APPLICATION SECURITY | ECOSYSTEMS | OPEN SOURCE

# An unintimidating introduction to the dark arts of C/C++ vulnerabilities

Aviad Hahami
April 15, 2022



LUMOS!

As Synk announces its support of unmanaged dependencies (mostly C/C++ libraries), we thought it would be beneficial to introduce our non-C community to some common, high-risk dangers that lurk in the C world (get it?). Think of this as a "beginners guide" to C and C++ vulnerabilities, how they look, what problems they may cause, and how to fix them.

## Magnificent vulnerabilities and where to find them

C and C++ (for the rest of this post we'll use "C" to refer to both) are considered low-level programming languages. When comparing low-level languages to high-level languages (JS, Python, etc.), we see that the machine's memory management process is the main difference. While high-level languages manage the allocation, consumption, and freeing of memory, C passes the memory management responsibility to the developer. This allows developers to be precise with their routines'

## Magnificent vulnerabilities and where to find them

C and C++ (for the rest of this post we'll use "C" to refer to both) are considered low-level programming languages. When comparing low-level languages to high-level languages (JS, Python, etc.), we see that the machine's memory management process is the main difference. While high-level languages manage the allocation, consumption, and freeing of memory, C passes the memory management responsibility to the developer. This allows developers to be precise with their routines' performance and implementation optimizations, but it may also introduce various problems that are unique to this domain of languages.

### Memory management (and exploitation)

Memory management (and exploitation) is a significant topic that can be difficult to understand. While I will not fully cover the internals of the memory realm in this article, I highly recommend watching the memory-layout intro here (01:33-05:32) to get the required context.

Watch video →

### BUFFER OVERFLOWS (CWE-121) AND OUT-OF-BOUNDS WRITE (CWE-787)

Buffer overflows are probably the most notorious memory-related vulnerability out there. While exploiting buffer overflows can be complicated, the vulnerability itself is simple – you overflow the buffer you were allocated. Definition-wise, "buffer overflows" usually refers to the actual *exploitation* of the vulnerability — but "stack-based buffer overflow" and "out-of-bounds write" are essentially the same weakness (and therefore we'll discuss them together).

The act of exploitation can be difficult because you can't always just "write code" into the stack (or

## BUFFER OVERFLOWS (CWE-121) AND OUT-OF-BOUNDS WRITE (CWE-787)

Buffer overflows are probably the most notorious memory-related vulnerability out there. While exploiting buffer overflows can be complicated, the vulnerability itself is simple – you overflow the buffer you were allocated. Definition-wise, "buffer overflows" usually refers to the actual *exploitation* of the vulnerability — but "stack-based buffer overflow" and "out-of-bounds write" are essentially the same weakness (and therefore we'll discuss them together).

The act of exploitation can be difficult because you can't always just "write code" into the stack (or heap) even if you overflow it. Over time, there have been attempts to harden and mitigate this problem, and few defense mechanisms have been introduced. These defense mechanisms depend on many factors, and may consider your OS, kernel features, compiler, and more when defending your code. We have mechanisms like ASLR (address space layout randomization), stack canaries, and DEP (data execution prevention), to name a few. All are aimed at preventing memory corruption bugs like buffer overflow. During runtime, failing any of these mechanisms will cause the OS to stop execution and throw a SEGFAULT, making the whole exploitation process less straightforward.

I'd like to give one example of such a vulnerability and its exploitation. Consider the following C program:

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

int main(int argc, char **argv)
  volatile int modified;
  char buffer[64];
```

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

int main(int argc, char **argv)
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
    printf("you have changed the 'modified' variable\n");
  } else {
    printf("Try again?\n");
  }
}
```

*Example courtesy of **0xRick**, as seen **on his blog.***

Looking at the code (even without knowing C), we can tell that `buffer` is allocated as a 64-character length buffer and that `modified` is an integer. Cool.

We also see on line 9 that `modified` is set to a value of 0, and on line 12, we verify whether the value is 0 or not. If you haven't already guessed, our goal is to make it not zero. 😉 On line 10, we use the `gets` function to read from stdin(=user input) to the `buffer` variable. For those unfamiliar with `gets`: it reads from stdin into a given buffer until a new line character (`\n`) is read.

