Sequoia: A Deep Root In Linux's Filesystem Layer

Authored by Qualys Security Advisory

Posted Jul 21, 2021

Qualys discovered a size_t-to-int conversion vulnerability in the Linux kernel's filesystem layer: by creating, mounting, and deleting a deep directory structure whose total path length exceeds 1GB, an unprivileged local attacker can write the 10-byte string "//deleted" to an offset of exactly -2GB-10B below the beginning of a vmalloc()ated kernel buffer. They successfully exploited this uncontrolled out-of-bounds write, and obtained full root privileges on default installations of Ubuntu 20.04, Ubuntu 20.10, Ubuntu 21.04, Debian 11, and Fedora 34 Workstation; other Linux distributions are certainly vulnerable, and probably exploitable. A basic proof of concept (a crasher) is attached to this advisory.

tags | exploit, kernel, local, root, proof of concept
systems | linux, debian, fedora, ubuntu
advisories | CVE-2021-33909, CVE-2021-33910
SHA-256 | 0c0b69962c7c4951fd574d5a8b85049490d77ada7568b05cfb4bce7ca40aa09a

Qualys Security Advisory

Sequoia: A deep root in Linux's filesystem layer (CVE-2021-33909)

==
Contents
==

Summary
Analysis
Exploitation overview
Exploitation details
Mitigations
Acknowledgments
Timeline

==
Summary
==

We discovered a size_t-to-int conversion vulnerability in the Linux kernel's filesystem layer: by creating, mounting, and deleting a deep directory structure whose total path length exceeds 1GB, an unprivileged local attacker can write the 10-byte string "//deleted" to an offset of exactly -2GB-10B below the beginning of a vmalloc()ated kernel buffer.

We successfully exploited this uncontrolled out-of-bounds write, and obtained full root privileges on default installations of Ubuntu 20.04, Ubuntu 20.10, Ubuntu 21.04, Debian 11, and Fedora 34 Workstation; other Linux distributions are certainly vulnerable, and probably exploitable. Our exploit requires approximately 5GB of memory and 1M inodes; we will publish it in the near future. A basic proof of concept (a crasher) is attached to this advisory and is available at:

https://www.qualys.com/research/security-advisories/

To the best of our knowledge, this vulnerability was introduced in July 2014 (Linux 3.16) by commit 058504ed ("fs/seq_file: fallback to vmalloc allocation").

==
Analysis
==

The Linux kernel's seq_file interface produces virtual files that contain sequences of records (for example, many files in /proc are seq_files, and records are usually lines). Each record must fit into a seq_file buffer, which is therefore enlarged as needed, by doubling its size at line 242 (seq_buf_alloc() is a simple wrapper around kvmalloc()):

```
------------------------------------------------------------------------
 168 ssize_t seq_read_iter(struct kiocb *iocb, struct iov_iter *iter)
 169 {
 170         struct seq_file *m = iocb->ki_filp->private_data;
 ...
 205         /* grab buffer if we didn't have one */
 206         if (!m->buf) {
 207                 m->buf = seq_buf_alloc(m->size = PAGE_SIZE);
 ...
 210         }
 ...
 220         // get a non-empty record in the buffer
 ...
 223         while (1) {
 ...
 227                 err = m->op->show(m, p);
 ...
 236                 if (!seq_has_overflowed(m)) // got it
 237                         goto Fill;
 238                 // need a bigger buffer
 ...
 240                 kvfree(m->buf);
 ...
 242                 m->buf = seq_buf_alloc(m->size <<= 1);
 ...
 246         }
------------------------------------------------------------------------
```

This size multiplication is not a vulnerability in itself, because m->size is a size_t (an unsigned 64-bit integer, on x86_64), and the system would run out of memory long before this multiplication overflows the integer m->size.

