Safer functions: Bounds-checking interfaces introduced in C11

By Ayush Dixit, Jun, Xiao Guo, Heng Jin, Daniel Imondo

Group 10

Introduction

Issues with older functions cause for making an informed decisions which functions are best to use.

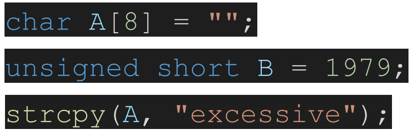
We will explain why these functions are not safe in detail, and examine alternative safer implementations.

Hopefully this presentation will allow you to write more informed code, and avoid having your code exploited with a buffer overflow.

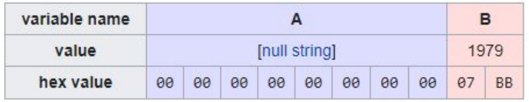
Why old functions are not safe

Buffer Overflow : When the writing behavior overruns the buffer's boundary, some adjacent locations might be overwritten, thus cause unexpected consequences.

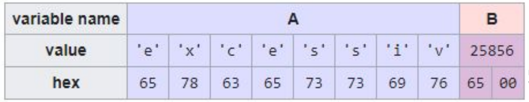
```
char buf[24];

printf("Please enter your name\n");

gets(buf); -> Segmentation Fault
```



```
char A[8] = "";
unsigned short B = 1979;
strcpy(A, "excessive");
```

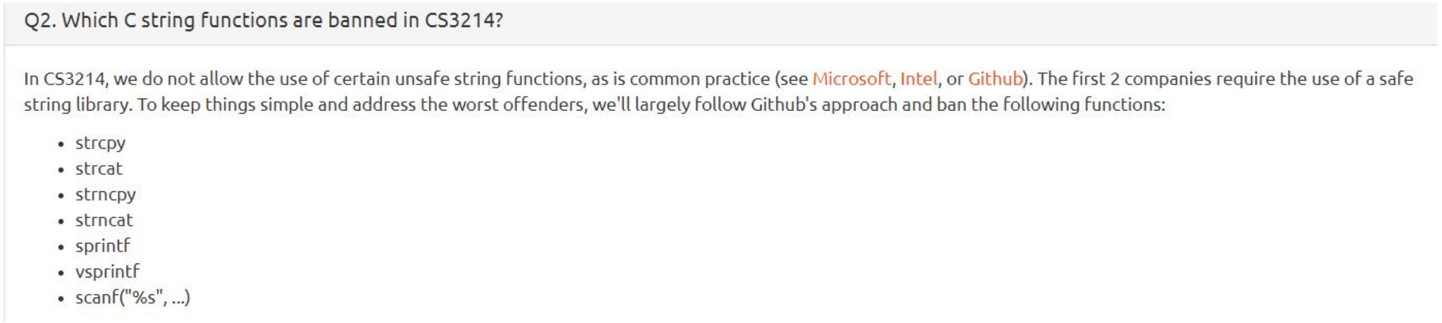


variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB



variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

https://en.wikipedia.org/wiki/Buffer_overflow#:~:text=A%20buffer%20overflow%20occurs%20when,fits%20within%20the%20destination%20buffer.



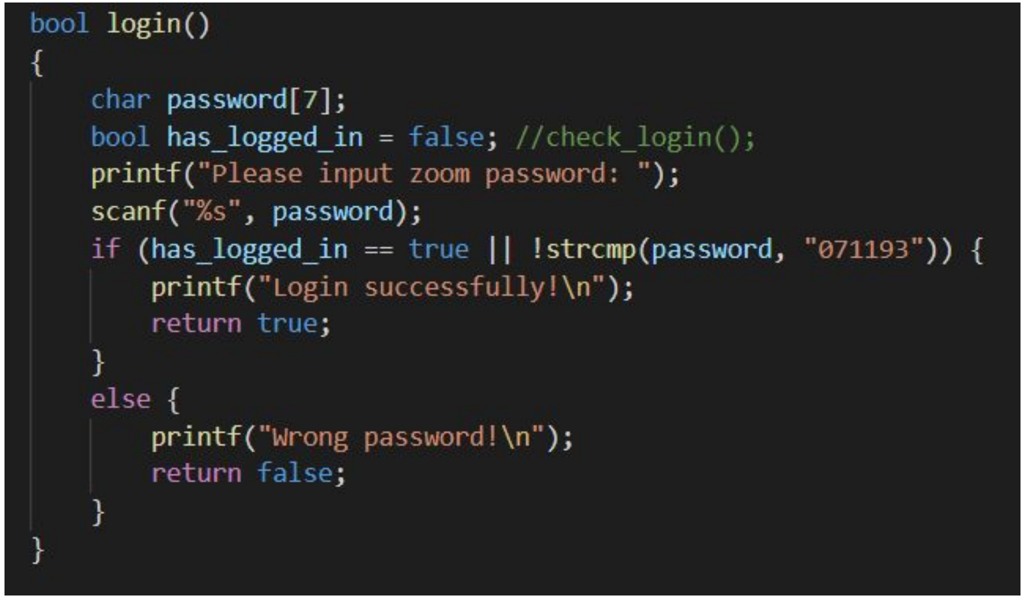
Q2. Which C string functions are banned in CS3214?

In CS3214, we do not allow the use of certain unsafe string functions, as is common practice (see Microsoft, Intel, or Github). The first 2 companies require the use of a safe string library. To keep things simple and address the worst offenders, we'll largely follow Github's approach and ban the following functions:

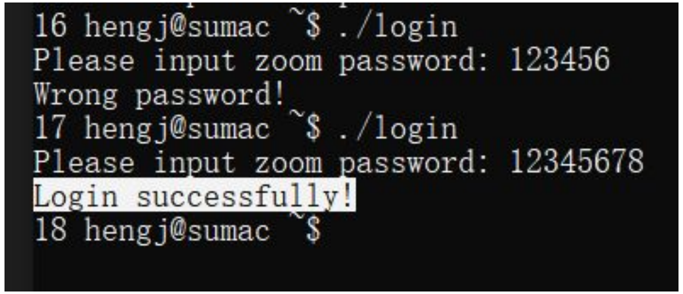
- strcpy
- strcat
- strncpy
- strncat
- sprintf
- vsprintf
- scanf("%s", ...)

Why old functions are not safe

How buffer overflow can bypass the security check



```c
bool login()
{
    char password[7];
    bool has_logged_in = false; //check_login();
    printf("Please input zoom password: ");
    scanf("%s", password);
    if (has_logged_in == true || !strcmp(password, "071193")) {
        printf("Login successfully!\n");
        return true;
    }
    else {
        printf("Wrong password!\n");
        return false;
    }
}
```



```
16 hengj@sumac ~$ ./login
Please input zoom password: 123456
Wrong password!
17 hengj@sumac ~$ ./login
Please input zoom password: 12345678
Login successfully!
18 hengj@sumac ~$
```

Instruction

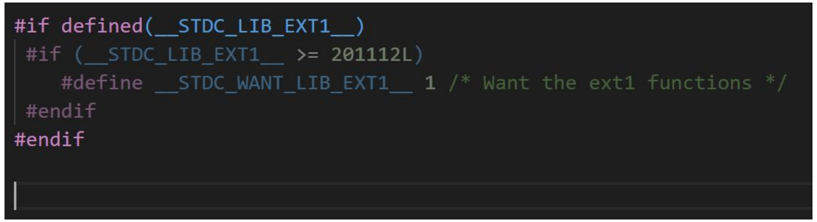
Introduce new type :`errno_t` (C11).

- Is a type defined in `<errno.h>` (C11).
- Is a type `int`.
- Used to safe functions that return errno values

Introduce new marco/extension :

- `__STDC_LIB_EXT1` (C11).
- `__STDC_WANT_LIB_EXT1` (C11).

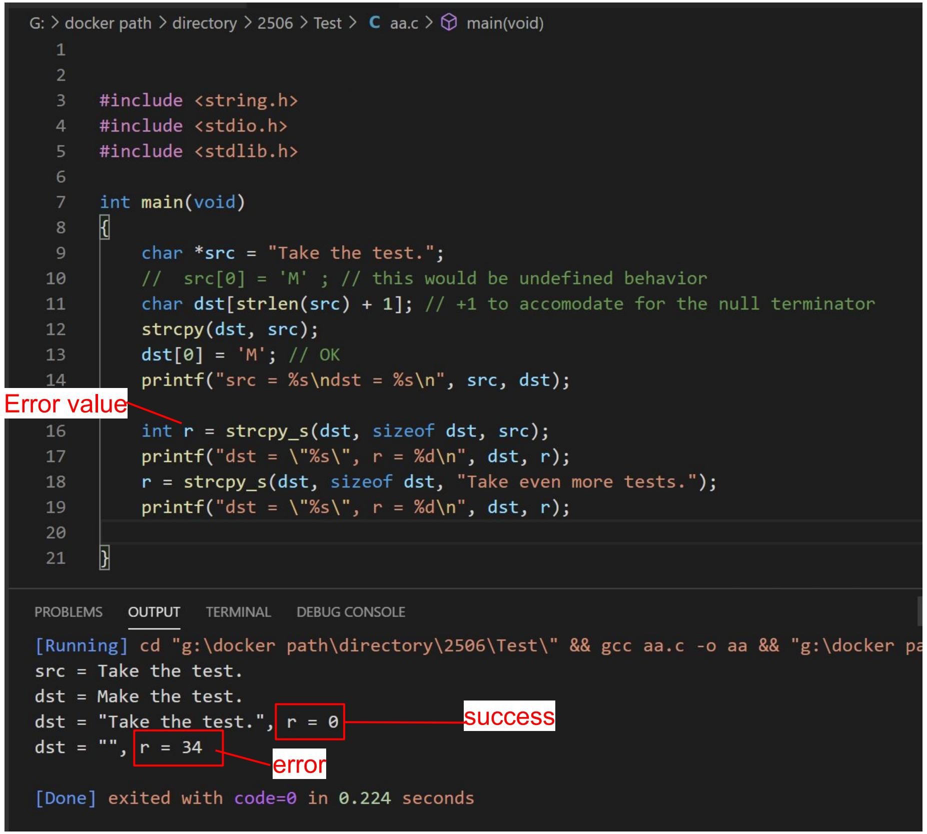
Instruction example



```
#if defined(__STDC_LIB_EXT1__)
 #if (__STDC_LIB_EXT1__ >= 201112L)
    #define __STDC_WANT_LIB_EXT1__ 1 /* Want the ext1 functions */
 #endif
#endif
```

`__STDC_LIB_EXT1__` The integer constant 201ymmL, intended to indicate support for the extensions defined in annex K (Bounds-checking interfaces).

This checks for the C11 optional language features in Annex K.