Since we know (because you watched YouTube video linked above) that the stack grows into a lower address, and that due to the stack data structure, `modified` lives "under" `buffer` in the memory — if we write more than 64 characters into `buffer` we'll start overriding the value of `modified`! And

Looking at the code (even without knowing C), we can tell that `buffer` is allocated as a 64-character length buffer and that `modified` is an integer. Cool.

We also see on line 9 that `modified` is set to a value of 0, and on line 12, we verify whether the value is 0 or not. If you haven't already guessed, our goal is to make it not zero. 😉 On line 10, we use the `gets` function to read from stdin(=user input) to the `buffer` variable. For those unfamiliar with `gets`: it reads from stdin into a given buffer until a new line character (`\n`) is read.

Since we know (because you watched YouTube video linked above) that the stack grows into a lower address, and that due to the stack data structure, `modified` lives "under" `buffer` in the memory — if we write more than 64 characters into `buffer` we'll start overriding the value of `modified`! And **that** is the magical buffer overflow!

This specific example is not very harmful (as this is a demo). But imagine what could happen if you were to overwrite the value for a password variable, or for the URL that the machine targets. Mitigation is simple: the recommendation is to use the `fgets` function, which also checks the length of the input and not only the existence of the "end sequence character."

## USE AFTER FREE (CWE-416)

Use after free vulnerabilities are appropriately self-described: they occur when you use a variable reference after it was freed. This vulnerability results from a memory management fault concerning the software flow, when the variable was used **after** it was cleared, thus performing an unexpected action or having an unanticipated residual effect on the application.

To see the vulnerability and its exploitation in practice, I recommend LiveOverflow's video on how he exploits a UAF vulnerability in a challenge.

To see the vulnerability and its exploitation in practice, I recommend LiveOverflow's video on how he exploits a UAF vulnerability in a challenge.

## INTEGER OVERFLOW/UNDERFLOW (CWE-190 & CWE-191)

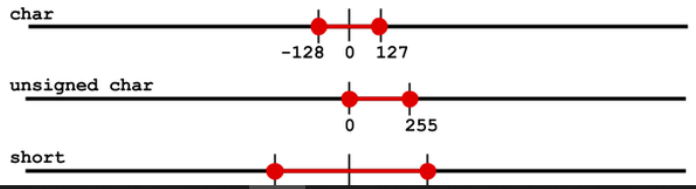
Integer overflows and underflows are two similar types of bugs (and later vulnerabilities) that occur due to number representations in computers.

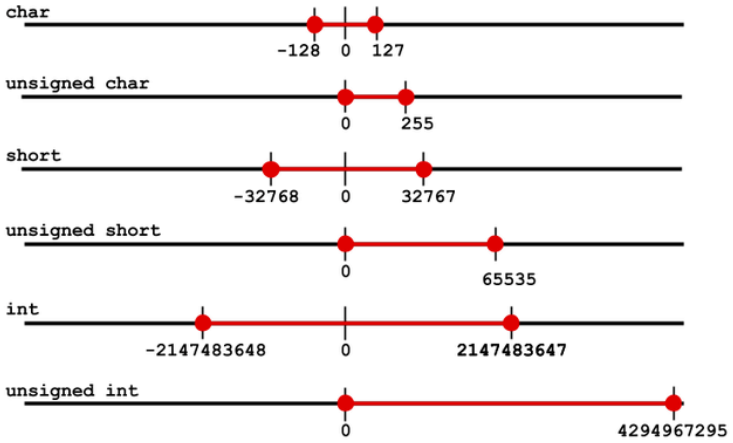
While I will not dive into variable types and how exactly numbers are represented in computers, I'll mention that there are two main representation methods for numbers: **signed** and **unsigned**. While **signed** variables may represent negative and positive numbers, **unsigned** variables store only positive numbers.

An integer **over**flow means that the value we asked the machine to store is **greater** than the maximal storable value. An integer **under**flow means we asked the machine to store a value **smaller** than the minimal value (for example, asking an unsigned integer to store a negative number).

In both cases, the result will be similar. The value will "wrap around" (that is, will start from the beginning or the end of the storable range), and thus will change its value. For an overflow, the wrapped-around value will start from 0, whereas for an underflow, it'll start from the maximal storable value (that is, the 8-bit unsigned integer will wrap around to 256 [decimal]).

This diagram illustrates the variable types and the values they can hold:








An example exploit of an integer overflow that led to a buffer overflow was caught on OpenSSH v3.3 (CVE-2002-0639).

