4 OOB read in libuv

Share:

TIMELINE

ericsesterhenn submitted a report to Node.js. May 26th (2 years ago)

Summary:

The pointer p is read and increased without checking whether it is beyond pe, with the latter holding a pointer to the end of the buffer. This can lead to information disclosures or crashes. This function can be triggered via uv_getaddrinfo(). nodejs seems to use libuv and is possibly affected by this as well.

Description:
An out-of-bound read can occur when uv__idna_toascii() is used to convert strings to ASCII. The pointer p is read and increased without checking whether it is beyond pe, with the latter holding a pointer to the end of the buffer. This can lead to information disclosures or crashes. This function can be triggered via uv_getaddrinfo(). nodejs seems to use libuv and is possibly affected by this as well.

Steps To Reproduce:

i attached a testcase and the ad-hoc fuzzer I used to identify the issues. If you need further help reproducing, please let me know.

Code 571 Bytes Wrap lines Copy Download

```
static unsigned uv__utf8_decode1_slow(const char** p,
                                      const char* pe,
                                      unsigned a) {
  unsigned b;
  unsigned c;
  unsigned d;
  unsigned min;

  if (a > 0xF7)
    return -1;

  switch (*p - pe) {
  default:
    if (a > 0xEF) {
      if (p + 3 > pe)
        return -1;
      min = 0x10000;
      a = a & 7;
      b = (unsigned char) *(*p)++;   // OOB READ
      c = (unsigned char) *(*p)++;   // OOB READ
      d = (unsigned char) *(*p)++;   // OOB READ
      break;
    }
    /* Fall through. */
```

Impact: [add why this issue matters]

Possiblity to crash the process when untrusted hostnames are passed to uv__getaddrinfo()

Supporting Material/References:

-

Misc

This issue was found during an audit of Cure53 for ExpressVPN but ExpressVPN is not affected by the issue. I reported it to the libuv project, whose maintainers suggested that i report it to nodejs directly as well.

Impact

An oob read that does not seem to be abused to leak data, but possibly read to a guarded page which segfaults the process.

3 attachments:
F1315537: desc.txt
F1315538: fuzz.tar.bz2
F1315539: testcase_oob_read

mcollina Node.js staff changed the status to o **Triaged**. May 27th (2 years ago)

Thanks, this is confirmed by the libuv team and a fix is almost ready to go, it'll be released together with the next Node.js security release.

ericsesterhenn posted a comment. Jun 1st (2 years ago)

According to the libuv maintainers, a patch should be available by now.

mcollina Node.js staff posted a comment. Jun 1st (2 years ago)

A patch is indeed available. It will be part of the next security release that we are planning.

danbev updated CVE reference to CVE-2021-22918. Jun 21st (about 1 year ago)

In Node's case we call `uv_getaddrinfo` which gets passed a `hostname`. This is then passed to `uv__idna_toascii` and the value of the pointer to the end of the buffer is computed as the second argument:

Code 322 Bytes Wrap lines Copy Download

```
    rc = uv__idna_toascii(hostname,
                          hostname + strlen(hostname),
                          hostname_ascii,
                          hostname_ascii + sizeof(hostname_ascii));
```

Code 961 Bytes Wrap lines Copy Download

```
ong uv__idna_toascii(const char* s, const char* se, char* d, char* de) {
  const char* si;
  const char* st;
  unsigned c;
  char* ds;
  int rc;

  ds = d;

  for (si = s; si < se; /* empty */) {
    st = si;
    c = uv__utf8_decode1(&si, se);
```

Without being able to directly set the pointer to the end of the string buffer (se), is this still exploitable?

