PONG

Introduction to software security

Goals of this lab:

- Get practical experience with manual and automatic code review
- Get practical experience with basic exploit development
- Get practical experience with protection against exploits
- Get practical experience with repairing vulnerable code

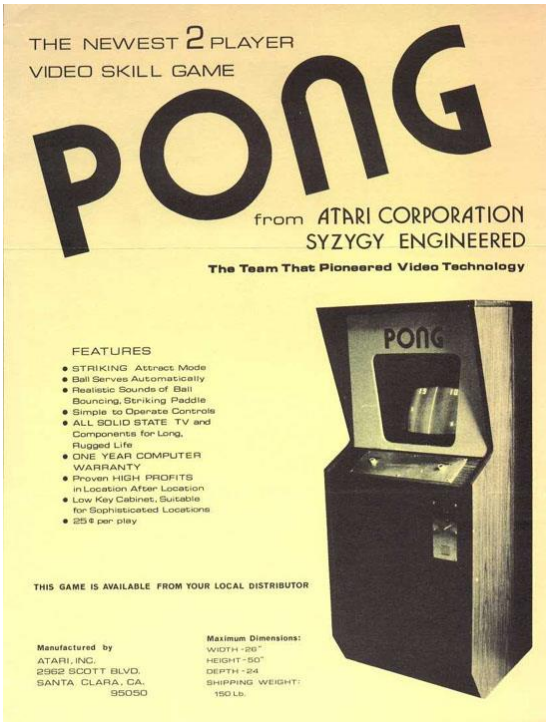
Prerequisites: A basic understanding of security in general

TABLE OF CONTENTS

MAIN LAB

In this lab you will conduct a number of different experiments on vulnerable software, including exploiting it, analyzing it, fixing it, and preventing it from causing harm. Some of the labs can be done entirely on paper. Others require access to a computer, and still others require access to a computer on which you are allowed to run exploits.

For the exercises where you run exploits or need system administrator (root) access, we will use virtualized Linux systems. However, to make things a little more interesting, you will not be told what the password for root is; you will have to exploit vulnerable software on the system to gain root access.



Part 1: Using the Linux command line

If you are familiar with Linux and the command line, you may skip this section.

This lab requires you to be able to use the Linux command line to perform basic tasks, such as editing, reading and copying files, as well as some more advanced tasks, such as compiling and debugging programs.

The shell

The *shell* in Linux is a program that interprets the commands that you type. There are many different such interpreters, the most common on Linux being bash.

The prompt

The shell prints a prompt – a string at the beginning of each line – where you can type commands. In Linux, the prompt takes different forms depending on whether the user is a normal user or has administrator privileges. If the prompt ends with a hash mark (#), then the user typically has administrator privileges.

Paths

A *path* is the name of a file. In Linux, paths consist of components separated by a forward slash (unlike e.g. Windows, which separates components using a backslash). A path that starts with a forward slash is a complete path, interpreted the same regardless of how it is used. A path that does not start with a slash is relative, and is interpreted relative the current working directory (CWD) of whatever program is executing. In the shell, you manipulate the current working directory using the cd command.

Here are some examples of paths and what they mean:

Path	CWD	Meaning
/data/kurs/adit	Doesn't matter	The file or directory *adit*, within the directory *kurs*, within the directory *data*, which in turn is a top-level directory.
kurs/adit	/data	The file or directory *adit* within the directory *kurs*, within the directory *data*, which in turn is a top-level directory.
kurs/adit	/home	The file or directory *adit* within the directory *kurs*, within the directory *home*, which in turn is a top-level directory.
../	/kurs/data/adit	The same as /kurs/data – the name .. refers to the the directory one step up.
./	/kurs/data/adit/bin	The same as /kurs/data/adit/bin – the name "." Refers to the current working directory.

Commands

To issue a command, simply type it at the prompt and hit the enter key.

There are two kinds of commands in Linux, shell built-ins and regular commands. Shell built-ins are commands that the shell itself implements. Examples include cd, exec, and set. Regular commands are simply programs stored in one of the directories that the shell searches for commands in. Examples include ls, cat and gcc. This makes it easy to add new commands to a Linux system.

If a command is not in any of the directories the shell searches for commands in, you can still issue the command by typing a complete or relative path to it that contains at least two components. For example, if the command pong is in the directory /home/user/lab, and the current working directory is /home/user, you can run pong by typing ./lab/pong.

Documentation

To get documentation about a command, simply use the man command.

Command	Purpose
man *topic*	Show the documentation for *topic*.
man -k *keyword*	Show a list of topics related to *keyword*.

Commands for manipulating files

The following commands are useful for manipulating files and directories.

Command	Purpose
touch *filename*	Change the creation date of *filename* (creating it if necessary).
pwd	Displays the current working directory.
cd *directory*	Changes the current working directory to *directory*.
ls	Lists the contents of directory. If directory is omitted, lists the contents of the current working directory. With arguments, can display information about each file (see the manual page).
cat *filename*	Display the contents of *filename*
less *filename*	Displays the contents of *filename* page-by-page (less is a so-called pager). Press the space bar to advance one page; b to go back one page; q to quit; and h for help on all commands in less.
rm *filename*	Removes the file *filename* from the file system.
mv *oldname newname*	Renames (moves) the file *oldname* to *newname*. If *newname* is an existing directory, moves *oldname* into the directory *newname*.
mkdir *dirname*	Creates a new directory named *dirname*.

Command	Purpose
rmdir *dirname*	Removes the directory *dirname*. The directory must be empty for rmdir to work.
cp *filename newname*	Creates a copy of *filename* named *newname*. If *newname* is a directory, creates a copy named filename in the directory *newname*.
chmod *modes filename*	Change permissions on *filename* according to *modes*.
chgrp *group filename*	Change the group of *filename* to *group*.
chown *user filename*	Change the owner of *filename* to *user*.