Unfortunately, this size_t is also passed to functions whose size argument is an int (a signed 32-bit integer), not a size_t. For example, the show_mountinfo() function (which is called at line 227 to format the records in /proc/self/mountinfo) calls seq_dentry() (at line 150), which calls dentry_path() (at line 530), which calls prepend() (at line 387):

```
------------------------------------------------------------------------
135 static int show_mountinfo(struct seq_file *m, struct vfsmount *mnt)
136 {
...
150                 seq_dentry(m, mnt->mnt_root, " \t\n\\");
------------------------------------------------------------------------
 523 int seq_dentry(struct seq_file *m, struct dentry *dentry, const char *esc)
 524 {
 525         char *buf;
 526         size_t size = seq_get_buf(m, &buf);
 ...
 529         if (size) {
 530                 char *p = dentry_path(dentry, buf, size);
------------------------------------------------------------------------
380 char *dentry_path(struct dentry *dentry, char *buf, int buflen)
381 {
382         char *p = NULL;
...
385         if (d_unlinked(dentry)) {
```

```
386                 p = buf + buflen;
387                 if (prepend(&p, &buflen, "//deleted", 10) != 0)
------------------------------------------------------------------------
 11 static int prepend(char **buffer, int *buflen, const char *str, int namelen)
 12 {
 13         *buflen -= namelen;
 14         if (*buflen < 0)
 15                 return -ENAMETOOLONG;
 16         *buffer -= namelen;
 17         memcpy(*buffer, str, namelen);
------------------------------------------------------------------------

As a result, if an unprivileged local attacker creates, mounts, and
deletes a deep directory structure whose total path length exceeds 1GB,
and if the attacker open()s and read()s /proc/self/mountinfo, then:

- in seq_read_iter(), a 2GB buffer is vmalloc()ated (line 242), and
  show_mountinfo() is called (line 227);

- in show_mountinfo(), seq_dentry() is called with the empty 2GB buffer
  (line 150);

- in seq_dentry(), dentry_path() is called with a 2GB size (line 530);

- in dentry_path(), the int buflen is therefore negative (INT_MIN,
  -2GB), p points to an offset of -2GB below the vmalloc()ated buffer
  (line 386), and prepend() is called (line 387);

- in prepend(), *buflen is decreased by 10 bytes and becomes a large but
  positive int (line 13), *buffer is decreased by 10 bytes and points to
  an offset of -2GB-10B below the vmalloc()ated buffer (line 16), and
  the 10-byte string "//deleted" is written out of bounds (line 17).


========================================================================
Exploitation overview
========================================================================

1/ We mkdir() a deep directory structure (roughly 1M nested directories)
whose total path length exceeds 1GB, we bind-mount it in an unprivileged
user namespace, and rmdir() it.

2/ We create a thread that vmalloc()ates a small eBPF program (via
BPF_PROG_LOAD), and we block this thread (via userfaultfd or FUSE) after
our eBPF program has been validated by the kernel eBPF verifier but
before it is JIT-compiled by the kernel.

3/ We open() /proc/self/mountinfo in our unprivileged user namespace,
and start read()ing the long path of our bind-mounted directory, thereby
writing the string "//deleted" to an offset of exactly -2GB-10B below
the beginning of a vmalloc()ated buffer.

4/ We arrange for this "//deleted" string to overwrite an instruction of
our validated eBPF program (and therefore nullify the security checks of
the kernel eBPF verifier), and transform this uncontrolled out-of-bounds
write into an information disclosure, and into a limited but controlled
out-of-bounds write.

5/ We transform this limited out-of-bounds write into an arbitrary read
and write of kernel memory, by reusing Manfred Paul's beautiful btf and
map_push_elem techniques from:

https://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-
verification

6/ We use this arbitrary read to locate the modprobe_path[] buffer in
kernel memory, and use the arbitrary write to replace the contents of
this buffer ("/sbin/modprobe" by default) with a path to our own
executable, thus obtaining full root privileges.


========================================================================
Exploitation details
========================================================================

