Märt Mäemees

CVE-2019-6250

Essay

Tartu 2019
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>The Vulnerability</td>
<td>4</td>
</tr>
<tr>
<td>The Fix</td>
<td>7</td>
</tr>
<tr>
<td>References</td>
<td>8</td>
</tr>
</tbody>
</table>
Introduction

CVE-2019-6250 is a vulnerability in the ZeroMQ library, that allows the attacker to perform remote code execution on a machine with a listening ZeroMQ socket. ZeroMQ is a lightweight messaging library that allows for communication between computers over a network, processes running in the same computer, or even threads in the same process. This can be achieved over multiple protocols, such as TCP, IPC (POSIX), inproc and so on. ZeroMQ was originally written in C/C++ and currently has six alternative ports of the engine and many more language bindings for the original engine.

The security vulnerability was reported by Guido Vranken on 8.01.2019 to the GitHub issues page of zeromq/libzmq.
The Vulnerability

Guido Vranken, the reporter and fixer of the vulnerability provided quite a specific description and an example attack in his report [1] on GitHub. The problems arises, because the message sender provides the size of the sent message themselves and therefore can set it as high as they want. To prevent processing a too large message, the library does a check on the msg_size_ value in

```cpp
if (unlikely (!_zero_copy
    || ((unsigned char *) read_pos_ + msg_size_
        > (allocator.data () + allocator.size ())))) {
```

This however, contains a problem found the the reporter, that when the value of msg_size_ is too large, the addition can overflow the pointer, making the test pass, even though msg_size_ is too large.

The attacker is allowed to send as many bytes, as the msg_size_ indicates and libzmq will copy all that data to its internal buffer, which means that if msg_size_ is larger than the buffer, the attacker is able to write into space that is outside of that buffer.

The buffer in memory is immediately followed by a `struct content_t`

```cpp
struct content_t
{
    void *data;
    size_t size;
    msg_free_fn *ffn;
    void *hint;
```
Here, `ffn` represents a function pointer that is executed with two parameters that are taken from `data` and `hint`, when the connection is closed. This means, that should the attacker know the memory location of the function they want to execute, they can overwrite the `ffn` pointer with the given location and point the `data` and `hint` pointers to the required strings.

In his report, Guido Vranken includes code, to demonstrate the exploit. It’s a small program, that begins by creating a thread, that starts infinitely listening for messages on tcp socket 6666. Then, after sleeping for a moment, the main program first sends a message to the listening thread, that using the exploit, executes `strcpy`, to copy characters from one buffer to another. (In a real attack situation, you might need to do this multiple times, to combine characters from multiple sources, into the necessary string.) Then the second message executes the `system` function with the content of `destbuffer` as the first argument, in effect executing the command in `destbuffer`.

The demo exploit, modified to run `touch test.file` (by changing the `srcbuffer` value to “touch test.file” and adding some logging):

```
  → bin l
  total 228K
  drwxr-xr-x 2 mart mart 4.0K Apr 27 17:41 .
  drwxr-xr-x 6 mart mart 4.0K Apr 27 17:40 ..
  -rwxr-xr-x 1 mart mart 217K Apr 27 17:41 at_exploit
  → bin ./at_exploit
  ZeroMQ Version: 4.3.0
  Executing: touch test.file
  → bin l
  total 228K
  drwxr-xr-x 2 mart mart 4.0K Apr 27 17:41 .
  drwxr-xr-x 6 mart mart 4.0K Apr 27 17:40 ..
  -rwxr-xr-x 1 mart mart 217K Apr 27 17:41 at_exploit
  -rw-r----- 1 mart mart 0 Apr 27 17:41 test.file
  → bin []
```
The Fix

The fix was also written by Guide Vranken in pull request #3353, which was merged for the 4.3.1 release of libzmq. He referenced a rule in the language specification:

When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements.

Therefore, he moved the `read_pos_`, to the right side of the comparator, since `allocator.data() + allocator.size()` results in a valid pointer.

```c
if (unlikely (!_zero_copy
-   || ((unsigned char *) read_pos_ + msg_size_
-   > (allocator.data () + allocator.size ())))
+   || msg_size_ >
+   (size_t)(allocator.data () + allocator.size () - read_pos_))) {
```

This should prevent the pointer overflow.

The modified demonstration program, run with the fixed version of libzmq:
References

2. The pull request: https://github.com/zeromq/libzmq/pull/3353 (27.04.2019)