Commands for manipulating processes

In Linux, every command you run becomes a process. The following commands are useful for manipulating processes.

Command	Purpose
ps aux	List all running processes.
kill -*signal pid*	Send signal number *signal* to process with ID *pid*. Omit signal to just terminate the process. If *pid* has the form %*n*, then send signal to job *n*.
kill -9 *pid*	Send signal number 9 (SIGKILL) to process with ID *pid*. This is a last-resort method to terminate a process.
pkill *pattern*	Kill all processes that match pattern. By default, only the command name is searched for pattern.
jobs	Display running jobs.
Ctrl+C	Interrupts (terminates) the process currently in the foreground.
Ctrl+Z	Suspends the process currently running in the foreground.
Ctrl+S	Stops output in the active terminal (this is not strictly process control, but output control).
Ctrl+Q	Resumes output in the active terminal.
command &	Runs *command* in the background.
bg	Resumes a suspended process in the background.
fg	Brings a process in the background to the foreground. This will resume the process if it is currently suspended.
su	Used to run a command as root. Useful if you have to perform a task that requires root privileges while logged in as a regular user. (Will prompt for the root password.)

Part 2: The QEMU lab environment

Since this lab requires root access, the lab is conducted within a QEMU virtual machine. We use a copy-on-write setup for the lab, with a central read-only master image from which each student group creates a local slave-image. Since the local image will only contain the changes compared to the master image, we avoid each student having to copy the full image to their home directory.

Setting up the local image

We have prepared a script for you in the TDDC90 home directory, which will setup a local QEMU image in your home directory. To run the script, execute the following in a shell:

```
/courses/TDDC90/pong/scripts/setup_pong.sh
```

The script will ask for your Webreg group number (the number next to the mail icon in the leftmost column of the Webreg signup page). **It is very important that you provide the correct group number**, as this is used to generate a unique port number for connecting to the virtual machine.

The script will create a directory called "pong" in your home directory. It will contain the local image (pong.img), a number of shell scripts, and a directory called "shared", which can later be used to mount a shared directory with the virtual machine.

Installing the lab environment on your own computer (optional)

This information is only for those who want to install the lab files on their own Linux system. Note that we cannot provide technical support for problems you may encounter when running the lab on your own machine.

1. First, make sure that you have installed QEMU's x86 emulator.
2. Then, copy the directories "scripts" and "qemu" from /courses/TDDC90/pong/ to a local directory on your computer.
3. Next, edit "scripts/setup_pong.sh" and change the variable PONG_ROOT to the local directory in the above step.
4. Finally, run the modified setup script as described above to install the lab files in your local home directory.

Starting the virtual machine

In the newly created "pong" directory, you will find a script called "start_pong_group*NN*.sh", where *NN* is the group number you provided during setup. Run this script to start up the virtual machine:

```
pong/start_pong_group<i>NN</i>.sh
```

This will launch a QEMU window. After the machine has booted up (this takes up to a minute), you should see a login prompt. To log on, use *tddc90* both as username and password.

IMPORTANT: You should **immediately** set a new hard-to-guess password for your virtual machine by issuing the below command. First type the default password *tddc90* and then give the new password twice. Make sure to not forget/lose your password!

```
passwd
```

If you do not change the default password, it is trivial for someone else to either deliberately or by mistake (e.g., because they used the wrong Webreg group number) access your virtual machine while you are working on the shared ThinLinc system!

You can work with the lab directly in the QEMU window, but for convenience, we recommend that you use SSH, as described below.

Note: If you click inside the QEMU window, QEMU will "grab" the mouse for use inside the virtual machine. Since we use a non-graphical environment for the lab, this will result in the mouse pointer

disappearing. If you mistakenly click inside the QEMU window, press the *left* Control and Alt keys simultaneously to ungrab the mouse.

To power off the virtual machine, run

```
poweroff
```

in the QEMU window.

Note: A regular user can only power off the virtual machine directly from the QEMU window. The poweroff command will require the root password when run in an SSH shell.

Accessing the virtual machine using SSH

It is often convenient to access the virtual machine using SSH instead of directly in the QEMU window. This allows you to have several shells open at the same time, and to use copy-paste and scrolling in the shell. To open an SSH connection to the virtual machine, use the provided script:

```
pong/ssh_pong_group<i>NN</i>.sh
```

Use the password you set in the previous step. (The first time you SSH into the virtual machine, you will be asked to confirm that you trust the remote machine by typing "yes" and pressing enter.) You can work from the SSH shell just the same way as from the shell in the QEMU window. To close the SSH connection, issue the command "exit" in the SSH shell.

Technical side note: Our current setup does not support a fully simulated virtual network. Therefore, we instead redirect your unique port number (computed based on your Webreg group) on the host machine to port 22 on the virtual machine, and make all connections to localhost. The scripts in the pong directory are generated as a convenience for you, to make sure that the right port number is used. We recommend always using the provided scripts for connecting to the virtual machine, in order to avoid network conflicts and other problems.

Mounting a shared directory

If you want to use a GUI-enabled text editor to read or edit files on the virtual machine, you can mount a "mirror" of the home directory in the virtual machine in pong/shared as mentioned above. After the virtual machine has booted up, run (on the host, not in the virtual machine)

```
pong/mount_shared_group<i>NN</i>.sh
```

Now you can access all files in /home/tddc90 on the virtual machine through the directory *HOME*/pong/shared, where *HOME* is your home directory on the student system.

