Lab 1 Buffer Overflow

You should perform this lab using a Linux-based machine. If you have trouble on a 64-bit Ubuntu system, installing `libc6-dev-i386` may help.

1. Warmup

Open two terminal windows: a top window for running a web server, and a bottom window for exploiting that web server.
In the top window, download and decompress lab1_codes.tgz.

```
top% tar xf lab1_codes.tgz
top% cd lab1_codes
top% make
gcc -m32 -g -std=c99 -fno-stack-protector -Wall -D_GNU_SOURCE   -c -o httpd.o httpd.c
gcc -m32 -z execstack httpd.o -o httpd-ex
gcc -m32 httpd.o -o httpd-nx
gcc -m32   -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
top%
```

You should get two web server binaries, httpd-ex and httpd-nx, and two incomplete exploit scripts, exploit-ex.py and exploit-nx.py. We will provide instructions to help you complete these exploit scripts.
Start the httpd-ex web server in the top window.

```
top% ./run.sh setarch i386 -R ./httpd-ex
Web server running at localhost:4000
```

Your web server may print a different address other than localhost:4000. In that case, replace all occurrences of localhost:4000 with that printed address for the rest of this lab.
To test the web server, open a web browser and type the URL http://localhost:4000/ in the address bar. If the web server is running, you should see a "Grades" web page. You can stop the web server at any time by pressing Ctrl+C in the top window.
Again, if you saw a different web server address printed in the top window, use that address in the browser. Note that the address (especially the port number like 4000) may change every time you start the web server.
You can also view the web page via a command-line program called curl. Keep the web server running in the top window. Run the following command in the bottom window.

```
bottom% curl http://localhost:4000/app.py
```

```
<h1>Grades</h1>
<pre>
Bob         F
Alice       A
</pre>
```

The web server and clients (e.g., your browser and curl) communicate using the HTTP protocol. Here is a tutorial of the HTTP protocol. If you want to observe the details of HTTP requests and responses, add -v to curl.

```
bottom% curl -v http://localhost:4000/app.py
* About to connect() to localhost port 4000 (#0)
*   Trying 18.9.64.12... connected
> GET /app.py HTTP/1.1
> User-Agent: curl/7.22.0 ...
> Host: localhost:4000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html
<
<h1>Grades</h1>
<pre>
Bob     F
Alice       A
</pre>
* Closing connection #0
```

Bob is unhappy with the grades. He discovers that the content of the "Grades" web page is loaded from a file named grades.txt on the TA's web server. Bob then decides to remove this grades.txt file. Since he doesn't have write access to the TA's web server, Bob tries to send malicious HTTP requests over the network, which will trick the web server into removing that file. Your goal is to "help" Bob in this lab.

2. Stack Smashing

Bob's TA first runs the httpd-ex web server. The httpd-ex binary has an executable stack, which makes it easier to inject executable code into the web server. The goal of the injected code is to remove grades.txt on the server.

Start the httpd-ex web server in the top window and keep it running there.

```
top% ./run.sh setarch i386 -R ./httpd-ex
Web server running at localhost:4000
```

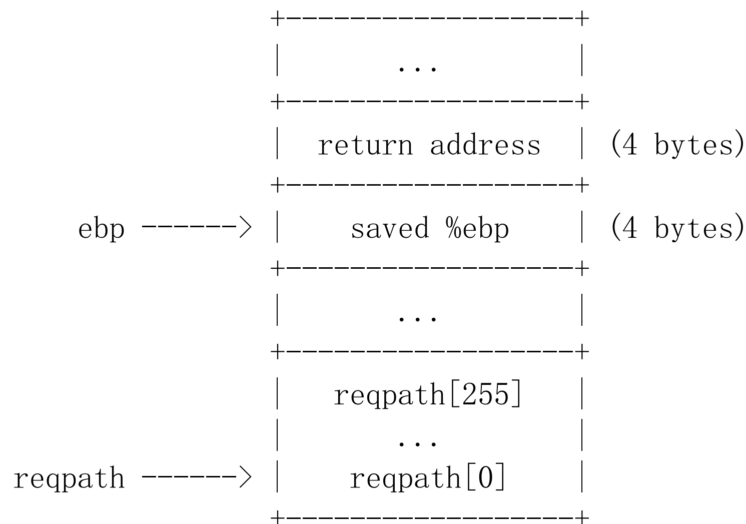
Here run.sh creates a grades.txt file every time it starts.

We have provided a modified version of Aleph One's shell code in shellcode.S for injection. View shellcode.S in an editor.

Question 1: What system call(s) does shellcode.S invoke to remove the grades.txt file?

When you run make earlier, make produces shellcode.bin, a compiled version of the shell code. We will use exploit-ex.py to inject shellcode.bin into the web server using a malicious HTTP request.

The script exploit-ex.py sends malicious HTTP requests to the web server and exploits buffer overruns in the function process_client (httpd.c:158) in httpd-ex. When executing this function, the stack looks like follows:



Normally, before calling the function process_client, the return address is pushed onto the stack (right above ebp). The local variables of process_client (such as reqpath defined at httpd.c:162) are below ebp. When the function exits, the control flow jumps to the return address saved on the stack.

Unfortunately, the web server doesn't check the length when filling up the buffer reqpath with a user-provided URL. In other words, Bob can use a very long URL to trick the web server into writing memory beyond reqpath.

Basically, the code injection in exploit-ex.py consists of three parts:

1) inject the shell code into the buffer reqpath, from bottom to top;
2) continue to fill up the memory with “x”s from bottom to top, until past ebp;
3) overwrite the return address with the address of reqpath.

In doing so, when the function process_client returns, the control flow is hijacked to execute the injected code in reqpath.

To make the exploit work, you need the values of reqpath and ebp. To get these values, either visit the web server using your browser, or run curl in the bottom window.

```
bottom% curl http://localhost:4000/app.py
```

You should find the values of reqpath and ebp printed in the top window.

Complete exploit-ex.py and fill in the values of reqpath and ebp (marked as “FIXME”). After that, run the exploit in the bottom window.

```
bottom% ./exploit-ex.py localhost:4000
```

```
HTTP request:
GET %EB%1B%5E%89...grades.txtxxx...xxx%XX%YY%ZZ%WW HTTP/1.0

Connecting to localhost:4000...
Connected, sending request...
Request sent, waiting for reply...
Received reply.
HTTP response:
...
bottom%
```

You should find a long string in the GET line (i.e., the second line in the output), in the form GET long-string HTTP/1.0. This long string is a malicious URL you crafted to help Bob remove grades.txt. Write down this malicious URL for answering the next question.

Now verify that grades.txt has been successfully removed, using ls or cat you practiced in the UNIX lab. You can also refresh your browser or re-run curl in the bottom window. This time you should see an error message.

```
bottom% curl http://localhost:4000/app.py
...
IOError: [Errno 2] No such file or directory: 'grades.txt'
...
```

Question 2: Explain the malicious URL you wrote down earlier. The URL consists of three parts: the first part starts with %EB%1B%5E%89 and ends with grades.txt; the second part is a list of "x"s, after which you should see the third part in the form %XX%YY%ZZ%WW. Where does each part come from?

3. Arc Injection

Bob's TA notices this attack and upgrades the web server to use httpd-nx, the stack of which is not executable anymore. Press Ctrl+C in the top window to stop any running web server, and start the new httpd-nx web server.

```
top% ./run.sh setarch i386 -R ./httpd-nx
Web server running at localhost:4000
```

Question 3: Save a copy of your exploit-ex.py. If http-nx prints different values of reqpath and ebp, update exploit-ex.py with the new values. Run exploit-ex.py in the bottom window. Is your exploit script able to remove grades.txt this time? Why or why not?

As the TA deploys httpd-nx, Bob tries to implement a new exploit method called arc injection (also known as return-to-libc) in exploit-nx.py. This exploit script requires no shell code. The basic idea is to hijack the return address to execute

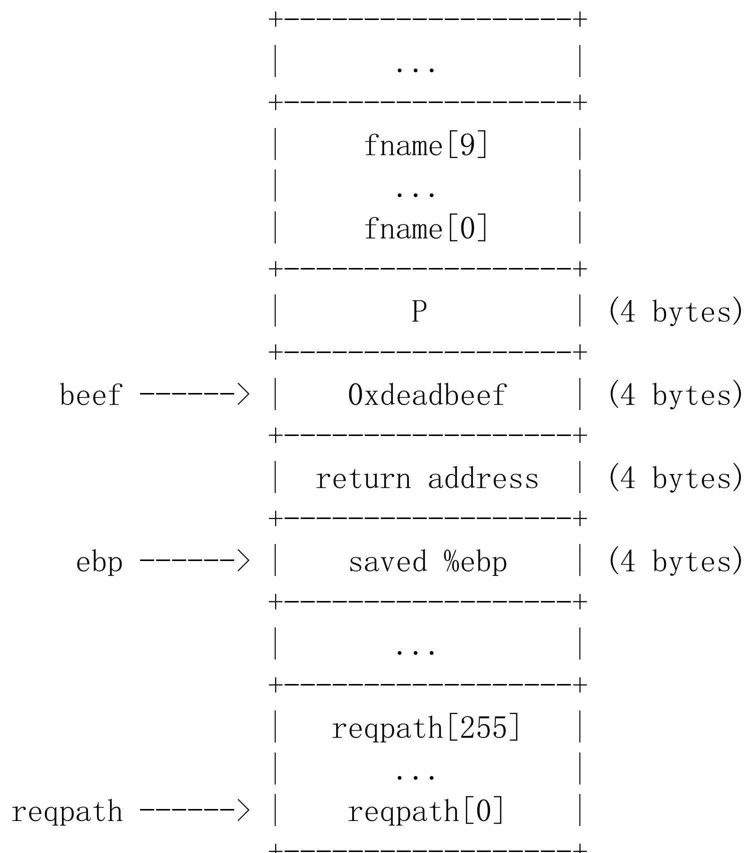
an existing function to remove grades.txt, such as unlink.
Bob hasn't completed the exploit script exploit-nx.py. In addition to reqpath and ebp, the exploit also needs the address of the unlink function. Help Bob fill in the three values marked as "FIXME" in exploit-nx.py. You can observe these values from the output of httpd-nx in the top window, either using your browser or by running curl in the bottom window.

```
bottom% curl http://localhost:4000/app.py
```

After completing exploit-nx.py, run it in the bottom window.

```
bottom% ./exploit-nx.py localhost:4000
```

Verify that this exploit has successfully removed grades.txt.
Question 4: Bob runs exploit-nx.py you wrote to trick httpd-nx into removing grades.txt. When executing the function process_client (httpd.c:158) in httpd-nx, the stack is shown as the following diagram. What are the values at the return address and P after a successful exploit? Hint: you can find these values in exploit-nx.py.



4. Beyond Stack

Linux, as well as many other operating systems, employs address space layout

randomization (ASLR) to enhance security. To observe ASLR, start the web server without setarch.

```
top% ./run.sh ./httpd-nx
Web server running at localhost:4000

bottom% curl http://localhost:4000/app.py
```

Press Ctrl+C in the top window to stop the web server. Write down the values of reqpath and ebp shown in the top window. Now repeat the above commands in both windows: restart the web server in the top window, and re-run curl in the bottom window.

Question 5: Do the values of reqpath and ebp change in the top window after restarting the web server? Does the difference “ebp - reqpath” change? If you observe any changes, do you think these changes make your exploit harder to succeed or not? Why?

The web server has other vulnerabilities. For example, the /etc/passwd file often contains sensitive user information and should not be exposed to the network. However, with a running httpd-nx web server, Bob can access that file remotely using a crafted URL.

```
top% ./run.sh ./httpd-nx
Web server running at localhost:4000

bottom% curl http://localhost:4000/[MAGIC-PATH]/etc/passwd
```

Question 6: Fill in the MAGIC-PATH part and complete the URL for retrieving the server's /etc/passwd using curl. Can you use the same URL in your browser to access that file?

When done, submit your answers to the questions and your source code for both exploit-ex.py and exploit-nx.py.

5. More Fun

Question 7 (optional): Removing grades.txt doesn't really help Bob. Bob decides to raise his grade to an A from an F by exploiting the TA's httpd-nx web server. Can you modify exploit-nx.py to help Bob achieve this goal? Hint: you may find the sed command (with the -i option) useful.

Question 8 (optional): Bob's TA discovers some strange entries in the system log, by running the following command in the top window.

```
top% dmesg -T | grep httpd-nx
 [Mon Apr 15 10:10:10 2013] httpd-nx[1234]: segfault at deadbeef ...
error 14
```

Every time Bob runs exploit-nx.py, an httpd-nx process crashes, and the system adds

one such segfault entry to the log. Can you improve exploit-nx.py to avoid such entries (i.e., do not crash httpd-nx)? Hint: fix the “beef” value.
In real world, it's trickier to make your exploits work. For example, web servers don't give away critical information (e.g., the values of reqpath and ebp). Also, many techniques have been deployed to defend against buffer overruns.

Acknowledge

This lecture is extended and modified from lecture notes by:

- Prof. Frans Kaashoek MIT 6.033 sp.13
- Specially thanks Dr. Xi Wang (TA of 6.033)