Consider the following snippet:

```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++)
  response[i] = packet_get_string(NULL);
}
```

Suppose nresp is `1073741824` and `sizeof(char*)` is 4 (which is the typical pointer size), the `nresp*sizeof(char*)` results in an overflow (because it will wrap around and result in the value 0).

(CVE-2002-0639).

Consider the following snippet:

```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++)
  response[i] = packet_get_string(NULL);
}
```

Suppose nresp is `1073741824` and `sizeof(char*)` is 4 (which is the typical pointer size), the `nresp*sizeof(char*)` results in an overflow (because it will wrap around and result in the value 0). Therefore, `xmalloc()` receives and allocates a 0-byte buffer. The following loop causes a heap buffer overflow as we write to a non-allocated memory location, which may, in turn, be used by an attacker to execute arbitrary code.

## NULL POINTER DEREFERENCE (CWE-467)

Dereferencing is when we perform an action on a value at an address. To better explain this vulnerability, let's look at an example:

```
#include <stddef.h>

void main(){
        int *x = NULL;
        *x = 1;
}
```

According to the C standard, running the above code may result in an "**undefined behavior**." However,

### NULL POINTER DEREFERENCE (CWE-467)

Dereferencing is when we perform an action on a value at an address. To better explain this vulnerability, let's look at an example:

```
#include <stddef.h>

void main(){
        int *x = NULL;
        *x = 1;
}
```

According to the C standard, running the above code may result in an "**undefined behavior**." However, most implementations will panic with SEGFAULT, which means the software has attempted to access a restricted area of memory (thus committing a memory access violation). This results in your software being terminated by the operating system.

### OUT-OF-BOUNDS READ (CWE-125)

An out-of-bounds read occurrs when "you read outside the designated location/buffer." The outcome of such a vulnerability can be a system crash (in the best case) or disclosure of information from within your app (i.e. other users' passwords), which is not good.

For an example of such a vulnerability, here's a snippet from the PureFTPd application. Consider line 17. If the length of `s1` is greater than the length of `s2`, then — since line 8 iterates over the length of `s1` — the information we'll access on line 10 will exceed the bounds of `s2`. This will result in an out-of-bounds read.

```
int pure_memcmp(const void *const b1_, const void *const b2_, size_t len)
```

## OUT-OF-BOUNDS READ (CWE-125)

An out-of-bounds read occurrs when "you read outside the designated location/buffer." The outcome of such a vulnerability can be a system crash (in the best case) or disclosure of information from within your app (i.e. other users' passwords), which is not good.

For an example of such a vulnerability, here's a snippet from the PureFTPd application. Consider line 17. If the length of `s1` is greater than the length of `s2`, then — since line 8 iterates over the length of `s1` — the information we'll access on line 10 will exceed the bounds of `s2`. This will result in an out-of-bounds read.

```
int pure_memcmp(const void *const b1_, const void *const b2_, size_t len)
  {
   const unsigned char *b1 = (const unsigned char *) b1_;
   const unsigned char *b2 = (const unsigned char *) b2_;
   size_t i;
   unsigned char d = (unsigned char) 0 U;
   for (i = 0 U; i < len; i++)
   {
     d |= b1[i] ^ b2[i];
   }
   return (int)((1 &((d - 1) >> 8)) - 1);
}

int pure_strcmp(const char *const s1, const char *const s2)
{
  return pure_memcmp(s1, s2, strlen(s1) + 1 U);
}
```

This bug received CVE-2020-9365, and you can read the report.

```
{
  return pure_memcmp(s1, s2, strlen(s1) + 1 U);
}
```

This bug received CVE-2020-9365, and you can read the report.

## Conclusion and next steps

By now, we hope you have a (rough) understanding of what C/C++ vulnerabilities look like, where they usually reside, and what their form is. While some of these exploits may seem non-trivial at first, gaining a deeper understanding of them will improve your overall understanding of the software's deep internals and may help you avoid and prevent critical bugs.

As shown in the integer overflow explanation above, such vulnerabilities can be chained to one another, creating weak chain that is open to malicious exploitation.

Now that we've launched C/C++ for Snyk Open Source, we'll be sharing more content on topics that demonstrate how to find, exploit, and fix C and C++ vulnerabilities.