Note that you need to re-run this command every time you start the virtual machine. Also, remember to save any files you have open through pong/shared before powering off the virtual machine, otherwise you will lose your changes!

Note: Due to technical problems with the new centralized Linux environment, you may not be able to mount the shared directory again after shutting down and restarting the QEMU virtual machine. Logging out and in again should fix this problem.

Becoming root

The user with ID 0 on a Unix system is known as root, and can do nearly anything. For several of the exercises you will need to be logged in as root. There are two useful ways to do this:

- At the login prompt, log in as root instead of your normal user.
- When already logged in, issue the command su to become root.

Of course, **at this point neither option works**. You need to set the root password, and to do that you need to exploit a software vulnerability to become root.

Part 3: Introduction to the vulnerable software

For all of these exercises you will experiment with a version of ping that has been made vulnerable to some kind of attack (it's up to you to figure out what the problems are). You have access to the binary and to the source code. To prevent confusion with the real version of ping (which is an important system utility), our version is named pong, and is located in /bin/pong.

What is ping and how does it work?

ping is a utility used to check if a remote computer on a network is up and running. It works by sending out an ICMP echo request packet to the target computer and recording any ICMP echo reply packets that arrive in response. ping prints information about each packet, as well as statistics of the entire execution (round-trip-time, packet loss, etc).

It might look something like this:

```
% ping 10.0.2.2
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2 (10.0.2.2): icmp_seq=1 ttl=255 time=0.231 ms
64 bytes from 10.0.2.2 (10.0.2.2): icmp_seq=2 ttl=255 time=0.175 ms
64 bytes from 10.0.2.2 (10.0.2.2): icmp_seq=3 ttl=255 time=0.140 ms
64 bytes from 10.0.2.2 (10.0.2.2): icmp_seq=6 ttl=255 time=0.144 ms

--- 10.0.2.2 ping statistics ---
6 packets transmitted, 4 received, 33% packet loss, time 5000ms
rtt min/avg/max/mdev = 0.140/0.171/0.231/0.031 ms
```

Here, we have tested the connectivity to the virtual gateway 10.0.2.2. The ping program has sent six ICMP echo request packets, and received four replies (number 4 and 5 are missing).

Note that due to the setup of the virtual network, you will not be able to receive responses if you ping other addresses (e.g. www.liu.se). If you want to test that ping or pong works, **always use the address 10.0.2.2**.

Why might ping be vulnerable?

In order to send ICMP packets, ping must be able to open a raw IP socket – a socket on which any IP packet can be sent and where every incoming IP packet can be read (a socket is a communications endpoint within the operating system). Opening raw sockets requires privileged access – otherwise any user would be able to forge any kind of network traffic, and eavesdrop on all incoming traffic. To get privileged access, ping is a setuid binary. A setuid binary assumes the user ID of its owner every time it is run. In this case, ping is owned by root, the all-powerful superuser who can do anything on the system. Binaries that are setuid root are always dangerous, as a vulnerability could lead to an intruder gaining privileged (a.k.a. root or superuser) access to the system.

The standard ping program on Linux is not vulnerable. We have doctored pong so that it contains several exploitable vulnerabilities (and some that aren't exploitable).

Exercise 1: Familiarize yourself with ping

1-1 Run ping to test connectivity to 10.0.2.2.

1-2 What does the -I option to ping do?

Report: No report required for this exercise

Compiling and installing pong, the vulnerable ping

In several of the following exercises you will be expected to be able to compile pong, making changes to the source code and to how it is compiled. The following section is a tutorial on this process.

In the home directory /home/tddc90 on your virtual machine, there is a directory called pong. Inside this directory there are several items:

- A directory named src, which contains the source code for pong.

- A directory named exploit, which contains some generic shellcode you can use when exploiting the vulnerability in pong.
- A directory named gcc, which you will use in one of the lab exercises.

This directory is also available on /home/TDDC90/pong on the student systems.

Compiling

To compile pong, simply change the working directory to src (using the cd command), and type make. A new binary will be created in the src directory named pong. You will compile pong using the Gnu C Compiler (gcc). In order to make exploiting the software a little simpler, we have turned off optimization and the default 16-byte stack alignment.

If you need to change how pong is compiled, then edit the file named Makefile. There are comments in the file to guide you. The virtual machine has vim and nano installed, or you can use any text editor on the lab computer by mounting a shared directory, as explained earlier.

Installing

If you want to test whether a version of pong that you have built can be exploited, you should install it in /bin. You will need to be logged in as root to do this. (Note that you will not be able to log in as root until you have successfully exploited pong.) The following command will install pong, provided your working directory is the source code directory for pong:

```
make install
```

After you install your own version of pong, the original one is removed. However, you can still access it under the name pong.org.

Restoring

If at any time you want to restore the vulnerable version of pong (it is a good idea to do so after you are done with each custom version), simply issue the following command:

```
restore_pong
```

This will copy /bin/pong.org to /bin/pong, so if you have changed /bin/pong.org as well, the command won't work as intended. Note that this command is only available when you are logged in as root.

Part 4: Manual code review

The source code for pong is available under /home/TDDC90/pong and in the pong directory on the virtual machine. You will inspect this code for security flaws that need to be corrected.

In order to conduct a manual code review, you need to know two things: what to look for and how to look for it. In this lab you will get minimal guidance on these issues: it is up to you to figure them out for yourselves.

In software engineering, code inspection has been used since the late 1970s. They started to garner serious attention through the work of Michael Fagan at IBM, and became known as "Fagan Inspections". A Fagan inspection is a structured group review process that can be applied to any artifact from any process. Code is an example of an artifact, and software development an example of a process.

Besides knowing how to conduct a review, it is necessary to determine what the goals of the review are. In the case of security, the obvious goal is to find vulnerabilities in the code. However, the more specific you can be, the more effective the review is likely to be. You can use existing catalogs of vulnerability types to guide you. There are links on the course homepage that may be helpful.

Exercise 2: Manual code review

2-1 Document a procedure for code review. The procedure does not need to include *what* to look for, but should make it very clear *how* to look for security problems. You may base your process on existing processes, but ensure that what you come up with is suitable for security review.

2-2 Develop a checklist (or similar artifact) detailing what to look for in the code that can be used with the review procedure you have defined. Ensure that the checklist is suitable for inspecting the ping (or pong) program.

Hint: Focus on implementation-level bugs mentioned in the course literature and slides.

2-3 Perform a security review of your code and document all problems you find. For each problem, attempt to determine if the problem is exploitable or not. Base your analysis on the type of bug, and how much control the potential attacker has on input to the vulnerable part of the code. *To pass this lab moment you should find at least 5 security bugs that you deem exploitable*, excluding the bug you are given later in Part 6 of the lab. (There are more bugs than this in the code.) For each exploitable bug, include a brief description (a sentence or two) describing how an attacker would trigger the bug. Using the bug you exploit later in Part 6 as an example, you could mention that the bug is triggered by supplying a very long string to the "-I" option of Pong. You don't need to explain how to exploit the bugs for e.g. code execution. Include both ping.c and ping_common.c in your code review.

Report: Hand in your review procedure, checklist (or equivalent) and results of the review. You will be assessed on the suitability of your procedure, completeness and relevancy of your checklist, and quality of the review.

Part 5: Automatic code review

There are a number of useful tools available that automatically detect security problems in source code. The best tools are commercial; in this lab you will use some of the non-commercial tools, which can still be quite helpful. Both these tools are installed on your virtual machine.

Exercise 3: Use flawfinder and rats to analyze pong

3-1 Run flawfinder on the source code.

3-2 Run rats on the source code.

3-3 Compare the output from flawfinder to the output from rats. Compare both to your results from manual code inspection.

Report: Submit a report reflecting on the properties of these two tools, their strengths and weaknesses. Consider how effective they are, their false positive and false negative rates, and their general usability.

Part 6: Exploit pong

Preparation

Before attempting this exercise, you need to understand how a stack-based buffer overflow vulnerability works and can be exploited. We have simplified this exercise considerably, but you still need a firm understanding of how stack-based buffer overflows work. The paper "Smashing the stack for fun and profit" is probably a helpful source for this exercise. (You can find a link in the Additional Literature section of the course web page.)

It is also helpful to understand a bit about how Intel x86 assembly language works.

In this exercise you will gain privileged access to a system by exploiting vulnerable software that has been installed on the system. The vulnerability is a simple stack-based buffer overflow. Developing a reliable exploit can be very tricky. To simplify matters, we have done most of the work for you.

The vulnerability

The vulnerability you will exploit is in handing the -I argument to pong. The argument to -I is copied into a buffer stored on the stack (a field in the variable named ifr). Because there is no check ensuring that only as much data as fits into ifr is copied, the call may overwrite the stack, including the return address of the main function.

The relevant portion of the source code is shown below, with the vulnerability and the places where the main function returns highlighted in bold:

```
if (device) {
        memset(&ifr, 0, sizeof(ifr));
        strcpy(ifr.ifr_name, device);
        if (setsockopt(probe_fd, SOL_SOCKET, SO_BINDTODEVICE,
                        device, strlen(device)+1) == -1) {
                if (IN_MULTICAST(ntohl(dst.sin_addr.s_addr))) {
                        if (ioctl(probe_fd, SIOCGIFINDEX, &ifr) < 0) {
                                return(2);
                        }
                        memset(&imr, 0, sizeof(imr));
                        imr.imr_ifindex = ifr.ifr_ifindex;
                        if (setsockopt(probe_fd, SOL_IP, IP_MULTICAST_IF,
                                &imr, sizeof(imr)) == -1) {
                                return(2);
                        }
                }
        }
}
```

In order to exploit the vulnerability, control must reach one of the return statements. This is guaranteed to happen if the destination specified on the pong command line is a multicast address, such as 224.224.224.224. (Like with any IP address other than 10.0.2.2, you will never receive a response when using this address. However, for our purposes here, that does not matter.)

Exploiting the vulnerability

To exploit this vulnerability, you need to cause the data to be copied to consist of a NOP sled, some shellcode, followed by an address pointing into ifr, repeated enough times to overwrite the return address of main, which is stored on the stack.

There are several questions you need to answer in order to construct an exploit, foremost of which is what address to replace the return address with. In this lab, determining the return address is fairly easy. All you need to do is run pong in a debugger, using the exact same command line you would when running normally, place a breakpoint at the entry to main, and print the address of ifr, the buffer you will overwrite with malicious data. To this end we have installed gdb on your systems.

Exercise 4: Acquire the address of ifr

4-1 In a terminal window, use cd to change working directory to one that contains the source code for pong.

4-2 Load pong into the debugger using the following command:

```
gdb /bin/pong
```

You should now see a gdb prompt, where you can type gdb commands.

4-3 Place a breakpoint at the entrance to main using the following command in gdb:

```
br main
```

You should receive confirmation that a breakpoint has been set.

4-4 Run the program using the same argument as you would on the command line. For example, if your normal command line is pong -I eth0 224.224.224.224, then run the program in gdb using the following command:

```
run -I eth0 224.224.224.224
```

It is important that you use the same command line as you will when you are trying to exploit the program, as it may affect the address of ifr on the stack. It is very likely that you will have to return to this step more than once.

