rules.sonarsource.com

C static code analysis: Using "strcat" or "wcscat" is security-sensitive

3-4 minutes

In C, a string is just a buffer of characters, normally using the `null` character as a sentinel for the end of the string. This means that the developer has to be aware of low-level details such as buffer sizes or having an extra character to store the final `null` character. Doing that correctly and consistently is notoriously difficult and any error can lead to a security vulnerability, for instance, giving access to sensitive data or allowing arbitrary code execution.

The function `char *strcat( char *restrict dest, const char *restrict src );` appends the characters of string `src` at the end of `dest`. The `wcscat` does the same for wide characters and should be used with the same guidelines.

Note: the functions `strncat` and `wcsncat` might look like attractive safe replacements for `strcat` and `wcscaty`, but they have their own set of issues (see {rule:cpp:S5815}), and you should probably prefer another more adapted alternative.

Ask Yourself Whether

- There is a possibility that either the `src` or the `dest` pointer is `null`
- The current string length of `dest` plus the current string length

of `src` plus 1 (for the final `null` character) is larger than the size of the buffer pointer-to by `src`

- There is a possibility that either string is not correctly `null`-terminated

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- C11 provides, in its annex K, the `strcat_s` and the `wcscat_s` that were designed as safer alternatives to `strcat` and `wcscat`. It’s not recommended to use them in all circumstances, because they introduce a runtime overhead and require to write more code for error handling, but they perform checks that will limit the consequences of calling the function with bad arguments.
- Even if your compiler does not exactly support annex K, you probably have access to similar functions
- If you are writing C++ code, using `std::string` to manipulate strings is much simpler and less error-prone

Sensitive Code Example

```
int f(char *src) {
  char dest[256];
  strcpy(dest, "Result: ");
  strcat(dest, src); // Sensitive: might overflow
  return doSomethingWith(dest);
}
```

Compliant Solution

```
int f(char *src) {
  char result[] = "Result: ";
  char *dest = malloc(sizeof(result) + strlen(src)); // Not need of +1 for final 0 because sizeof will already count one 0
  strcpy(dest, result);
  strcat(dest, src); // Compliant: the buffer size was carefully crafted
  int r = doSomethingWith(dest);
  free(dest);
  return r;
}
```

See

- OWASP Top 10 2021 Category A6 - Vulnerable and Outdated Components
- OWASP Top 10 2017 Category A9 - Using Components with Known Vulnerabilities
- MITRE, CWE-120 - Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CERT, STR07-C. - Use the bounds-checking interfaces for string manipulation

Available In: