CERN Computer Security

Computer security emergency contact
Computer.Security@cern.ch 70500
Contact en cas d'incident de sécurité informatique

Common vulnerabilities guide for C programmers

Intro

Most vulnerabilities in C are related to buffer overflows and string manipulation. In most cases, this would result in a segmentation fault, but specially crafted malicious input values, adapted to the architecture and environment could yield to arbitrary code execution. You will find below a list of the most common errors and suggested fixes/solutions. (*Some tips for C++ are available here.*)

gets

The stdio `gets()` function does not check for buffer length and always results in a vulnerability.

Vulnerable code

```
#include <stdio.h>
int main () {
    char username[8];
    int allow = 0;
    printf ("Enter your username, please: ");
    gets(username); // user inputs "malicious"
    if (grantAccess(username)) {
        allow = 1;
    }
    if (allow != 0) { // has been overwritten by the overflow of the username.
        privilegedAction();
    }
    return 0;
}
```

Mitigation

Prefer using fgets (and dynamically allocated memory!):

```
#include <stdio.h>
#include <stdlib.h>
#define LENGTH 8
int main () {
    char* username, *nlptr;
    int allow = 0;

    username = malloc(LENGTH * sizeof(*username));
    if (!username)
        return EXIT_FAILURE;
    printf ("Enter your username, please: ");
    fgets(username,LENGTH, stdin);
    // fgets stops after LENGTH-1 characters or at a newline character, which ever co
    // but it considers \n a valid character, so you might want to remove it:
    nlptr = strchr(username, '\n');
```

```
    if (nlptr) *nlptr = '\0';

    if (grantAccess(username)) {
        allow = 1;
    }
    if (allow != 0) {
        priviledgedAction();
    }

    free(username);

    return 0;
}
```

strcpy

The `strcpy` built-in function does not check buffer lengths and may very well overwrite memory zone contiguous to the intended destination. In fact, the whole family of functions is similarly vulnerable: `strcpy`, `strcat` and `strcmp`.

Vulnerable code

```
char str1[10];
char str2[]="abcdefghijklmn";
strcpy(str1,str2);
```

Mitigation

The best way to mitigate this issue is to use `strlcpy` if it is readily available (which is only the case on BSD systems). However, it is very simple to define it yourself, as shown below:

```
#include <stdio.h>

#ifndef strlcpy
#define strlcpy(dst,src,sz) snprintf((dst), (sz), "%s", (src))
#endif

enum { BUFFER_SIZE = 10 };

int main() {
    char dst[BUFFER_SIZE];
    char src[] = "abcdefghijk";

    int buffer_length = strlcpy(dst, src, BUFFER_SIZE);

    if (buffer_length >= BUFFER_SIZE) {
        printf ("String too long: %d (%d expected)\n",
                buffer_length, BUFFER_SIZE-1);
    }

    printf ("String copied: %s\n", dst);

    return 0;
}
```

Another and may be slightly less convenient way is to use `strncpy`, which prevents buffer overflows, but does not guarantee '\0'-termination.

```
enum { BUFFER_SIZE = 10 };
char str1[BUFFER_SIZE];
char str2[]="abcdefghijklmn";

strncpy(str1,str2, BUFFER_SIZE); /* limit number of characters to be copied */
// We need to set the limit to BUFFER_SIZE, so that all characters in the buffer
// are set to '\0'. If the source buffer is longer than BUFFER_SIZE, all the '\0'
// characters will be overwritten and the copy will be truncated.

if (str1[BUFFER_SIZE-1] != '\0') {
    /* buffer was truncated, handle error? */
}
```

For the other functions in the family, the `*n*` variants exist as well: `strncpm` and `strncat`

sprintf

Just as the previous functions, `sprintf` does not check the buffer boundaries and is vulnerable to overflows.

