Talos Vulnerability Report

TALOS-2022-1517

uClibC and uClibC-ng libpthread linuxthreads memory corruption vulnerabilities

SEPTEMBER 22, 2022

CVE NUMBER

CVE-2022-29503

SUMMARY

A memory corruption vulnerability exists in the libpthread linuxthreads functionality of uClibC 0.9.33.2 and uClibC-ng 1.0.40. Thread allocation can lead to memory corruption. An attacker can create threads to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

uClibC 0.9.33.2
uClibC-ng 1.0.40
Anker Eufy Homebase 2 2.1.8.8h

PRODUCT URLS

uClibC-ng - https://uclibc-ng.org Eufy Homebase 2 - https://us.eufylife.com/products/t88411d1 uClibC - https://www.uclibc.org/

CVSSV3 SCORE

8.1 - CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE

CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer

DETAILS

uClibC and uClibC-ng are both standalone replacements for glibc. uClibC and uClibC-ng are significantly smaller and easily portable to various architectures and embedded environments.

Libpthread is an extremely common library in a very large subset of unix-based devices, providing lightweight threading for processes. Libpthread built with uClibC using the `linuxthreads.old` implementation, or with uClibC-ng using the `linuxthreads` implementation, are both vulnerable to a memory corruption that occurs when a large number of threads are created. Both of these libraries have been used extensively in the Buildroot project with the uClibC option of `linuxthreads(stable/old)` and the uClibC-ng option of `linuxthreads` for threading implementation within the Buildroot menuconfig.

When a call to `pthread_create` occurs, `pthread_handle_create` is called during the initialization of the thread.

```
static int pthread_handle_create(pthread_t *thread, const pthread_attr_t *attr,
                 void * (*start_routine)(void *), void *arg,
                 sigset_t * mask, int father_pid,
                 int report_events,
                 td_thr_events_t *event_maskp)
{
  size_t sseg;
  int pid;
  pthread_descr new_thread;
  char * new_thread_bottom;
  char * new_thread_top;
  pthread_t new_thread_id;
  char *guardaddr = NULL;
  size_t guardsize = 0;
  int pagesize = getpagesize();
  int saved_errno = 0;

  /* First check whether we have to change the policy and if yes, whether
     we can  do this.  Normally this should be done by examining the
     return value of the sched_setscheduler call in pthread_start_thread
     but this is hard to implement.  FIXME  */
  if (attr != NULL && attr->__schedpolicy != SCHED_OTHER && geteuid () != 0)
    return EPERM;
  /* Find a free segment for the thread, and allocate a stack if needed */
  for (sseg = 2; ; sseg++)
[1]
    {
      if (sseg >= PTHREAD_THREADS_MAX)
    return EAGAIN;
      if (__pthread_handles[sseg].h_descr != NULL)
    continue;
      if (pthread_allocate_stack(attr, thread_segment(sseg), pagesize,
[2]
                                 &new_thread, &new_thread_bottom,
                                 &guardaddr, &guardsize) == 0)
      break;
      ...
```

At [1] the segments incremented as threads are created. Threads can be created up to the PTHREAD_THREADS_MAX limit, and for each thread being created the stack will be allocated using pthread_allocate_stack at [2]. thread_segment is an inlined function that is responsible for decrementing the starting address of the stack

```
static __inline__ pthread_descr thread_segment(int seg)
{
  return (pthread_descr)(THREAD_STACK_START_ADDRESS - (seg - 1) * STACK_SIZE)
         - 1;
}
```

In both of these code bases `THREAD_STACK_START_ADDRESS` is 0x40000000000m, but is a machine specific variable, and `STACK_SIZE` is 2 MB by default for all machines. This means that this condition is far more likely to be seen in 32-bit memory spaces. Once the pointer has been calculated it is passed on to `pthread_allocate_stack`.

```
  static int pthread_allocate_stack(const pthread_attr_t *attr,
                                    pthread_descr default_new_thread,
                                    int pagesize,
                                    pthread_descr * out_new_thread,
                                    char ** out_new_thread_bottom,
                                    char ** out_guardaddr,
                                    size_t * out_guardsize)
{
  pthread_descr new_thread;
  char * new_thread_bottom;
  char * guardaddr;
  size_t stacksize, guardsize;

    ...
    else {
      stacksize = STACK_SIZE - pagesize;
      if (attr != NULL)
        stacksize = MIN(stacksize, roundup(attr->__stacksize, pagesize));
      /* Allocate space for stack and thread descriptor at default address */
      new_thread = default_new_thread;
      new_thread_bottom = (char *) (new_thread + 1) - stacksize;
      if (mmap((caddr_t)((char *)(new_thread + 1) - INITIAL_STACK_SIZE),
[3]
               INITIAL_STACK_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC,
               MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED | MAP_GROWSDOWN,
               -1, 0) == MAP_FAILED)
        return -1;
```

Within `pthread_allocate_stack` as long as a stack hasn't been provided (already allocated), new allocations occur using the `mmap` call at `[3]`. This call includes the flag `MAP_FIXED` which forces mmap to take the address provided as the required address of the allocation, instead of as a hint. As `sseg` is incremented, the allocation moves to lower memory addresses for each allocation, eventually overwriting loaded libraries or even the code of the application itself.

Both vulnerabilities center around an issue while creating thread stacks using `mmap` with the `MAP_FIXED` flag. The manual page of `mmap` has the following information on `MAP_FIXED`

```
MAP_FIXED
              Don't interpret addr as a hint: place the mapping at
              exactly that address.  addr must be suitably aligned: for
              most architectures a multiple of the page size is
              sufficient; however, some architectures may impose
              additional restrictions.  If the memory region specified
              by addr and length overlaps pages of any existing
              mapping(s), then the overlapped part of the existing
              mapping(s) will be discarded.  If the specified address
              cannot be used, mmap() will fail.
```

By using `MAP_FIXED`, creating a large number of threads will remap any memory for use by the thread. This generally will first remap libraries loaded into the memory space of an application.

The vulnerable code within the latest version of uClibC is based on the `linuxthreads.old` implementation. A newer implementation based on linuxthreads is also present, and aptly named `linuxthreads`. The `linuxthreads.old` implementation is present in older versions of Buildroot and presented as `linuxthreads (stable/old)`, compared to just `linuxthreads`. All images built based on the stable/old implementation are vulnerable. Since uClibC-ng is a fork of the original uClibC code, it stands that the vulnerability was present at the time of the code fork,. Since that time, the code has diverged into unique codebases.uClibC-ng only has a single `linuxthreads` implementation within the codebase, and as such, any `linuxthreads`-based implementation of libpthread is vulnerable. All new versions of Buildroot that rely upon uClibC-ng will present this vulnerable implementation as a possible option for the threading implementation in the image.

TIMELINE

2022-05-04 - Vendor Disclosure
2022-05-17 - Vendor Disclosure
2022-05-17 - Initial Vendor Contact
2022-09-22 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.