a/ We create a directory whose total path length exceeds 1GB: in theory,
we need to create over 1GB/256B=4M nested directories (NAME_MAX is 255);
in practice, show_mountinfo() replaces each '\\' character in our long
directory with the 4-byte string "\\134", and we therefore need to
create only 1M nested directories.

b/ We fill all large vmalloc holes: we bind-mount (MS_BIND) various
parts of our long directory in several unprivileged user namespaces and
vmalloc()ate large seq_file buffers by read()ing /proc/self/mountinfo.
For example, we vmalloc()ate 768MB of large buffers in our exploit.

c/ We vmalloc()ate two 1GB buffers and one 2GB buffer (by bind-mounting
our long directory in three different user namespaces, and by read()ing
/proc/self/mountinfo), and we check that "//deleted" is indeed written
to an offset of -2GB-10B below the beginning of our 2GB buffer (i.e.,
8182B above the beginning of our first 1GB buffer -- the "XXX"s are
guard pages):

        "//deleted"
             |
      4KB    v   1GB        4KB        1GB        4KB        2GB
-----|---|---+-------------|---|-----------------|---|-----------------|
 ... |XXX| seq_file buffer |XXX| seq_file buffer |XXX| seq_file buffer |
-----|---|---+-------------|---|-----------------|---|-----------------|
         |   |                                   |
         |   \----<----<----<----<----<----<----/
        8182B              -2GB-10B

d/ We fill all small vmalloc holes: we vmalloc()ate various small socket
buffers by send()ing numerous NETLINK_USERSOCK messages. For example, we
vmalloc()ate 256MB of small buffers in our exploit.

e/ We create 1024 user-space threads; each thread starts loading an eBPF
program into the kernel, but (via userfaultfd or FUSE) we block every
thread in kernel space (at line 2101), before our eBPF programs are
actually vmalloc()ated (at line 2162):

------------------------------------------------------------------------
2076 static int bpf_prog_load(union bpf_attr *attr, union bpf_attr __user *uattr)
2077 {
....
2100         /* copy eBPF program license from user space */
2101         if (strncpy_from_user(license, u64_to_user_ptr(attr->license),
....
2161         /* plain bpf_prog allocation */
2162         prog = bpf_prog_alloc(bpf_prog_size(attr->insn_cnt), GFP_USER);
------------------------------------------------------------------------

f/ We vfree() our first 1GB seq_file buffer (where "//deleted" was
written out of bounds), and we immediately unblock all 1024 threads; our
eBPF programs are vmalloc()ated into the 1GB hole that we just vfree()d:

      4KB        1GB        4KB        1GB        4KB        2GB
-----|---|-----------------|---|-----------------|---|-----------------|
 ... |XXX|  eBPF programs  |XXX| seq_file buffer |XXX| seq_file buffer |
-----|---|-----------------|---|-----------------|---|-----------------|

g/ Next, (again via userfaultfd or FUSE) we block one of our threads (at
line 12795) after its eBPF program has been validated by the kernel eBPF
verifier but before it is JIT-compiled by the kernel:

------------------------------------------------------------------------
12640 int bpf_check(struct bpf_prog **prog, union bpf_attr *attr,
12641               union bpf_attr __user *uattr)
12642 {
.....
12795          print_verification_stats(env);
------------------------------------------------------------------------

h/ Last, we overwrite an instruction of this eBPF program with an
out-of-bounds "//deleted" string (again via our 2GB seq_file buffer),
and therefore nullify the security checks of the kernel eBPF verifier:

        "//deleted"
             |
      4KB    v   1GB        4KB        1GB        4KB        2GB
-----|---|---+-------------|---|-----------------|---|-----------------|
 ... |XXX|  eBPF programs  |XXX| seq_file buffer |XXX| seq_file buffer |
-----|---|---+-------------|---|-----------------|---|-----------------|
         |   |                                   |
         |   \----<----<----<----<----<----<----/
        8182B              -2GB-10B

First, we transform this uncontrolled eBPF-program corruption into an
information disclosure. Our first, uncorrupted eBPF program is deemed
safe by the kernel eBPF verifier ("storage" and "control" are two basic
BPF_MAP_TYPE_ARRAYs, readable and writable from user space via
BPF_MAP_LOOKUP_ELEM and BPF_MAP_UPDATE_ELEM):
```


