Vulnerabilities and Defensive Programming

INF-744: Security and Privacy for IoT

Diego F. Aranha

Institute of Computing – University of Campinas

Introduction

We classify software vulnerabilities in:

1. **Input representation:** buffer overflows
2. **Interface abuse** dangerous functions
3. **Time and state:** synchronization errors and race conditions
4. **Error handling:** function return values and exception handling
5. **Memory management:** memory allocation and liberation
6. **Security mechanisms:** design and implementation of cryptography

Tools: For static or dynamic code analysis, `cppcheck`, `flawfinder`, *Fortify*, *Coverity*, *Valgrind*.

Vulnerabilities

Vulnerabilities can have different impacts in security properties:

- **Integrity:** corruption of memory and execution state
- **Availability:** denial of service by exaggerate resource consumption
- **Confidentiality:** exposure of secret data by interface
- **Access control:** privilege escalation
- **Predictability:** software behavior can become unpredictable

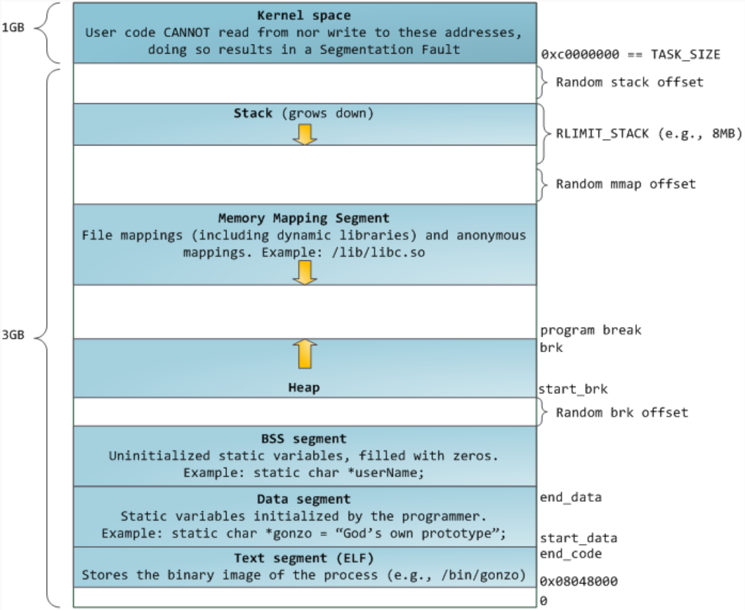
Buffer overflow

Main vulnerability in the C programming language, occurs when the adversary writes more data than the buffer is able to store, overwriting local execution contest and aiming to continue execution in an attacker-provided code segment.

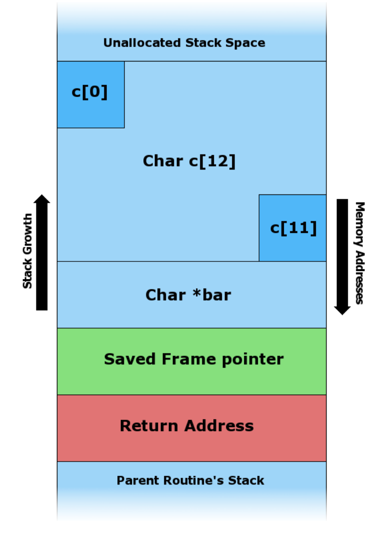
Example:

```c
#include <string.h>

void foo (char *bar) {
   char  c[12];
   strcpy(c, bar); // Vulnerable call...
}

int main (int argc, char **argv) {
   foo(argv[1]);
}
```

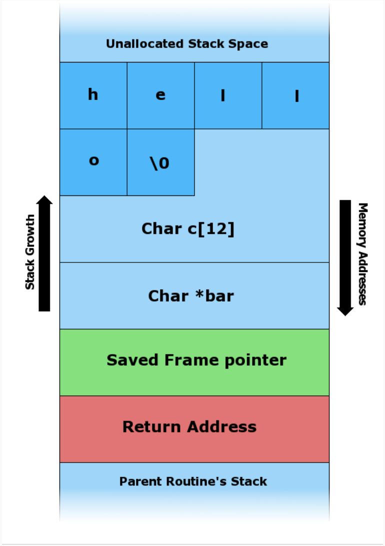
Buffer overflow



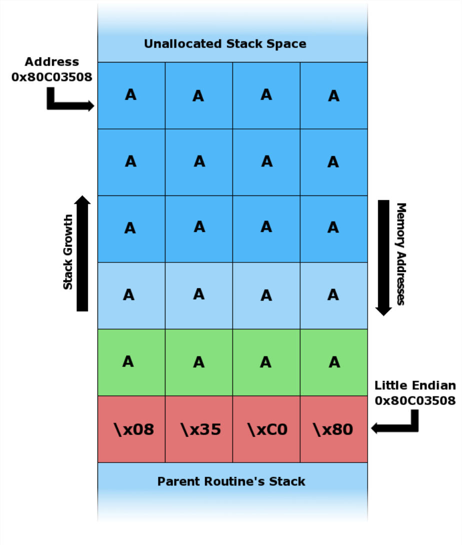
Buffer overflow



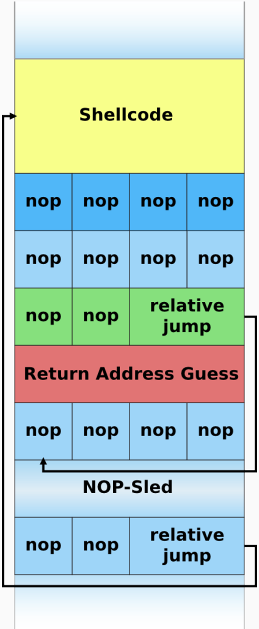
Buffer overflow



Buffer overflow



Buffer overflow



Stack overflow

Buffer overflow on data allocated in the stack.

Buffer overflow

Stack overflow

Buffer overflow on data allocated in the stack.

Defenses:

- Manual/automatic static/dynamic verification
- Canaries for detection
- Address randomization
- Safe functions
- Non-executable stack

Buffer overflow

Stack overflow

Buffer overflow on data allocated in the stack.

Defenses:

- Manual/automatic static/dynamic verification
- Canaries for detection
- Address randomization
- Safe functions
- Non-executable stack

Heap overflow

Analogous to a stack overflow, but in the heap. The attack usually overwrites pointers in the double linked list implementing the heap in a way that when memory is freed, the return address will be overwritten.

Defenses: Same as stack overflow.

Problem: Stack and heap marked as non-executable.

Buffer overflow

Problem: Stack and heap marked as non-executable.

Return to `libc`

Buffer overflow that overwrites the return address with the address of a function of interest in the program or supporting libraries. Context is usually built on the stack for the function execution.

Defense: Function arguments passed by registers.

Buffer overflow

Problem: Stack and heap marked as non-executable.

Return to `libc`

Buffer overflow that overwrites the return address with the address of a function of interest in the program or supporting libraries. Context is usually built on the stack for the function execution.

Defense: Function arguments passed by registers.

Return-oriented programming

When the program does not provide useful functions, one can construct arbitrary code from the chaining of useful execution sequences.

Defense: Address randomization, canaries and dynamic monitoring.

Buffer overflows

Integer overflows

Consists in the manipulation of integers to reduce buffer capacity, facilitating a buffer overflow exploit.

Defense: Manual or automatic verification of valid values, native multi-precision support in the programming language.

Example:

```c
char *buf;
int i, len;

read(fd, &len, sizeof(len));

/* OOPS! We forgot about the negatives! */
if (len > 8000) { error("too large length"); return; }

buf = malloc(len);
read(fd, buf, len); /* Overflow after sign conversion */
```

Important: The attack is also possible without involving type conversion.

