark as absent
  - 80% mandatory

Buffer Overflow

A buffer is a region of memory that is used to store data.

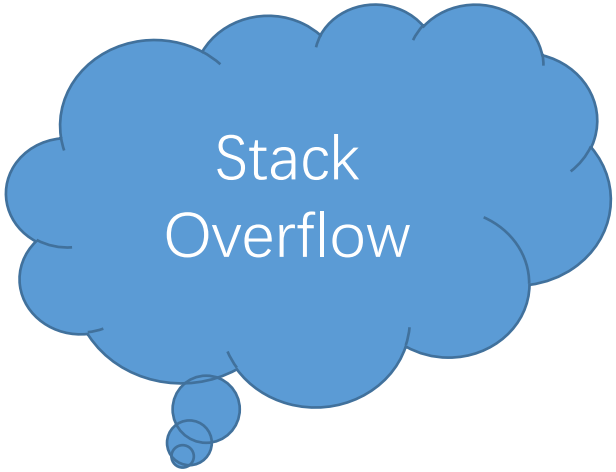
Buffers are usually unstructured: dont contain objects, records, integers or other structured data, but merely bytes

Buffer space is usually needed only temporarily.

Buffers are most often used during I/O (reading or writing).

A buffer overflow happens when data is written beyond the end of the buffer.

Buffer Overflow



Why Buffer Overflow happens?

- Copying of data into the buffer without checking the size of input

```
#include <string.h>

void copy me(const char *s) {
    char copy[1024];
    strcpy(copy, s);
}
```

Why Buffer Overflow happens?

- Copying of data into the buffer without checking the size of input

```
#include <stdio.h>
#define BUFFER_SIZE 1024

void fill_buffer(char *buf, FILE *fp)
{
    fread(buf, 1, BUFFER SIZE, fp);
    if (!ferror(fp))
        buf[BUFFER SIZE] = '\0'; /*
Must null-terminate string */
}

10

void f() {
    char buf[BUFFER SIZE];
    fill buffer(buf, stdin);
}
```

Why Buffer Overflow happens?

- Accessing the buffer with incorrect length values

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

void unsafe_copy() {
    char buffer[10];
    printf("Enter a string: ");
    gets(buffer);
    printf("You entered: %s\n",
buffer);
}

int main() {
    unsafe_copy();
    return 0;
}
```

Why Buffer Overflow happens?

- Improper validation of array (simplest expression of a buffer) index

```
#include <stdio.h>

void unsafe_access() {
    int arr[5] = {1, 2, 3, 4, 5};
    int index;

    printf("Enter an index (0-4): ");
    scanf("%d", &index);

printf("Array element at index %d:
%d\n", index, arr[index]);
}

int main() {
    unsafe_access();
    return 0;
}
```

Why Buffer Overflow happens?

- Incorrect calculation of buffer size before its allocation

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

void unsafe_allocation() {
    char input[10];

    printf("Enter a string: ");
    fgets(input, 50, stdin);

    printf("You entered: %s\n",
input);
}

int main() {
    unsafe_allocation();
    return 0;
}
```

Why Buffer Overflow happens?

- Integer overflows or wraparounds

```
#include <stdio.h>

int main() {
    unsigned char x = 255;
    printf("Initial value of x: %u\n",
x);

    // Adding 1 will cause an overflow
    x = x + 1;

    printf("Value of x after overflow:
%u\n", x); return 0;
}
```

So Whats The Deal?

First, note that there is data as well as control flow information on the stack

If there is an automatic variable there that we can overflow, maybe we can overwrite the return address

If we can overwrite the return address, we can (usually) make the program execute code anywhere in the process address space

And if the variable we have overflowed is a buffer, why not fill the buffer with our code to execute?

This is what is meant **“allows execution of arbitrary code”**

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated

Making an Exploit

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

void vulnerableFunction(char *input) {
    char buffer[16];
    strcpy(buffer, input);
    printf("Received: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        vulnerableFunction(argv[1]);
    } else {
        printf("Usage: %s <input>\n",
argv[0]);
    }

    return 0;
}
```

Explanation of the Vulnerability

The function **vulnerableFunction** defines a local buffer char **buffer[16];** that can hold up to 16 characters.

The function uses **strcpy** to copy the contents of the input (provided by the user via argv[1]) into the buffer. However, **strcpy** does not check the length of the input, so if the input is larger than 16 characters, it will overflow the buffer and overwrite adjacent memory on the stack.

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

void vulnerableFunction(char *input) {
    char buffer[16];
    strcpy(buffer, input);
    printf("Received: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        vulnerableFunction(argv[1]);
    } else {
        printf("Usage: %s <input>\n",
argv[0]);
    }

    return 0;
}
```

Potential Exploitation

When more than 16 characters are passed to the program as an argument, the `strcpy` function will copy the excess characters into memory locations beyond the buffer. This can overwrite the **return address** or other critical data on the stack, potentially allowing the attacker to:

- Execute arbitrary code (e.g., shellcode placed by the attacker).
- Redirect the program's execution flow to malicious code (e.g., jumping to the attacker's payload).

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

void vulnerableFunction(char *input) {
    char buffer[16];
    strcpy(buffer, input);
    printf("Received: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        vulnerableFunction(argv[1]);
    } else {
        printf("Usage: %s <input>\n",
argv[0]);
    }

    return 0;
}
```

Exploitation Example

```
./vulnerable_program AAAAAAAAAAAAAAAAAA
```

Here, the input string has 18 A characters, which exceeds the size of buffer (16 characters)

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

void vulnerableFunction(char *input) {
    char buffer[16];
    strcpy(buffer, input);
    printf("Received: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        vulnerableFunction(argv[1]);
    } else {
        printf("Usage: %s <input>\n",
argv[0]);
    }

    return 0;
}
```

Common Exploit Methods

Stack Smashing: By overflowing the buffer and overwriting the return address, the attacker can cause the program to jump to a location where they have injected malicious code (such as shellcode).

Return-to-libc Attacks: In the absence of executable stack protections (like DEP - Data Execution Prevention), an attacker might use a return-to-libc attack. Instead of injecting shellcode, they may overwrite the return address with the address of a function like system() in the C standard library to execute arbitrary commands

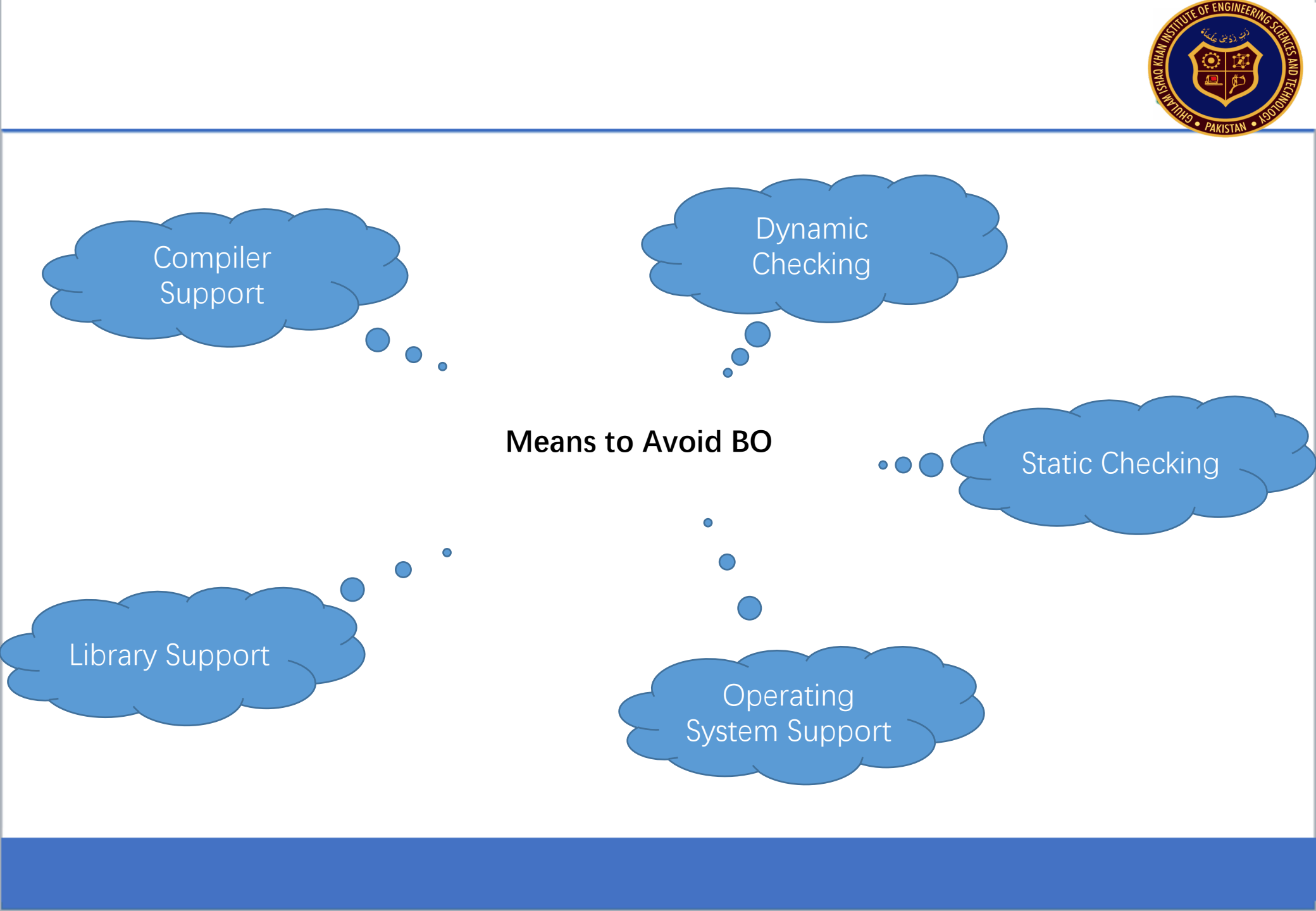
```
#include <stdio.h>
#include <string.h>

void vulnerableFunction(char *input) {
    char buffer[16];
    strcpy(buffer, input);
    printf("Received: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        vulnerableFunction(argv[1]);
    } else {
        printf("Usage: %s <input>\n",
argv[0]);
    }

    return 0;
}
```