Vulnerable code

```
#include <stdio.h>
#include <stdlib.h>

enum { BUFFER_SIZE = 10 };

int main() {
    char buffer[BUFFER_SIZE];
    int check = 0;

    sprintf(buffer, "%s", "This string is too long!");

    printf("check: %d", check); /* This will not print 0! */

    return EXIT_SUCCESS;
}
```

Mitigation

Prefer using `snprintf`, which has the double advantage of preventing buffers overflows and returning the minimal size of buffer needed to fit the whole formatted string.

```
#include <stdio.h>
#include <stdlib.h>

enum { BUFFER_SIZE = 10 };

int main() {
    char buffer[BUFFER_SIZE];

    int length = snprintf(buffer, BUFFER_SIZE, "%s%s", "long-name", "suffix");

    if (length >= BUFFER_SIZE) {
        /* handle string truncation! */
    }

    return EXIT_SUCCESS;
}
```

printf and friends

One other vulnerability category is concerned with string formatting attacks, those can cause information leakage, overwriting of memory, … This error can be exploited in any of the following functions: `printf`, `fprintf`, `sprintf` and `snprintf`, *i.e.* all functions that take a “format string” as argument.

Vulnerable code

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

int main(int argc, char **argv) {
    char *secret = "This is a secret!\n";

    printf(argv[1]);

    return 0;
}
```

Now, this code, if compiled with the `-mpreferred-stack-boundary=2` option (*on a 32-bit platform; on 64-bit things work slightly differently, but the code still is vulnerable!*), can yield interesting results.

If called with `./FormatString %s`, it will print the secret string.

```
$ gcc -mpreferred-stack-boundary=2 FormatString.c -o FormatString
$ ./FormatString %s
This is a secret!
$
```

Note: the `-mpreferred-stack-boundary=2` option is in no way necessary to cause information leakage and not setting it does not make your code more secure by any means. It just allows for a simpler and more straight forward example.

More info / detailed explanations on: this forum post

Mitigation

It's really simple: **always** hardcode the format string. At least, **never** let it come directly from any user's input.

File opening

Much care must be taken when opening files, as many issues can arise. This is covered in much detail by Kupsch and Miller in this tutorial. They also provide libraries implementing their approach. Out of the many ways file handling can be attacked, we will only present two brief examples below.

Some of the basic pitfalls are described below.

Symbolic link attack

It is a good idea to check whether a file exists or not before creating it. However, a malicious user might create a file (or worse, a symbolic link to a critical system file) between your check and the moment you actually use the file.

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

#define MY_TMP_FILE "/tmp/file.tmp"


int main(int argc, char* argv[])
{
    FILE * f;
    if (!access(MY_TMP_FILE, F_OK)) {
        printf("File exists!\n");
        return EXIT_FAILURE;
    }
    /* At this point the attacker creates a symlink from /tmp/file.tmp to /etc/passwd
    tmpFile = fopen(MY_TMP_FILE, "w");

    if (tmpFile == NULL) {
        return EXIT_FAILURE;
    }

    fputs("Some text...\n", tmpFile);

    fclose(tmpFile);
    /* You successfully overwrote /etc/passwd (at least if you ran this as root) */

    return EXIT_SUCCESS;
}
```

Mitigation

Avoid the race condition by accessing directly the file, and don't overwrite it if it

already exists. So,

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>

#define MY_TMP_FILE "/tmp/file.tmp"

enum { FILE_MODE = 0600 };

int main(int argc, char* argv[])
{
    int fd;
    FILE* f;

    /* Remove possible symlinks */
    unlink(MY_TMP_FILE);
    /* Open, but fail if someone raced us and restored the symlink (secure version of
    fd = open(MY_TMP_FILE, O_WRONLY|O_CREAT|O_EXCL, FILE_MODE);
    if (fd == -1) {
        perror("Failed to open the file");
        return EXIT_FAILURE;
    }
    /* Get a FILE*, as they are easier and more efficient than plan file descriptors
    f = fdopen(fd, "w");
    if (f == NULL) {
        perror("Failed to associate file descriptor with a stream");
        return EXIT_FAILURE;
    }
    fprintf(f, "Hello, world\n");
    fclose(f);
    /* fd is already closed by fclose()!!! */
    return EXIT_SUCCESS;
}
```


e-mail: **Computer.Security@cern.ch**
Phone: +41 22 767 0500
(Please listen to the recorded instructions.)

Please use the following PGP key to encrypt messages sent to
CERN Computer Security Team <**Computer.Security@cern.ch**>
A9A7C562 / 322E 144D B867 5403 6747 F465 8089 8536 A9A7 C562