Example:

```c
char *buf;
size_t len;

read(fd, &len, sizeof(len));

/* Forgot to verify maximum size */

buf = malloc(len+1); /* +1 can overflow to malloc(0) */
read(fd, buf, len);
buf[len] = '\0';
```

Other input vulnerabilities

Command injection

Programs running under privilege and with shell execution functionality can be forced to execute arbitrary commands. Non-trusted libraries can be manipulated to execute arbitrary code during load.

Defense: Reduce privileges and unnecessary support for commands. Consider static linking.

Resource abuse

Allows the adversary to manipulate resource identifier to force a program to corrupt or read protected files.

Defense: Minimize privileges.

Other input vulnerabilities

Illegal pointers

Manipulating pointers returned by functions to point to other buffers of interest can substantially change the behavior of the program.

Defense: Verify pointers returned by functions and reject invalid values.

Record forgery

Manipulate specific events after a buffer overflow to hide attacker actions.

Defense: Use cryptography to protect integrity of log records.

Dangerous functions

Known dangerous functions should never be used (and can easily be detected by static code analysis). Classical examples are string functions.

Defense: Static code analysis.

Interface abuse

Dangerous functions

Known dangerous functions should never be used (and can easily be detected by static code analysis). Classical examples are string functions.

Defense: Static code analysis.

Heap inspection

Reallocating buffer that store sensitive information may prevent its destruction.

Defense: Do not dynamically reallocate buffers storing sensitive information.

Interface abuse

Dangerous functions

Known dangerous functions should never be used (and can easily be detected by static code analysis). Classical examples are string functions.

Defense: Static code analysis.

Heap inspection

Reallocating buffer that store sensitive information may prevent its destruction.

Defense: Do not dynamically reallocate buffers storing sensitive information.

Inconsistent implementations

Using functions with inconsistent implementations in different platforms can make the program behavior unpredictable.

Defense: Restrict function calls to standard and portable ones.

Synchronization errors

Acquisition and liberation of locks can affect unavailability.

Defense: Require locks to be acquired in a well-defined order and test corner cases.

Time and state

Synchronization errors

Acquisition and liberation of locks can affect unavailability.

Defense: Require locks to be acquired in a well-defined order and test corner cases.

Race conditions

The time between the verification of a file property and using the file can allow privilege scalation, specially if files are temporary.

Defense: Atomic file manipulation calls.

Example:

```
char *filename; int fd;
do {
    filename = tempnam (NULL, "foo");
    fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
    free (filename);
} while (fd == -1);
```

Insufficient handling of return values

Not handling return values can change program behavior in case of error.

Defense: Verify all return values and handle errors correctly.

Exception triggering

Non-captured exceptions can affect availability of software.

Defense: Handle rigorously all exceptions threw by the program.

Memory management

Double free

Freeing the same memory position consecutively can cause a buffer overflow through manipulation of the pointer metadata in the doubly linked list used to implement the heap.

Memory management

Double free

Freeing the same memory position consecutively can cause a buffer overflow through manipulation of the pointer metadata in the doubly linked list used to implement the heap.

Memory leak

Allocating memory and not freeing it may allow the attacker to exhaust resources used by the program, impacting availability. Always zero sensitive data before freeing memory blocks.

Memory management

Double free

Freeing the same memory position consecutively can cause a buffer overflow through manipulation of the pointer metadata in the doubly linked list used to implement the heap.

Memory leak

Allocating memory and not freeing it may allow the attacker to exhaust resources used by the program, impacting availability. Always zero sensitive data before freeing memory blocks.

Use after free

Referring to memory already free can crash the program (unavailability), corrupt the execution state and even allow execution of arbitrary code.

Defense: Careful memory management with a dynamic analysis tool, assign `NULL` to invalid pointers.

Case study: Heartbleed