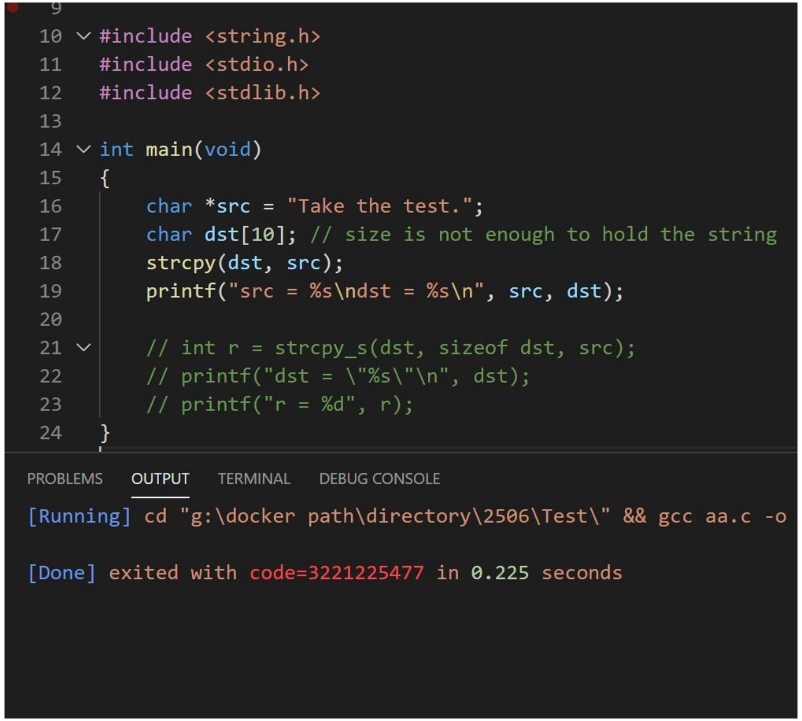
G: > docker path > directory > 2506 > Test > aa.c > main(void)
Error value
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE
success
error

```c


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

int main(void)
{
    char *src = "Take the test.";
    //  src[0] = 'M' ; // this would be undefined behavior
    char dst[strlen(src) + 1]; // +1 to accomodate for the null terminator
    strcpy(dst, src);
    dst[0] = 'M'; // OK
    printf("src = %s\ndst = %s\n", src, dst);

    int r = strcpy_s(dst, sizeof dst, src);
    printf("dst = \"%s\", r = %d\n", dst, r);
    r = strcpy_s(dst, sizeof dst, "Take even more tests.");
    printf("dst = \"%s\", r = %d\n", dst, r);

}
```

```
[Running] cd "g:\docker path\directory\2506\Test\" && gcc aa.c -o aa && "g:\docker pa
src = Take the test.
dst = Make the test.
dst = "Take the test.", r = 0
dst = "", r = 34

[Done] exited with code=0 in 0.224 seconds
```

Instruction example

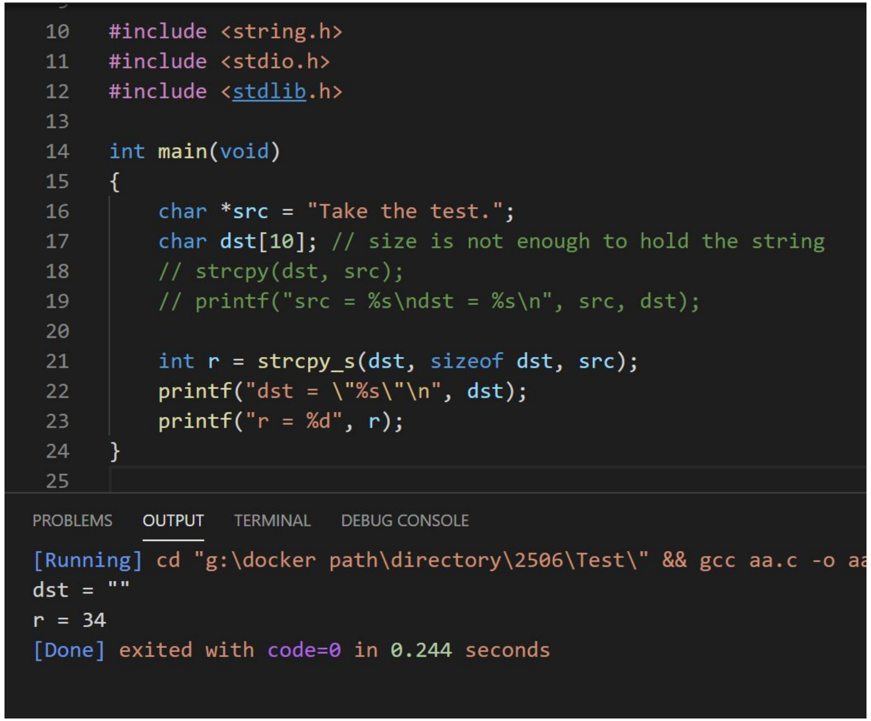
strcpy()



