

---

# Dangerous APIs

Ibéria Medeiros

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

# Introduction

---

- Some of the commonly used functions can be utilized in ways that facilitate the introduction of errors
- Therefore, whenever possible, one should discourage their use although
  - ☞ *“there are no such things as dangerous functions, only dangerous developers”,*  
Dave Cutler, chief architect of Windows NT
- As we will see through the course, most of the security vulnerabilities are created due to unjustified trust that is put in some program input
  - ☞ developers can program with dangerous functions *as long as* input data is well formatted and comes from a trustworthy source

# Introduction

---

- Two ideas to keep in mind

☞ Typically, a software component **cannot** be made secure simply by substituting the dangerous functions by their more secure versions

☞ In order to build more secure software, one needs to **follow the data** as it is processed through the code, and determine the correction and level of trust that can be put whenever the data is used in the program

# (Some) Problems

---

- Buffer Overflow
- Off-by-one
- Name-Squatting
- Denial of service

# Example Vulnerability

---

- US-CERT VU#513062

👉 **Title:** “metamail contains **multiple buffer overflow vulnerabilities**”

👉 **Summary:** “Multiple buffer overflows in the metamail package could allow a remote attacker to **execute arbitrary code** on a vulnerable system. An attacker may be able to exploit these vulnerabilities via a **specially-crafted email message**.”

👉 **Description:** “Two buffer overflows due to **incorrect use of strcpy()** have been discovered in various portions of the metamail codebase”

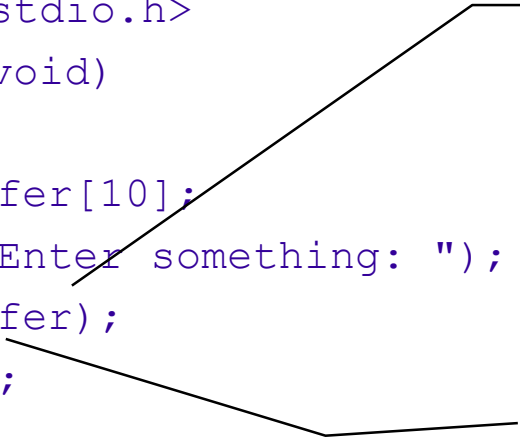
# gets()

---

```
char *gets(char *buffer);
```

reads a line from stdin into the buffer pointed to by `buffer` until either a terminating newline or EOF, which it replaces with `'\0'`

```
#include <stdio.h>
int main (void)
{
    char buffer[10];
    printf("Enter something: ");
    gets(buffer);
    return 0;
}
```



There is no control over how much information is read from stdin!!!

Notice that sometimes you might read more than 10 bytes and the program continues to work correctly!

# fgets()

---

`char *fgets(char *buffer, int size, FILE *stream);`

reads in at most **one less than** `size` characters from `stream` and stores them into the `buffer`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A `'\0'` is stored after the last character in the buffer.

```
int  nbytes = 8; /* max number of bytes
int main (void)
{
    char buffer[nbytes], buffer1[nbytes];
    sprintf(buffer1, "Ola");
    printf("Enter something [max %d]: ", nbytes);
    fgets(buffer, nbytes+1, stdin);
    printf("\nbuf = %s\n", buffer);
    printf("buf1  = %s\n", buffer1);
    return 0;
}
```

Off-by-one vulnerability:  
Here the problem is NOT due to `fgets()` but due to a bad calculation for the buffer size.

**NOTE:** In practice, this problem **may or may not** have an impact because of the way variables are aligned in memory!

# strcpy()

`char *strcpy(char *dest, const char *src);`

copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

```
#define STR1 "12345678"
int main (void)
{
    static char buffer[8], buffer1[8];
    char *str = "ola!!!";
    strcpy(buffer1, STR1);
    strcpy(buffer, str);
    printf("\nbuf = %s\n", buffer);
    printf("buf1 = %s\n", buffer1);
    return 0;
}
```

Off-by-one vulnerability:  
The problem is that STR has 9 bytes due to the `'\0'` at the end.

This bug will probably have an impact because variables might not be aligned in the “uninitialized data” segment, and therefore, there are no “spaces” between them!