I've tried to reproduce this issue by only calling `uv_getaddrinfo` to cause some sort of side effect, like a segmentation fault or something but I've not had any luck. This following is what I've tried:

If we specify the following as the `hostname`, with the goal of getting the values correct so that the above code path is taken:

Code 81 Bytes Wrap lines Copy Download

```
  int r = uv_getaddrinfo(loop, &resolver, on_resolved, "\xF0\xd0", "80", &hints);
```

Now, if we take a look at the memory before anything has been read (before `a` is assigned in `uv__utf8_decode1`):

Code 485 Bytes Wrap lines Copy Download

```
(lldb) memory read -f d -s 1 -c 5 --num-per-line 1 *p
0x00427061: -16
0x00427062: -48
0x00427063: 0
0x00427064: 103
0x00427065: 101
```

And we can check what the end pointer (`pe`) is pointing to:

Code 167 Bytes Wrap lines Copy Download

```
(lldb) memory read -f d -c 1 pe
0x00427063: 1952802560
```

So first `a` will be assigned to the first entry and `p` incremented leaving the memory looking like this:

Code 485 Bytes Wrap lines Copy Download

```
(lldb) memory read -f d -s 1 -c 5 --num-per-line 1 *p
0x00427062: -48
0x00427063: 0
0x00427064: 103
0x00427065: 101
0x00427066: 116
```

Then in `uv__utf8_decode1_slow` we will entry the default switch clause:

Code 728 Bytes Wrap lines Copy Download

```
  default:
    if (a > 0xEF) {
      min = 0x10000;
      a = a & 7;
      b = (unsigned char) *(*p)++;
      c = (unsigned char) *(*p)++;
      d = (unsigned char) *(*p)++;
      break;
    }
```

Next, `b` will be assigned to the next value, and `p` is incremented.

Code 728 Bytes Wrap lines Copy Download

```
(lldb) expr -f d -- (unsigned char) b
(unsigned char) $45 = -48

(lldb) memory read -f d -s 1 -c 5 --num-per-line 1 *p
0x00427063: 0
```

```
0x00427067: 97
```

Next, `c` will be assigned to the next value, and `p` is incremented. Notice that the address of this value is the address of the end pointer.

Code 728 Bytes Wrap lines Copy Download

```
(lldb) expr -f d -- (unsigned char) c
(unsigned char) $46 = 0

(lldb) memory read -f d -s 1 -c 5 --num-per-line 1 *p
0x00427064: 103
0x00427065: 101
0x00427066: 116
0x00427067: 97
0x00427068: 100
```

Then, `d` will be assigned to the next value which is reading past the end pointer:

Code 161 Bytes Wrap lines Copy Download

```
(lldb) expr -f d -- (unsigned char) d
(unsigned char) $50 = 103
```

So we can see that we have read passed the end of the buffer but if we continue stepping through this function it will return and the only thing that happens is that the look up fails. Is there something I'm missing here?

ericsesterhenn posted a comment. Jun 22nd (about 1 year ago)

Reading out of bounds only causes a crash in certain situations, which usually depend on the type of memory allocator being used. A segfault is caused when reading into a memory page which the current process is not allowed to read. Whether this read continues into such a protected page mostly depends on two factors:

1) Rounding of the allocator
Most allocators (such as the one in GNU libc) have buckets (linked lists) for various memory sizes, which are used to keep track of free memory. So when you call malloc(10), you might actually get a buffer of 16 bytes. In this case, an out of bound reads will not cause any sideeffects (in regards to segfaults).

2) Guard Pages
Memory allocators use guard pages to catch out of bound accesses (some even have a guard page after every memory block allocated), so for a crash to happen, we would need to read into one of these.

Besides that, an out of bound reads might also lead to an information leak, eg giving an attacker information about a part of the memory that would normally not be accessible. I didn't investigate this option for this case.

danbev posted a comment. Jun 22nd (about 1 year ago)

@ericsesterhenn Thanks for the detailed answer, I appreciate it clarified things for me.

danbev closed the report and changed the status to **0 Resolved**. Jul 1st (about 1 year ago)

The fix for this has now been released.

danbev requested to disclose this report. Jul 1st (about 1 year ago)

ericsesterhenn agreed to disclose this report. Jul 5th (about 1 year ago)

This report has been disclosed. Jul 5th (about 1 year ago)

The Internet Bug Bounty rewarded ericsesterhenn with a **$250** bounty. Jul 6th (about 1 year ago)

Thank you for this report, we appreciate it.