Can it be a coincidence that Buffer Overflow and Body Odor have the same acronym? After all, they are very much alike: it happend, but nobody wants it.


Compiler Support
Dynamic Checking
Means to Avoid BO
Static Checking
Library Support
Operating System Support

Compiler Support

Products like **MemGuard** protect the return address on the stack by writing certain random values (**canaries**) on the stack in front of the return address and checking whether these values have changed just before returning.

- Large performance impact
- An attacker that can guess a canary value can attack the system
- Doesnt help against buffer overflows that dont overwrite the return address

Operating System Support

Memory on the stack can be marked as non-executable by the Memory Management Unit (MMU), under control by the operating system.

When the processor fetches an instruction from a memory location that is not OK for execution, it raises a trap. The instruction is not executed

Helps, but not totally: The attacker can perhaps not execute arbitrary code that he wrote, but he can still execute arbitrary code that is already in the application, because he can still jump to any location in the code.

Library Support

One weakness of C is that strings carry no intrinsic length.

Therefore, operations like `memcpy` or `strcpy` can write beyond the end of the string.

That cant be changed, but libraries can find the extent of the stack frame the variable is in and refuse to copy more bytes than are in the stack frame.

Pros: works for a large class of stack smashing attacks.

Cons: buffer overflows happen not only through library functions (though most do); performance impact; removes the symptoms, not the illness.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

A tool decomposes the source text into tokens and looks for patterns

Another tool computes slices to check dependencies between variables in a program

Still another tool makes symbolic bounds analyses for array and pointer accesses.

The general problem is uncomputable. Some partial success is possible for simple cases.

Dynamic Methods

Work by checking every array and pointer access at run time

Since C has such an unwieldy array model, this has a huge performance impact

Also, the program cannot meaningfully continue to run after a buffer pverflow has been detected ⇒ can be used only in development, not in production

Can only be used to find overflows as they occur (actual faults) as opposed to overflows that could occur (potential faults).

How to Write BO-Free Code

In practice, its hard to avoid them if they are at all possible

Its not enough simply to avoid certain functions (although that helps).

Its also not enough to make your source code open for peer review because most people simply dont do peer review:

- A buffer overflow was present for almost a decade in wu-ftpd, an FTP server program before it was finally noticed and removed.

. • Buffer overflows continue to be found in sendmail. Therefore:

Some Hard-And-Fast Rules

- Dont use C
- If you use C++, use smart buffers that do their own range checking
- Always include range checking code
- Never disable range checking code "for performance reasons"
- Use languages with managed memory like Java, Perl, Python
- Design your program so that it is secure from the start

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby