# Function Pointers, Vulnerabilities

## CS 2130: Computer Systems and Organization 1

**Xinyao Yi** Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

## Announcements

- Homework 10 due next Monday
- Lab 12 tomorrow
- Final exam: 7-10pm April 30, Gilmer 301 (different room!)
  - Cumulative, see practice tests
  - Exam conflict form in email

# Using write pig latin example

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    char buf[10];
    ssize_t count = write(1, buf, 10);
    printf("\nWrote %ld bytes\n", count);
}
```

It writes out 10 bytes from the buffer. Then it tells me how many bytes it wrote.

man 2 write

**What is a file descriptor?**

A **file descriptor** is simply:

**An integer index used by the operating system to refer to an open file-like object.**

That's it. It's just a number.

0 standard input
1 standard output
2 standard error
3 the next opened file
4 another file

**1. man 2 write**

- write() takes:
  - a file descriptor (an integer)
  - a pointer to a buffer
  - a number of bytes to write
- It returns an ssize_t:
  - a **non-negative number** = how many bytes were actually written
  - **−1** if there was an error(Check the return section in the manual)

Important:
**write() does NOT know anything about strings.**
It does not stop at '\0', it does not check characters.
It simply copies raw bytes from memory to the file descriptor.

**2. "What is inside buf[10] right now?"**

- The buffer is **uninitialized**.
- The program never assigned anything to buf.
- Therefore, **whatever bytes were already in memory** get printed.

"It could literally be *anything*.
It could be your password, my password, garbage, random binary…
Whatever happened to be in that region of memory before."

**3. What output do we actually see?**

- Your terminal is trying to interpret those bytes as text.
- But they might not be printable characters.
- When it encounters a **zero byte ('\0')**, nothing appears (because terminals don't show null bytes).
- That's why you might see only 3–4 characters even though it "wrote 10 bytes."

**4. Why is this happening?**

"write() is copying bytes directly from memory to stdout.
Nothing checks what they are.
Nothing makes them printable.
The terminal does its best to show them as characters."

**5. other file descriptors**

- write(1, buf, 10) → writes to **stdout**
- write(2, buf, 10) → writes to **stderr**
  - Shows that stdout and stderr appear the same in this terminal
- write(0, buf, 10) → writes to **stdin**
  - Says it's generally a bad idea but surprisingly works on our system
- write(3, buf, 10)
  - Fails with return value **−1**
  - Because fd 3 does not refer to any open file

**6. Why does write(0, buf, 10) sometimes work?**

write(0, buf, 10) means "write 10 bytes to file descriptor 0."
File descriptor 0 is normally **stdin**, which is usually **read-only**.
So in general, writing to stdin is a **bad idea**—it often fails, and on many systems it will just return −1.

However, on the professor's machine, both stdin (fd 0) and stdout (fd 1) were attached to the **same TTY device** (/dev/tty).
A TTY is a **bidirectional** device: you can both read from it and write to it.
So writing to stdin simply wrote those bytes **to the terminal**, and the terminal displayed them, making it *look like it worked*.

## 7. What if (0, NULL, 10)?

**what write() expects**

“write() expects a pointer to a buffer.
It will copy bytes from that buffer into the file descriptor.”

If the buffer pointer is:

- somewhere in memory → it copies bytes
- uninitialized → copies garbage
- **NULL** → points to address 0

“NULL means *invalid memory*.
Your program does not own that memory.”

**What happens when the kernel tries to dereference NULL?**

write() runs **inside the kernel**, so:

- It receives your arguments:
  - fd = 1
  - buf = NULL
  - count = 10
- The kernel tries to read 10 bytes starting at **address 0**.
- “Address 0 is not mapped into your program’s memory.
  So the kernel cannot read from it.
  Therefore the system call fails immediately.”

**Why no crash?**

This is important:

“The program does **not** segfault.
The kernel detects the NULL pointer and refuses to do the write.
It returns −1 instead of crashing your user program.”

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    char buf[10];
    ssize_t count = write(1, buf, 10);
    printf("\nWrote %ld bytes\n", count);
}
```

It writes out 10 bytes from the buffer. Then it tells me how many bytes it wrote.

man 2 write

**What is a file descriptor?**

A **file descriptor** is simply:

**An integer index used by the operating system to refer to an open file-like object.**

That's it. It's just a number.

0 standard input
1 standard output
2 standard error
3 the next opened file
4 another file

**1. man 2 write**

- write() takes:
  - a file descriptor (an integer)
  - a pointer to a buffer
  - a number of bytes to write
- It returns an ssize_t:
  - a **non-negative number** = how many bytes were actually written
  - **−1** if there was an error(Check the return section in the manual)

Important:
**write() does NOT know anything about strings.**
It does not stop at '\0', it does not check characters.
It simply copies raw bytes from memory to the file descriptor.

**2. "What is inside buf[10] right now?"**

- The buffer is **uninitialized**.
- The program never assigned anything to buf.
- Therefore, **whatever bytes were already in memory** get printed.

"It could literally be *anything*.
It could be your password, my password, garbage, random binary…
Whatever happened to be in that region of memory before."

**3. What output do we actually see?**

- Your terminal is trying to interpret those bytes as text.
- But they might not be printable characters.
- When it encounters a **zero byte ('\0')**, nothing appears (because terminals don't show null bytes).
- That's why you might see only 3–4 characters even though it "wrote 10 bytes."

**4. Why is this happening?**

"write() is copying bytes directly from memory to stdout.
Nothing checks what they are.
Nothing makes them printable.
The terminal does its best to show them as characters."