Spoof (2,166)
SQL Injection (16,102)
TCP (2,379)
Trojan (686)
UDP (876)
Virus (662)
Vulnerability (31,136)
Web (9,365)
Whitepaper (3,729)
x86 (946)
XSS (17,494)
Other
SUSE (1,444)
Ubuntu (8,199)
UNIX (9,159)
UnixWare (185)
Windows (6,511)
Other

```
- BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE, storage) loads the
  address of our storage map (which resides in kernel space and whose
  address is unknown to us) into the eBPF register BPF_REG_2;

- BPF_MOV64_IMM(BPF_REG_2, 0) immediately replaces the contents of
  BPF_REG_2 (the address of our storage map) with the constant value 0;

- BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE, control) loads the
  address of our control map into BPF_REG_3;

- BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_2, 0) stores the contents of
  BPF_REG_2 (the constant value 0) into our control map.

However, our eBPF-program corruption overwrites the instruction
BPF_MOV64_IMM(BPF_REG_2, 0) with the 8-byte string "deleted", which
translates into the instruction BPF_ALU32_IMM(BPF_LSH, BPF_REG_5, 0x74):
a NOP ("no operation"), because our program does not use BPF_REG_5. As a
result, we do not store the constant value 0 into our control map:
instead, we store and disclose the address of our storage map.

(This information disclosure allowed us to greatly reduce the number of
hardcoded kernel offsets in our exploit: our Ubuntu 20.04 exploit worked
out of the box on Ubuntu 20.10, Ubuntu 21.04, Debian 11, and Fedora 34.)

Second, we transform our uncontrolled eBPF-program corruption into a
limited but controlled out-of-bounds write. Our second, uncorrupted eBPF
program is also deemed safe by the kernel eBPF verifier ("corrupt" is a
3*64KB BPF_MAP_TYPE_ARRAY):

- BPF_LD_IMM64_RAW(BPF_REG_4, BPF_PSEUDO_MAP_VALUE, corrupt) loads the
  address of our corrupt map into BPF_REG_4;

- BPF_ALU64_IMM(BPF_ADD, BPF_REG_4, 3*64KB/2) points BPF_REG_4 to the
  middle of our corrupt map;

- BPF_ALU64_IMM(BPF_SUB, BPF_REG_4, 3*64KB/4) points BPF_REG_4 to the
  first quarter of our corrupt map;

- BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE, control) loads the
  address of our control map into BPF_REG_3;

- BPF_LDX_MEM(BPF_H, BPF_REG_7, BPF_REG_3, 0) loads a variable 16-bit
  offset from our control map into BPF_REG_7;

- BPF_ALU64_REG(BPF_ADD, BPF_REG_4, BPF_REG_7) adds BPF_REG_7 (our
  variable 16-bit offset) to BPF_REG_4, which therefore points safely
  within the bounds of our corrupt map (because BPF_REG_7 is in the
  [0,64KB] range).

However, our eBPF-program corruption overwrites the instruction
BPF_ALU64_IMM(BPF_ADD, BPF_REG_4, 3*64KB/2) with the string "deleted",
which translates into BPF_ALU32_IMM(BPF_LSH, BPF_REG_5, 0x74) (a NOP).
As a result, the following BPF_ALU64_IMM(BPF_SUB, BPF_REG_4, 3*64KB/4)
points BPF_REG_4 out of bounds and allows us to read from and write to
the struct bpf_map that precedes our corrupt map in kernel space.

Finally, we transform this limited out-of-bounds read and write into an
arbitrary read and write of kernel memory, by reusing Manfred Paul's btf
and map_push_elem techniques:

- With the arbitrary kernel read we locate the symbol "__request_module"
  and hence the function __request_module(), disassemble this function,
  and extract the address of modprobe_path[] from the instructions for
  "if (!modprobe_path[0])".

- With the arbitrary kernel write we overwrite the contents of
  modprobe_path[] ("/sbin/modprobe" by default) with a path to our own
  executable, and call request_module() (by creating a netlink socket),
  which executes modprobe_path, and hence our own executable, as root.


========================================================================
Mitigations
========================================================================

Important note: the following mitigations prevent only our specific
exploit from working (but other exploitation techniques may exist); to
completely fix this vulnerability, the kernel must be patched.

- Set /proc/sys/kernel/unprivileged_userns_clone to 0, to prevent an
  attacker from mounting a long directory in a user namespace. However,
  the attacker may mount a long directory via FUSE instead; we have not
  fully explored this possibility, because we accidentally stumbled upon
  CVE-2021-33910 in systemd: if an attacker FUSE-mounts a long directory
  (longer than 8MB), then systemd exhausts its stack, crashes, and
  therefore crashes the entire operating system (a kernel panic).

- Set /proc/sys/kernel/unprivileged_bpf_disabled to 1, to prevent an
  attacker from loading an eBPF program into the kernel. However, the
  attacker may corrupt other vmalloc()ated objects instead (for example,
  thread stacks), but we have not investigated this possibility.


========================================================================
Acknowledgments
========================================================================

We thank the PaX Team for answering our many questions about the Linux
kernel. We also thank Manfred Paul, Jann Horn, Brandon Azad, Simon
Scannell, and Bruce Leidl for their exploits and write-ups:

https://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification
https://googleprojectzero.blogspot.com/2016/06/exploiting-recursion-in-linux-kernel_20.html
https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html
https://scannell.io/posts/ebpf-fuzzing/
https://github.com/brl/grlh

We thank Red Hat Product Security and the members of
linux-distros@openwall and security@kernel for their work on this
coordinated disclosure. We also thank Mitre's CVE Assignment Team.
Finally, we thank Marco Ivaldi for his continued support.


========================================================================
Timeline
========================================================================

2021-06-09: We sent our advisories for CVE-2021-33909 and CVE-2021-33910
to Red Hat Product Security (the two vulnerabilities are closely related
and the systemd-security mailing list is hosted by Red Hat).

2021-07-06: We sent our advisories, and Red Hat sent the patches they
wrote, to the linux-distros@openwall mailing list.

2021-07-13: We sent our advisory for CVE-2021-33909, and Red Hat sent
the patch they wrote, to the security@kernel mailing list.

2021-07-20: Coordinated Release Date (12:00 PM UTC).
```