# Table of contents

1 Introduction .................................................. 3

2 Vulnerability description ..................................... 4

3 Fix for the vulnerability ....................................... 7

4 Used materials .................................................. 8
1 Introduction

ICU (International Components for Unicode) is a set of C/C++ and Java libraries which provide Unicode support for software applications. It is an open source library which means the source code for the library can be analyzed by security researchers for exploitable parts of code. One such vulnerability was recently found which could lead to an integer overflow followed by a heap-based buffer overflow and could cause a security risk for a software application using ICU libraries.

I hereby allow the University of Tartu to publicly display this report until 2025.


2 Vulnerability description

Integer overflow is a type of overflow that occurs when an integer value is incremented to a value that is too large to fit in the corresponding data type. In most architectures, an integer is a 4-byte value which means that for a signed integer the maximum value it can hold is $2^{31} - 1 = 2147483647$ since 4 bytes contains 32 bits and the one bit is used for the sign of the integer. Usually integer overflow happens after an arithmetic operation for which the result is larger than $2^{31} - 1$ and cannot be stored using 32 bits. Depending on the architecture the value can then wrap around to be a negative value or zero, which usually leads to unexpected behaviour of the application. The integer overflow in the ICU C/C++ library happens in the UnicodeString::doAppend() method. Here is a code snippet of the vulnerable section of the code.

```c
1562 }  
1563 }  
1564  
1565 int32_t oldLength = length();  
1566 int32_t newLength = oldLength + srcLength;  
1567  
1568 // Check for append onto ourself  
1569 const UChar* oldArray = getArrayStart();
```

On line 1566 two integer values are added together which represent the length of the old string and the string to be appended respectively. Since the end user of the application using ICU potentially has control of both of these variables, there is no guarantee that the sum of these variables does not exceed $2^{31} - 1$. The newLength variable is later used to clone the existing character array which leads to a heap-based buffer overflow when the array is copied using a memmove call with the source address pointing to an invalid memory address due to the overflow. Using the 65.1 version of the ICU library where this issue was not fixed, this behavior can be reproduced as follows.
In this example a large string of a random Unicode character (in this case the letter ‘n’) is constructed and then appended 16 times to the initially empty destination string. Since the length of the string is AFFFFFF in hexadecimal and 16 times AFFFFFF exceeds the maximum value of a 32-bit integer, this code fails with a segmentation fault. Compiling with the address sanitizer flags to see more information we can see that the overflow occurs when attempting to copy the values from memory using memmove and the memory is allocated on the heap using a malloc call in the allocate function.
A heap-based buffer overflow can be dangerous in multiple ways. The most obvious one is that it is possible to crash the application using this exploit to conduct a denial-of-service attack since the segmentation fault causes the application to crash. The potentially more dangerous consequence can be an execution of arbitrary code in the permissions of the application. This is particularly dangerous if the application or a service is running under root privileges. If an attacker can manipulate the strings that are used in the doAppend call then it is possible to overwrite function return pointers on the stack or overwrite entries in the global offset table of the program which both would lead to arbitrary code execution if successful.
### 3 Fix for the vulnerability

Fortunately it is rather easy to fix a integer overflow when it has been discovered. The fix is as follows

```plaintext
1566   -  int32_t newLength = oldLength + srcLength;
1566   +  int32_t newLength;
1567   +  if (uprv_add32_overflow(oldLength, srcLength, &newLength)) {
1568   +      setToBogus();
1569   +      return this;
1570   +  }
```

The old line which just added the two lengths has been removed in the newer versions of ICU and it is replaced with a function call `uprv_add32_overflow`. This function adds the first two arguments and stores the results in the third argument while returning a boolean value whether an overflow occurred or not. In case of an overflow the appending process is cancelled and the function returns early. This means that the code for copying the array does not execute and the buffer overflow does not occur.
4 Used materials

1. https://github.com/unicode-org/icu/commit/b7d08bc04a4296982fcef8b6b8a354a9e4e7afca

2. https://github.com/unicode-org/icu


5. http://site.icu-project.org/home