**5. other file descriptors**

- write(1, buf, 10) → writes to **stdout**
- write(2, buf, 10) → writes to **stderr**
  - Shows that stdout and stderr appear the same in this terminal
- write(0, buf, 10) → writes to **stdin**
  - Says it's generally a bad idea but surprisingly works on our system
- write(3, buf, 10)
  - Fails with return value **−1**
  - Because fd 3 does not refer to any open file

**6. Why does write(0, buf, 10) sometimes work?**

write(0, buf, 10) means "write 10 bytes to file descriptor 0."
File descriptor 0 is normally **stdin**, which is usually **read-only**.
So in general, writing to stdin is a **bad idea**—it often fails, and on many systems it will just return −1.

However, on the professor's machine, both stdin (fd 0) and stdout (fd 1) were attached to the **same TTY device** (/dev/tty).
A TTY is a **bidirectional** device: you can both read from it and write to it.
So writing to stdin simply wrote those bytes **to the terminal**, and the terminal displayed them, making it *look like it worked*.

## 7. What if (0, NULL, 10)?

**what write() expects**

“write() expects a pointer to a buffer.
It will copy bytes from that buffer into the file descriptor.”

If the buffer pointer is:

- somewhere in memory → it copies bytes
- uninitialized → copies garbage
- **NULL** → points to address 0

“NULL means *invalid memory*.
Your program does not own that memory.”

**What happens when the kernel tries to dereference NULL?**

write() runs **inside the kernel**, so:

- It receives your arguments:
  - fd = 1
  - buf = NULL
  - count = 10
- The kernel tries to read 10 bytes starting at **address 0**.
- “Address 0 is not mapped into your program’s memory.
  So the kernel cannot read from it.
  Therefore the system call fails immediately.”

**Why no crash?**

This is important:

“The program does **not** segfault.
The kernel detects the NULL pointer and refuses to do the write.
It returns −1 instead of crashing your user program.”

For read() : man 2 read

**1. "read() does NOT know what a string is"**

"read() is the mirror image of write().
Just like write() writes raw bytes,
read() *reads raw bytes* from a file descriptor into a buffer.
It has **no idea** what a C string is."

Meaning:

- No null termination
- No stopping at whitespace
- No concept of characters
- No conversion of anything
- Just bytes, exactly as stored in the file or device

This is **system-level, unbuffered, unformatted I/O**.

**2. The exact function signature**

ssize_t read(int fd, void *buf, size_t count);

void *buf

"read doesn't care what's inside the buffer. It's just a pointer to bytes."

count

"How many bytes do you want me to try to read? That's it."

**3. What does read() return?**

"read() returns an ssize_t — a signed size.
If it's negative, something went wrong.
If it's zero, you hit **end-of-file**.
Otherwise, you get the number of bytes actually read."

"Just like write(),
**the OS is allowed to return fewer bytes than you asked for**."

So:

read(fd, buf, 100);  // might return 100, or 32, or 12, or 0

**4. Real examples**

Example A — When you hit the end of the file

"If you're reading from a file and the file is empty, or you reach the end, read() returns **0**.
That's how you know it's EOF."

Example B — When stdin is being used

Homework 9:

"If you type something and then press Ctrl-D, that sends an EOF to your program.
So read() will return 0. That means you're done."

**5. read() does not wait for all requested bytes**

“If you ask for 20 bytes, the OS only guarantees that you might get **at least 1 byte**,
unless there’s an error or EOF.”

This matters for:

- pipes
- sockets
- terminals
- network I/O
- partial reads

“If you *must* get all 20 bytes, you have to loop until you’ve read them.”

**6. read() does NOT add '\0'**

“Standard I/O functions like scanf and fgets add the null byte. read() does **not**.
So if you want a C string, *you* must add the '\0' yourself.”

```
int n = read(0, buf, 100);
buf[n] = '\0';        // YOU must do this manually
printf("%s\n", buf);
```

**7. "read() has nothing to do with characters"**

“Read does not know or care about characters. It is reading *bytes*. Characters are an interpretation layer added later by the terminal or your code.”

So if you read emoji, multi-byte UTF-8, escape sequences, etc.:

- one "character" may be 2–4 bytes
- read may split a multi-byte character
- the terminal may render weird symbols

**8. compare read() with “buffered I/O” like printf, scanf, fgets**

“read() and write() bypass the stdio buffering.
They talk **directly** to the kernel.
That’s why you must not mix them with buffered I/O unless you know what you’re doing.”

This is why:

```
printf("Hello");
write(1, buf, 10);
printf("World\n");
```

may print things in the wrong order.

**9. How read() can overflow your buffer**

“If you pass the wrong size, or if you assume read() stops at a null byte, you will overflow your buffer.”

Example of a dangerous mistake:

```
char buf[10];
read(0, buf, 100);   // DANGEROUS — writes past buf's boundary
```

# Example: Pig Latin Converter.

## Quick Review:

- **`unistd.h` provides low-level UNIX system calls**, including
  - `read()`
  - `write()`
  - file descriptor operations
- **`stdio.h` and `<string.h>`** work at a higher level (buffered I/O, string utilities).
Understanding how these interact with system calls is essential.
- **Programs often combine both low-level and high-level functions**.
For example:
  - Use `read()` to get raw bytes
  - Use string functions (`strpbrk`, `isalpha`) to process text
  - Use `fwrite()` for consistent output
- **Sockets are also file descriptors**.
They use the same system-call interface (`read`, `write`, `close`), but instead of referring to files or the terminal, they refer to network connections.