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

int main(void)
{
    char *src = "Take the test.";
    char dst[10]; // size is not enough to hold the string
    strcpy(dst, src);
    printf("src = %s\ndst = %s\n", src, dst);

    // int r = strcpy_s(dst, sizeof dst, src);
    // printf("dst = \"%s\"\n", dst);
    // printf("r = %d", r);
}
```

```
[Running] cd "g:\docker path\directory\2506\Test\" && gcc aa.c -o

[Done] exited with code=3221225477 in 0.225 seconds
```

strcpy_s()

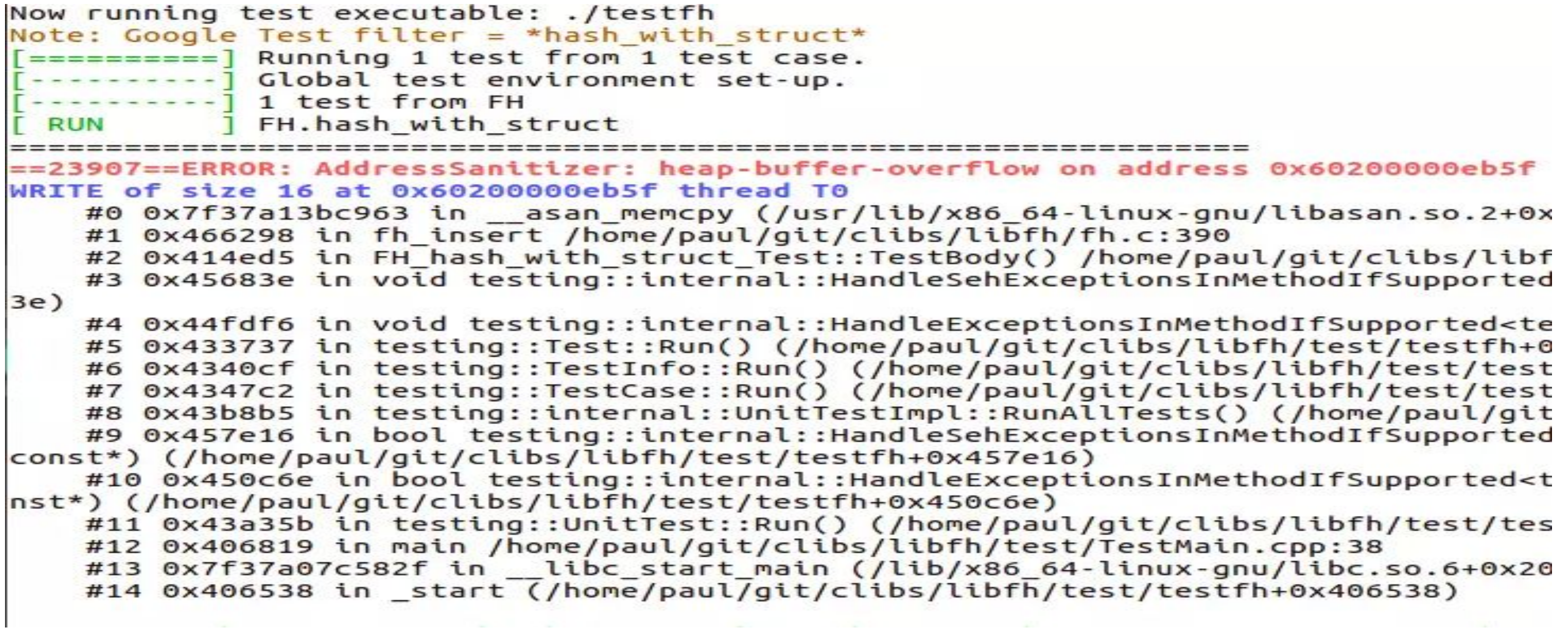