4-5 When the program stops at your breakpoint, print the address of ifr using the following command:

```
print &ifr
```

Make a note of the address. It is typically something along the lines of 0xbffff8c8. The 0x prefix indicates that it is hexadecimal notation.

4-6 Exit the program by using the quit command in gdb (if you want to continue running it, type “continue” or just “c”.

Report: No report is required for this exercise.

The next step is to combine shellcode with a NOP sled and the return address you found into a working exploit. This will probably not work the first time you try it, because chances are the address of ifr has changed. If this is the case, then you need to get it again using gdb.

Exercise 5: Create an exploit

5-1 Figure out how much room there is on the stack. The easiest way to do this is to run pong with a longer and longer argument to -I, until you get a crash (Segmentation fault, Illegal instruction, etc.) The longest possible argument is the amount of room there is. In order to exploit pong, you will have to construct an argument longer than this. Note that you may not be able to use all that space for your NOP sled and shellcode!

Hint: To conveniently construct a large argument to -I, you can use a Perl one-liner. For example, to use a string consisting of the letter “A” repeated 100 times:

```
pong -I $(perl -e 'print "A"x100;') 224.224.224.224
```

(For an explanation of the $(...) operator, see point 5-7 below.)

5-2 Compile the shellcode using nasm. The shellcode is located pong/exploit/shellcode.s. Explain, in your own words, what it does. You don’t need to understand in detail what every instruction does. Focus instead on the system calls used in the shellcode, and their purpose. To compile the shellcode, simply type the following command:

```
nasm -o shellcode ~/pong/exploit/shellcode.s
```

The compiled shellcode will be located in the current working directory.

5-3 Determine the size of your compiled shellcode. You can use the wc command to accomplish this:

```
wc -c shellcode
```

5-4 Determine the size of the NOP sled. Typically, a large NOP sled is desirable, but in this case it isn't necessary. Why is a large NOP sled usually preferable to a small one? What is the NOP sled for anyway?

5-5 What is the optimal return address to use, in order to make the exploit maximally reliable, given the address to ifr the you determined earlier?

5-6 Create a file containing the NOP sled, and a file with the desired return address repeated a number of times. This can be accomplished using the following commands:

```
perl -e 'print "\x90"xN;' > nopsled
perl -e 'print "\xaa\xbb\xcc\xdd"xM;' > returns
```

Here, *N* and *M* are decimal numbers, where *N* is the length of the NOP sled, *M* is the number of times to repeat the return address, and *aa*, *bb*, *cc*, and *dd* are the four bytes of the address of ifr in hexadecimal format with the least significant byte first. If you have correctly calculated the available size on the stack, using *M* = 1 should work, but repeating the return address a few times (say, about 5 times) will make the exploit a bit more reliable in case you modify the program as part of the later exercises. (Note that if you add local variables to main, the "distance" from ifr to the return address may increase, possibly causing your attack string to be too short unless you use some extra returns.)

Note that the return address must be aligned to a four-byte boundary when placed on the stack (i.e. the address where the return address is stored is divisible by 4). How can you ensure that the return address you provide in your shellcode is correctly aligned?

Hint 1: How are strings represented in C?

Hint 2: When your input is just long enough to trigger a crash, how large part of the return address is overwritten, and with what?

5-7 Run your exploit in gdb to verify that it seems to work. Start gdb, then (assuming you used the file names specified above), issue the following commands (set a breakpoint at the entry to main; one on line 252, which contains the call to strcpy; and one on line 260, which is a return statement):

```
br main
br 252
br 260
run -I $(cat nopsled shellcode returns) 224.224.224.224
```

The $(...) operator captures the output from the command inside the parentheses and "pastes" it into the command line. The effect is that pong is given an -I option consisting of the nopsled, shellcode and returns as one (very) long string. (Remember that you need to use the multicast address given above, otherwise pong will not take the desired code path.)

When execution stops the first time, at the entry to main, display the address of ifr. It must be equal to or somewhat less than the address you put into the returns file. If not, write the new address of ifr into returns, and re-run the program.

Continue running the program until it hits the breakpoint on line 252 (the line with the call to strcpy). Simply type C into gdb and press return. Now, examine the contents of the stack using the following commands.

```
x/10i &ifr
x/x $ebp
```

x is short for examine, and is used to view the contents of memory in gdb. The letter after the slash determines how gdb interprets the data, and the number how many data elements to display. For example, the first command disassembles ten instructions in memory, starting at the location of ifr. Just hit the return key to see more. The second command examines the stack starting at the address stored in register ebp (the frame

pointer), and displays it as a (32-bit) hexadecimal number. Hit return to see more. Before executing the strcpy call, the contents of ifr should be zero, which corresponds to the machine code instruction add %al,(%eax). The stack, starting at ebp should contain a stack address followed by a return address, often something like 0xb7yyyyyy. If you want to check if a particular address is a return address, use the x command. For example:

```
(gdb) x/x $ebp
0xbffff618:  0x00000000
(gdb) ↵
0xbffff61c:  0xb7e3ea63
(gdb) ↵
0xbffff620:  0x00000000
(gdb) x/i 0xb7e3ea63
 0xb7e3ea63 <__libc_start_main+243>:    mov    %eax,(%esp)
```

Here 0xb7e3ea63 looks like a return address. Using the x command again, we see that this is at line 243 of the function __libc_start_main, which is the function that called main. This is the return address you need to overwrite.

Now, execute the strcpy call. Simply type next and press return. Re-examine the contents of ifr and the stack. In ifr you should see a series of nop instructions followed by your shellcode. On the stack, you should see your desired return address repeated several times. If this is not the case, then your exploit is not constructed correctly. Check the size of your NOP sled and try again. If everything looks OK, it is time to watch the exploit in action. Continue to the next breakpoint by typing C and pressing return. This should be a return statement. Use the following command to set up a permanent display of the next instruction:

```
disp/i $eip
```

Now step through the code, instruction by instruction using the si command. You can always repeat the most recently issued command by simply hitting the return key in gdb. Continue executing one instruction at a time until you see ret. This instruction places the value at the top of the stack into the program counter. This should be your desired return address. Make sure by displaying the top of the stack:

```
x/x $esp
```

Use the si command again to execute the ret instruction. The next instruction should be a nop – part of your NOP sled. Verify that this is the case using the following command:

```
x/10i $eip
```

This shows the next ten instructions in memory. Hit return to see more. If you have indeed started to execute your NOP sled, the exploit is working as planned. If you continue execution, a new shell will be spawned, but since you are running in the debugger, it will not be a root shell.

5-8 Try to run your exploit for real using the following command line:

```
/bin/pong -I $(cat nopsled shellcode returns) 224.224.224.224
```

Before doing this you may want to alter the return address a little. It will not be the same as it was in gdb. If you have a reasonably large NOP sled, increase the return address by a few bytes. If the address of ifr increases a little compared to its value in gdb, this will improve the chances of your exploit working.

Typically, one of five things will happen: the program terminates normally; the program terminates with a segmentation fault, illegal instruction or other error; the program does not terminate at all; the program starts a shell in which the effective, but not real or saved, user ID is set to 0; or the program starts a shell in which the effective, real, and saved user ID are all set to 0 (in which case the exploit was successful).

The difference between the last two possibilities is subtle. In the first case, where only the effective user ID was changed, the exploit was a partial success: it succeeded in starting a shell, but not in permanently elevating privileges. You will still be able to change the

password for root, but not very conveniently. To see whether your exploit was completely successful or not, use the id command.

If your exploit was completely successful, you will see something like this:

```
# id
uid=0(root) gid=1000(tddc90) egid=0(root) ...
```

If you only succeeded in setting the effective user ID to root (and that is done for you when the setuid binary is started) you will see something like this:

```
# id
uid=1000(tddc90) gid=1000(tddc90) euid=0(root) ...
```

In this case, you are very close to success, and a small change to your exploit will generate complete success.

Another possibility is that your command shell will hang, or that you will get two prompts – one corresponding to your normal shell and one for a root shell. In these cases, you will have to attempt to resolve the situation. Hitting Control+C and Control+L repeatedly and randomly often clears the problem. Another possibility is that you have managed to turn off output to the terminal. If this is the case, the command stty sane followed by Control+L may clear the problem.

Explain how execution of the program has proceeded in each of these cases:

(a) Explain what has happened when the program terminates normally. Why was your exploit unsuccessful?

(b) Explain what has happened when the program terminates with a segmentation fault (or illegal instruction, bus error or similar error). Why was your exploit unsuccessful?

(c) Explain what has happened when the program starts a shell where the real and saved user IDs are your own, but the effective user ID is root.
Hint: Think about your answer to question 5-2, and reason about what may happen if your overwritten return address is a bit off.

If your exploit wasn't successful, figure out why, and repeat the process until it is. Note that this may entail going back to find the return address of ifr, as it changes depending on what you place on the command line. The location of the stack (and therefore ifr) may also be somewhat different depending on whether you use the main QEMU window or an SSH shell for starting pong.

Report: Submit your explanation of the shellcode (exercise 5-2), your answers concerning the NOP sled (exercise 5-4), the answer concerning alignment of the return address (exercise 5-6) and your analysis of the three ways execution can proceed (exercise 5-8). You will be assessed on the quality and correctness of your answers.

Exercise 6: Own the box

6-1 Use your working exploit to start a root shell.

6-2 Once an attacker has a root shell, he or she can, for example, use one of the following methods to ensure continued access to the computer:

(a) Use the passwd command to change the root password.

(b) Create a new user with user ID 0 and a known password using the adduser command (or by editing /etc/passwd and /etc/shadow).

(c) Install a rootkit.

(d) Write a new program that spawns a shell and make it setuid root.

Each method has advantages and disadvantages from an attacker's point of view. Reflect on the consequences, advantages and disadvantages of each option. *For this particular exercise, we strongly recommend that you change the root password to something you know*.

Note: The spawned root shell will not have environment variables properly set. Therefore, some commands may not function properly. We recommend that you quit the spawned shell with exit as soon as you have set a new root password. You can then launch a "proper" root shell using the su command when you need root access.

Report: Submit your reflections of the various methods to ensure continued access. You will be assessed on the quality and accuracy of your statements.

Part 7: Prevent pong from causing any harm

Preparation

You will need to have root access on the virtual machine before attempting these exercises. If you have not ensured that you can log in as root without exploiting the system, then do so before continuing.

Use the compiler to prevent exploitation

Successful exploitation using the simple methods in this lab can be prevented by the compiler. The gcc compiler has supported protection against certain types of stack-based buffer overflows for quite some time.

The gcc stack protector inserts additional code at the entry to and exit from certain functions. In /home/tddc90/pong/gcc on the virtual machines, there is a small program named sp.c that you will use for the following exercises.

Exercise 7: Exploring the gcc stack protector

7-1 Compile sp.c without generating machine code, once with and once without the stack protector enabled. You can use the following commands:

```
gcc -o sp_with.s -mpreferred-stack-boundary=4 \
        -masm=intel -fstack-protector -S \
        ~/pong/gcc/sp.c
gcc -o sp_without.s -mpreferred-stack-boundary=4 \
          -masm=intel -fno-stack-protector -S \
          ~/pong/gcc/sp.c
```

The files sp_with.s and sp_without.s now contain x86 assembly code for the sp.c program using Intel syntax (the most commonly used for Intel processors).

7-2 Extract the code at the entry and exit of the main function that implements the stack protector, and explain (in your own words) how it works. Copy the extracted code to your report and annotate the relevant parts with explanations.

Hint 1: Refer to the course slides on stack cookies, and try to figure out which parts of the assembly code that implements the different steps of the protection scheme.

Hint 2: You can use the diff command to view the differences between the two assembly files.

```
diff sp_with.s sp_without.s
```

Report: Submit your explanation of how the gcc stack protector works. You will be assessed on the quality of your explanation.

Exercise 8: Use the compiler to prevent exploitation

8-1 Re-compile pong with the stack protector turned on. To accomplish this you will need to change the Makefile so that gcc is called with the right arguments. There are comments in the Makefile to guide you.

8-2 Install and attempt to exploit pong. What happens?

8-3 Restore the Makefile so it does not call gcc with -fstack-protector and restore the original (vulnerable) version of pong using the restore_pong command.

Report: Submit your observations on this exercise. Explain what happened and why. You will be assessed on the quality and accuracy of your observations.

Randomize the stack address

The exploit you used depends on being able to predict the address of the stack. One way to prevent an attack is to ensure that this is impossible. Most modern versions of Linux, including the one powering your virtual machine, are capable of doing this.

Exercise 9: Exploring ASLR

9-1 Log in as root on the virtual machine and enable the kernel.randomize_va_space sysctl by issuing the following command:

```
sysctl -w kernel.randomize_va_space=2
```

When this sysctl is set, the Linux kernel randomizes the base address of the stack, heap, and shared libraries when starting an executable.

9-2 Start pong in gdb at least ten times, printing the address of ifr each time. What are the different addresses you get? How large is the difference between the highest and the lowest address you see? Is it possible to draw any conclusions from this concerning the magnitude of the randomization?

9-3 Explain, in your own words, at least one way to exploit a stack-based buffer overflow that does not rely on predicting the location of buffers on the stack.

9-4 Turn address space layout randomization off again using the following command:

```
sysctl -w kernel.randomize_va_space=0
```

Verify that it is off by running your exploit again.

Report: Submit your results and answers to the questions in 9-2 and 9-3. You will be assessed on the quality of your answers.

Part 8: Fix pong

You should already have fixed many of the low-level problems with pong (in the splint lab), but the vulnerable version of pong contains both code and design level flaws. First, you will fix the design-level flaw, which actually prevents exploitation. Next, you will fix all the problems in the code.

The main design level flaw in pong is that it does not relinquish its privileges as soon as possible. If it had relinquished privileges before the vulnerable code was executed, then the flaws would be impossible to exploit. The paper "Setuid Demystified" in the Additional Literature section of the course web page could be helpful when doing this exercise.

Exercise 10: Fix design-level vulnerabilities

10-1 Save a copy of ping.c named ping.c.org and a copy of ping_common.c named ping_common.c.org.

10-2 Cause pong to drop its privileges as soon as possible. You may have to rearrange the code a little, but the functionality of the program should not change (i.e. the user should not be able to notice the difference).

10-3 Create reports detailing your changes by using the following command:

```
diff -u ping.c.org ping.c
diff -u ping_common.c.org ping_common.c
```

10-4 Recompile and install pong and verify that the new version does not give a root-shell when exploited. *Also test pong to make sure that the new version can still ping 10.0.2.2 when running as a non-root user.*

10-5 Is there a security design pattern that would have been useful in implementing PONG, had it been applied from the beginning?

Report: Submit the reports showing your changes. You will be assessed on the quality and completeness of your changes.

The last step in this lab is to fix pong by eliminating all the vulnerabilities in the code.

Exercise 11: Fix code-level vulnerabilities

11-1 Save a copy of ping.c named ping.c.org and a copy of ping_common.c named ping_common.c.org.

11-2 Fix all the code-level vulnerabilities in pong (i.e. calls to unsafe functions, incorrect use of APIs etc).

11-3 Create reports detailing your changes by using the following command:

```
diff -u ping.c.org ping.c
diff -u ping_common.c.org ping_common.c
```

11-4 Recompile and install pong and verify that the new version still works, and cannot be exploited.

Report: Submit the reports showing your changes and a report explaining how you are sure you have fixed all the vulnerabilities in the code. You will be assessed on the quality and completeness of your results.

Part 9: Exploit pong again (Optional)

Preparation

Make sure to use the vulnerable version of pong when doing the exercises below

In this part you will exploit pong one more time, but using a Format String Attack. This attack lets the attacker print the contents of the process's memory and enables writing to arbitrary memory addresses.

The vulnerability

The vulnerability is reached by the same argument as above (the -I argument). The difference is that you will exploit an incorrect usage fprintf. Look at the source code below, in which the vulnerability has been highlighted.

```
if (device) {
        memset(&ifr, 0, sizeof(ifr));
        strcpy(ifr.ifr_name, device);
        if (setsockopt(probe_fd, SOL_SOCKET, SO_BINDTODEVICE, device,
strlen(device)+1) == -1) {
                if (IN_MULTICAST(ntohl(dst.sin_addr.s_addr))) {
                        struct ip_mreqn imr;
                        if (ioctl(probe_fd, SIOCGIFINDEX, &ifr) < 0) {
                                fprintf(stderr,"ping: unknown iface ");
                                fprintf(stderr, device);
                        fprintf(stderr, "\n");
                                return(2);
                        }
                        memset(&imr, 0, sizeof(imr));
                        imr.imr_ifindex = ifr.ifr_ifindex;
                        if (setsockopt(probe_fd, SOL_IP, IP_MULTICAST_IF, &imr,
sizeof(imr)) == -1) {
                                perror("ping: IP_MULTICAST_IF");
                                return(2);
                        }
                }
        }
}
```

For the same reason as in part 6 it is necessary to pass a multicast address to pong (224.224.224.224 will do). This part consists of two exercises, reading from memory and writing to memory.

Exercise 12: Read from memory

Reading from memory is straightforward, simply add some "%x" or similar and fprintf will print the contents of the stack in the chosen format.

Try executing the program to read the stack:

```
gdb pong
run -I “%x,%x,%x,%x” 224.224.224.224
```

The output will display four values from the stack in its hexadecimal representation. It should look similar to this:

```
(gdb) run -I "%x,%x,%x,%x" 224.224.224.224
Starting program: /home/ollwe447/pong/src/pong -I "%x,%x,%x,%x"
224.224.224.224
ping: unknown iface 14,4014b440,c,ffbfdbb0
```

The contents from the stack is printed in bold.

12-1 Since the stack is printed it is possible to print the arguments. Pass something extra, for example AAAA, before the %x’s and try to find it by adding more and more %x’s. A lot of %x’s might be needed.

Report: Show the assistant that you have managed to find the arguments

Exercise 13: Write to memory

The purpose of this exercise is to write to a chosen address in memory. %n will be used for writing. Whenever fprintf finds a %n in the provided format string it will write the number of characters printed into an address that is provided as an argument.

```
fprintf(“ABCD%nE”,&x);
```

The value four will be written into x because the string “ABCD” consists of four characters. If the argument &x is omitted the value will be written into an address that is found on the stack. This is the behavior that will be exploited. It will be done by inputting a string like this:

```
“\xc0\xdb\xbf\xff%x%x…%x%x%n”
```

This will write the number of characters written into the address 0xffbfdbc0 if the amount of %x’s is correct. The correct value should be one less than what is need to print the address by just using %x’s.

13-1 Use the method above to write to an address of your choosing. (**Hint:** Write to a high address like 0xffbfdbc0).

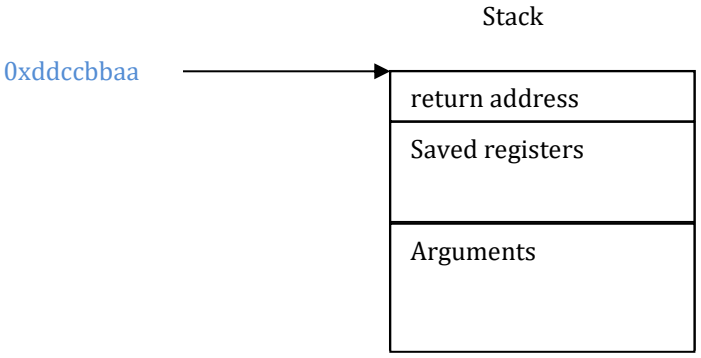
13-2 The number written is the amount of characters printed thus far. With that in mind write the value 0x140 to the chosen address.

Report: Show the assistant how you solved the tasks.

Exercise 14: Writing an address

So far you have written a small value into an unimportant address. A more useful task would be to overwrite the return address of the program to an address to provided shellcode. This introduces more issues. You need to put the shellcode somewhere and in this lab you will put it on the stack. This means that the address that you want to write, the one pointing to your shellcode, into the return address is also on the stack. The stack is located at a very high address, usually around 0xbffdfxxx. Remember that fprintf has to print the amount of characters that you want to write. If the address to your shellcode is 0xbffdfacc it means that you have to supply a string that is 4 290 763 468 (0xbffdfacc in decimal form) characters long.

A better method would be to do four writes and focus on getting one byte at a time correct, this means that you need to write the address 0xddccbbaa in the order aa, bb, cc and dd. So the format string would look like below, addresses are printed in blue and the padding A's printed in green.



0xddccbbaa	0xddccbbab	0xddccbbac	0xddccbbad	%x%x.....%x	A*y1	%n	A*y2	%n	A*y3	%n	A*y4	%n	shellcode

0xffbfdacc is the address to the beginning of the shellcode

The first four addresses points to the four bytes in the return address, these are the addresses that fprintf will use when it encounters %n. Next are the amount of %x's that is required to make the first %n write to t0xddccbbaa. After that is the amount of A's that are needed to have printed 0xcc characters by the point of the first %n. Say that 0x42 characters has been printed after the %x's. y1 will then be 0xcc – 0x42. y2 will be 0xda – y1 and so on. In the end is the shellcode which starts at address 0xffbfdacc.

14-1 Why is it necessary to write in that specific order? What would happen if you write in the order dd, cc, bb and aa?

14-2 What is the problem that occurs when calculating y3? How is it solved? (**Hint:** It is only important that the number has the smallest bits 0xbf, the other bits are unimportant).

14-3 There are now four steps left before you can exploit. You need to calculate how many characters that you need to write in total in order to write the address to your shellcode, discover the address where the return address is stored, discover the address where the shellcode is stored and overwrite the return address with the address to the shellcode.

(a) Calculate the number of characters needed to print. It is important to do this step first, why? Think about the impact that the number of characters has on task b and c.

(b) Discover the address where the return address is stored. Do this in the same way as in the stack overflow case.

(c) Discover the address where the shellcode is stored. Do this in the same way as in the stack overflow case. (**Hint:** Use gdb and print the memory with x/100x <stack address> and step through the stack. Find where the A's end).

(d) Overwrite the return address with the address to the shellcode. The addresses to where the return address is stored should be in the same place as 0xddccbbaa, 0xddccbbab… is above. The address to the shellcode is what you want to write using the %n.

Report: Explain your answers to the assistant and also how you used this vulnerability to exploit pong.