# strncpy()

```
char *strncpy(char *dest, const char *src, size_t len);
```

the function is similar, except that at most `len` bytes of `src` are copied. If there is no null byte among the first `len` bytes of `src`, the string placed in `dest` will not be null-terminated.

```
#define STR "12345678"
char buffer[8], buffer1[8];
int main (void)
{
    strncpy(buffer, STR, strlen(STR));
    strncpy(buffer1, STR, sizeof(buffer1)-1);
    printf("\nbuf = %s, buf1= %s\n", buffer, buffer1);
    return 0;
}
```

Off-by-one vulnerability:  
The problem is that `strlen()` does not count the `'\0'` at the end.

No problem here!

- 1) The “uninitialized data” segment variables are initialized with `'0'`, and therefore a `'\0'` is placed at the end of the buffer;
- 2) `sizeof()` gives the number of bytes of buffer, and we save space for the `'\0'`.

# strcat() and strncat()

---

`char *strcat(char *dest, const char *src);`

appends the `src` string to the `dest` string, overwriting the null byte (`'\0'`) at the end of `dest`, and then adds a terminating null byte. The strings may not overlap, and the `dest` string must have enough space for the result.

`char *strncat(char *dest, const char *src, size_t len);`

is similar, except that 1) it will use at most `len` characters from `src`; and 2) `src` does not need to be null-terminated if it contains `len` or more characters. As with `strcat()`, the resulting string in `dest` is always null-terminated.

If `src` contains `len` or more characters, `strncat()` writes `n+1` characters to `dest` (`len` from `src` plus the terminating null byte). Therefore, the size of `dest` must be at least `strlen(dest) + len + 1`.

# sprintf()

```
int sprintf(char *str, const char *format, ...);
```

write to the buffer `str` under the control of a `format` string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of `stdarg(3)`) are converted for output

```
#define STR "12345678"
char buffer[8], buffer1[8];
int main (void)
{
    sprintf(buffer, "%5s", STR);
    sprintf(buffer1, "%.5s", STR);
    printf("\nbuf = %s, buf1= %s\n", buffer, buffer1);
    return 0;
}
```

Buffer overflow:  
“5” indicates the minimum number of characters that are written.

No problem here!  
“.5” defines the maximum number of bytes that are written to the string.

# snprintf() and others

---

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

similar, but write at most `size` bytes (including the trailing null byte ('\\0')) to `str`.

Much more safer! But incorrect formats, for example, can lead to a `str` that is incorrectly modified (e.g., truncated).

Have a look at these others:

`int fprintf(FILE *stream, const char *format, ...);`

`int vfprintf(FILE *stream, const char *format, va_list ap);`

`int vsprintf(char *str, const char *format, va_list ap);`

`int vsnprintf(char *str, size_t size, const char *format, va_list ap);`

# scanf() and others

---

```
int scanf(const char *format, ...);
```

```
int fscanf(FILE *stream, const char *format, ...);
```

```
int sscanf(const char *str, const char *format, ...);
```

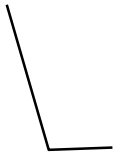
```
int vscanf(const char *format, va_list ap);
```

```
int vsscanf(const char *str, const char *format, va_list ap);
```

```
int vfscanf(FILE *stream, const char *format, va_list ap);
```

scans input according to `format`. This `format` may contain conversion specifications; the results from such conversions, if any, are stored in the locations pointed to by the pointer arguments that follow `format`.

Each pointer argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.



Any error on the definition of the format can lead to potential overflows (think about inputting a buffer of char). There are no “secure” versions of these functions!

# Do we have a vulnerability here?

---

```
replydirname(name, message)
const char *name, *message;
{
    char npath[MAXPATHLEN];
    int i;
    for (i = 0; *name != '\\0' && i < sizeof(npath) - 1;
        i++, name++) {
        npath[i] = *name;
        if (*name == '\\')
            npath[++i] = '\\';
    }
    npath[i] = '\\0';
    reply(257, "\\\"%s\\\" %s", npath, message);
}
```

# Bibliography

---

- Man pages (-;