```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *src = "Take the test.";
    char dst[10]; // size is not enough to hold the string
    // strcpy(dst, src);
    // printf("src = %s\ndst = %s\n", src, dst);

    int r = strcpy_s(dst, sizeof dst, src);
    printf("dst = \"%s\"\n", dst);
    printf("r = %d", r);
}

```

```
[Running] cd "g:\docker path\directory\2506\Test\" && gcc aa.c -o aa
dst = ""
r = 34
[Done] exited with code=0 in 0.244 seconds
```

Alternative Ways In Linux

- **Valgrind is a tool used for detecting memory bugs.**
- **C11 offers conditional support for bound checking interfaces.**



```
Now running test executable: ./testfh
Note: Google Test filter = *hash_with_struct*
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FH
[ RUN      ] FH.hash_with_struct
=================================================================
==23907==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000eb5f
WRITE of size 16 at 0x60200000eb5f thread T0
    #0 0x7f37a13bc963 in __asan_memcpy (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x
    #1 0x466298 in fh_insert /home/paul/git/clibs/libfh/fh.c:390
    #2 0x414ed5 in FH_hash_with_struct_Test::TestBody() /home/paul/git/clibs/libf
    #3 0x45683e in void testing::internal::HandleSehExceptionsInMethodIfSupported
3e)
    #4 0x44fdf6 in void testing::internal::HandleExceptionsInMethodIfSupported<te
    #5 0x433737 in testing::Test::Run() (/home/paul/git/clibs/libfh/test/testfh+0
    #6 0x4340cf in testing::TestInfo::Run() (/home/paul/git/clibs/libfh/test/test
    #7 0x4347c2 in testing::TestCase::Run() (/home/paul/git/clibs/libfh/test/test
    #8 0x43b8b5 in testing::internal::UnitTestImpl::RunAllTests() (/home/paul/git
    #9 0x457e16 in bool testing::internal::HandleSehExceptionsInMethodIfSupported
const*) (/home/paul/git/clibs/libfh/test/testfh+0x457e16)
    #10 0x450c6e in bool testing::internal::HandleExceptionsInMethodIfSupported<t
nst*) (/home/paul/git/clibs/libfh/test/testfh+0x450c6e)
    #11 0x43a35b in testing::UnitTest::Run() (/home/paul/git/clibs/libfh/test/tes
    #12 0x406819 in main /home/paul/git/clibs/libfh/test/TestMain.cpp:38
    #13 0x7f37a07c582f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x20
    #14 0x406538 in _start (/home/paul/git/clibs/libfh/test/testfh+0x406538)
```

Alternative ways with String Libraries

- SafeC Library
  - Only contains safe string functions
- Slibc Library
  - Complete implementation of bounds-checking interfaces
- BSD systems
  - strlcpy/strlcat have the similar functionality
  - Implemented in libbsd for Linux

Why should NOT use them

- **Not address the problem**

  Hide problems instead of solving problems

- **Make code unportable**

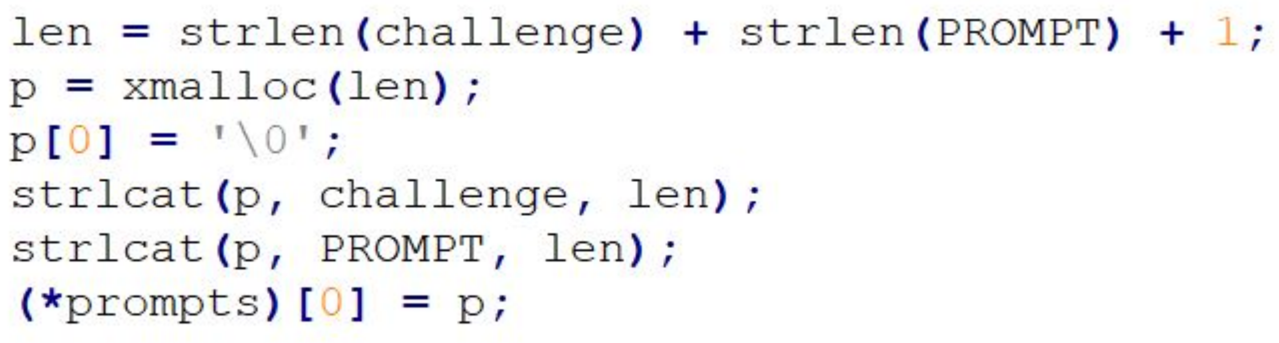
  Unimplemented in glibc on Linux.

- **Impact performance**

  Require extra computations

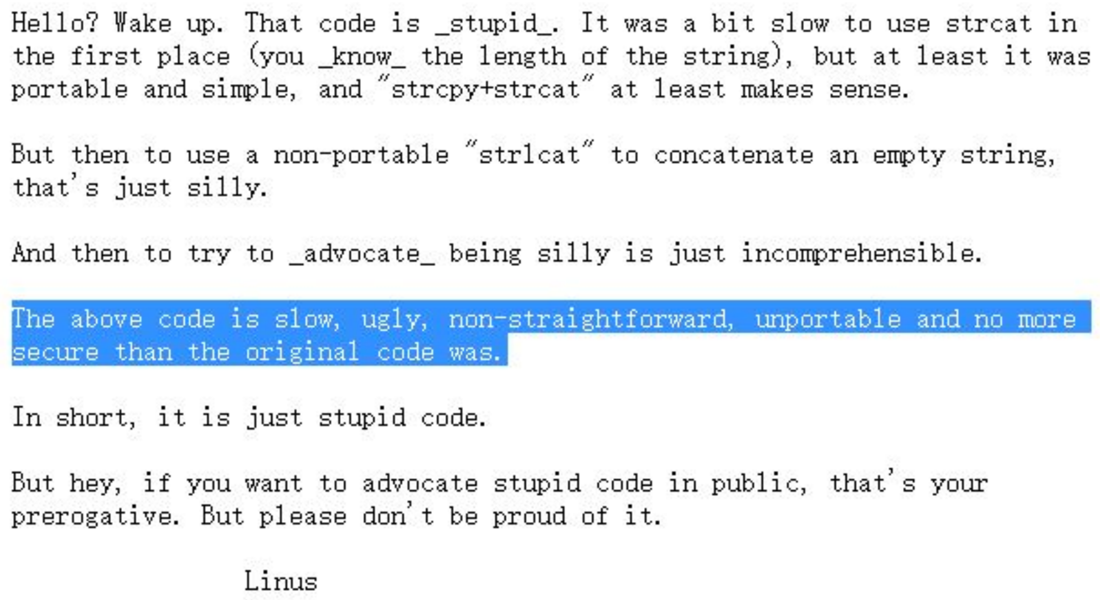
Why should NOT use them

- **Unnecessary uses**



```c
len = strlen(challenge) + strlen(PROMPT) + 1;
p = xmalloc(len);
p[0] = '\0';
strlcat(p, challenge, len);
strlcat(p, PROMPT, len);
(*prompts)[0] = p;
```

Why should NOT use them



Hello? Wake up. That code is _stupid_. It was a bit slow to use strcat in the first place (you _know_ the length of the string), but at least it was portable and simple, and "strcpy+strcat" at least makes sense.

But then to use a non-portable "strlcat" to concatenate an empty string, that's just silly.

And then to try to _advocate_ being silly is just incomprehensible.

The above code is slow, ugly, non-straightforward, unportable and no more secure than the original code was.

In short, it is just stupid code.

But hey, if you want to advocate stupid code in public, that's your prerogative. But please don't be proud of it.

Linus

Discussions

- Do you think bounds-checking interfaces are necessary?

- Are you willing to use them?

References

https://stackoverflow.com/questions/1694036/why-is-the-gets-function-so-dangerous-that-it-should-not-be-used

https://zhuanlan.zhihu.com/p/62471331

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1967.htm

https://sourceware.org/legacy-ml/libc-alpha/2002-01/msg00133.html

https://www.iso.org/standard/57853.html