Secure Software Design and Engineering
(CY-321)

Handling Memory Securely

Dr. Zubair Ahmad

Handling Sensitive Data

A secret key (or any other sensitive piece of data) must not ever be outside your control in unencrypted form!

Thats so important that we will repeat it:

A secret key (or any other sensitive piece of data) must not ever be outside your control in unencrypted form!

Handling Sensitive Data

Even if the data is officially erased (by overwriting it), it can still be restored (the phenomenon is called "**remanence**")

It very difficult to erase data from a disk

How Memory Remanence Works

Volatile memory, such as **Dynamic RAM (DRAM) and Static RAM (SRAM)**, is expected to lose its stored data when power is removed. However, due to the following factors, data can persist for milliseconds to minutes:

- **Capacitive Charge Retention**: In DRAM, memory cells are tiny capacitors that take time to fully discharge.
- **Low-Temperature Effects**: Freezing memory chips (e.g., using liquid nitrogen) can significantly extend data retention.
- **Partial Power Loss**: A sudden power loss does not always instantly erase data, especially in battery-backed or hybrid memory systems.

Cold Boot Attacks

An attacker physically reboots a computer without properly shutting it down and then dumps memory contents before the system clears them

The attacker forcefully reboots the system.

1.A lightweight OS is quickly loaded to extract memory contents.
2.The memory dump is analyzed for cryptographic keys or sensitive data.

Mitigation:

1.Use **Full Disk Encryption (FDE) with pre-boot authentication**
•**Disable booting from external devices**
•**Perform memory overwriting at shutdown** to clear sensitive data.

By cooling RAM modules (e.g., with liquid nitrogen or compressed air), attackers slow down the loss of residual data

Freeze Attack

The attacker cools down RAM to preserve data.

The RAM module is quickly removed and inserted into another system.

Memory is read and analyzed to extract sensitive data.

Dangling Pointers

Continues to reference a memory location **after the memory has been freed or deallocated**.

Accessing a dangling pointer leads to **undefined behavior**, which can cause crashes, data corruption, or security vulnerabilities.

Causes of Dangling Pointers

Use-After-Free

A pointer still references memory that has been freed

```
int *ptr = (int *)malloc(sizeof(int));
free(ptr);
*ptr = 10;
```

Causes of Dangling Pointers

Returning Address of a Local Variable

A function returns the address of a local variable, but the variable goes out of scope when the function exits.

```
int* getPointer() {
    int x = 10;
    return &x;
}
```

Causes of Dangling Pointers

Pointer Goes Out of Scope

A pointer to dynamically allocated memory is still in scope, but the memory is freed elsewhere.

```
int *ptr;
{
    int x = 5;
    ptr = &x;  // x goes out of scope
at the end of this block
}
*ptr = 10;  // Dangling pointer
access
```

Using mlock()

`mlock()` is a system call used to **lock a region of memory** into RAM, preventing it from being swapped out to disk. This is particularly important for securing **sensitive data** such as passwords, encryption keys, and authentication tokens.

```c
#include <sys/mman.h>
int mlock(const void *addr, size_t len);
```

Using mlock()

```
#include <sys/mman.h>
#include <stdlib.h>

/* Warning! mlock calls don't stack! */
key_t *lookup_secret_key(const char *user) {
    key_t *ret = (key_t *)malloc(sizeof(key_t));
    if (ret != NULL) {
        /* Must be root for this to succeed */
        if (mlock(ret, sizeof(key_t)) == 0) {
            /* Proceed */
        } else {
            /* Handle error */
        }
    }
    return ret;
}

int release(const void *buf, size_t len) {
    free((void *)buf);
    /* Must be root for this to succeed */
    return munlock(buf, len);
}
```

Using mlock()

```
unsigned char* encrypt_file(const char* file_name, const char* user) {
    key_t* secret_key = lookup_secret_key(user); /* Uses mlock(2)! */
    unsigned char* plaintext = read_file(file_name); /* Uses mlock(2)! */
    unsigned char* ciphertext = encrypt(plaintext, secret_key);

    release(secret_key);
    release(plaintext);

    return ciphertext;
}

int release(const void* buf, size_t len) {
    free(buf);
    update_page_counts(buf, len);

    if (page_counts_are_zero(buf, len)) {
        return munlock(buf, len);
    } else {
        return 0;
    }
}
```

Zeroing Memory

1. Allocate 1000 bytes
2. Fill 1000 bytes with secret key material
3. Use key material, then release buffer
4. Allocate 1000 bytes for new buffer
5. Fill only 500 bytes with harmless message
6. Write 1000 bytes to file
7. Release buffer

Result: 500 bytes of secret key material leaked
Happens quickly: Difference between length and size of a buffer often not well understood.

Why This Happens Quickly?

Developers often confuse the **allocated size** with the **valid length of the data** stored in a buffer.

If the program assumes that the full 1000-byte buffer is always safe to write (without considering what was overwritten), it may inadvertently include **stale, sensitive data**.

So You Think It Cant Happen?

Happened to Ethernet driver in Linux.

When a very small packet was received, the return packet was incompletely initialized.

Result: interesting information from the kernels memory was leaked. Could have been everything from Moms shopping list to passwords.

How to Avoid it?

```
#include <stdlib.h>

int release (void* buf, len_t len) {
memset(buf, ’\0’, len); /* <- Zeroize buffer before
freeing */
free(buf);
Update_page_counts(buf, len);
if (page_counts_are_zero(buf, len))
return munlock(buf, len);
else 10
return 0;
}
```

Locality of Reference

Or we call it **principle of locality**

How programs tend to access memory locations in a predictable manner

Subsequent data locations that are referenced when a program is run are often predictable and in proximity to previous locations based on time or space

Locality of Reference

Temporal Locality (Locality in Time)

If a memory location is accessed, it is likely to be accessed again soon.

Example: In a loop, a variable is used repeatedly within a short period

Locality of Reference

Spatial Locality (Locality in Space)

If a memory location is accessed, nearby memory locations are likely to be accessed soon.

Example: Arrays and sequential instructions in a program often lead to access patterns where adjacent memory locations are used.

Locality of Reference

Branch Locality

The tendency of a program to repeatedly execute the same branches (e.g., loops and conditional statements).

When a program makes a decision (e.g., an if-else statement or a loop), it is likely to follow the same path multiple times before switching.

Locality of Reference

Equidistant Locality

Memory access patterns where data is accessed at fixed, regular intervals.

Unlike spatial locality (where consecutive memory locations are accessed), equidistant locality occurs when memory locations are accessed with a constant step size.

Garbage Collection

An **automatic memory management** technique used by programming languages to reclaim memory that is no longer needed, preventing memory leaks and dangling pointers.

Garbage Collection

Reachability Analysis (Root Set)

The GC **tracks "live" objects** by starting from **root references** (global variables, stack variables, registers).

Any object **not reachable** from these roots is considered **garbage** and is eligible for collection.

Garbage Collection

Reference Counting

Each object has a **reference counter**.

When the counter reaches **zero**, the object is deleted.

Mark-and-Sweep

Mark Phase: Identify reachable objects.

Sweep Phase: Free memory occupied by unreachable objects.

Generational Garbage Collection
(Used in Java,)

Objects are classified into **Young, Old, and Permanent Generations**:

- **Young Generation**: Short-lived objects (e.g., temporary variables).
- **Old Generation**: Long-lived objects (e.g., static data).
- **Permanent Generation**: Metadata

Copying Garbage Collection

- Divides memory into two halves: **Active (From-Space) & Inactive (To-Space)**.
- **Live objects are copied** to the inactive space, and the old space is freed.
- **Efficient but wastes memory** (since only half is used at a time)

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby