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

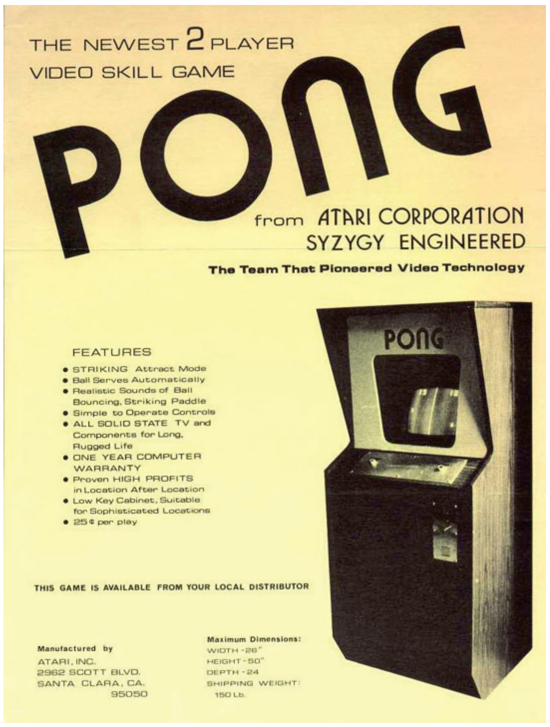
Prerequisites: A basic understanding of security in general

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. The virtualization method we use is called user-mode Linux, or simply UML. 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.


THE NEWEST 2 PLAYER
VIDEO SKILL GAME
PONG
from ATARI CORPORATION
SYZYGY ENGINEERED
The Team That Pioneered Video Technology
PONG
FEATURES
• STRIKING Attract Mode
• Ball Serves Automatically
• Realistic Sounds of Ball Bouncing, Striking Paddle
• Simple to Operate Controls
• ALL SOLID STATE TV and Components for Long, Rugged Life
• ONE YEAR COMPUTER WARRANTY
• Proven HIGH PROFITS in Location After Location
• Low Key Cabinet, Suitable for Sophisticated Locations
• 25¢ per play
THIS GAME IS AVAILABLE FROM YOUR LOCAL DISTRIBUTOR
Manufactured by
ATARI, INC.
2962 SCOTT BLVD.
SANTA CLARA, CA.
95050
Maximum Dimensions:
WIDTH - 26"
HEIGHT - 50"
DEPTH - 24
SHIPPING WEIGHT:
150 Lb.

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`. At IDA, however, the default shell is called `tcsh`.

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`, `gcc`, and `emacs`. 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*.

Command	Purpose
`mkdir` *`dirname`*	Creates a new directory named *dirname*.
`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.
CONTROL C	Interrupts (terminates) the process currently in the foreground.
CONTROL Z	Suspends the process currently running in the foreground.
CONTROL S	Stops output in the active terminal (this is not strictly process control, but output control).
CONTROL Q	Resumes output in the active terminal.
`command` `&`	Runs *command* in the background.
`bg`	Resumes a suspended process in the background. If the process needs to read from the terminal, it will be suspended again.
`fg`	Brings a process in the background to the foreground. This will resume the process if it is currently suspended.

Part 2: User-mode Linux and the mln tool

For this course and others that require root access, or where students can be expected to do strange things with the network, we use a virtualized environment that runs on a server known as **marsix.ida.liu.se**. You can access marsix from any normal lab at IDA, and from home.

Logging in

Use `ssh` to connect to marsix from the Sun lab (or from a PC, but then you'll use a GUI and not the following command):

```
ssh -X USER@marsix.ida.liu.se
```

Replace *USER* with the name of the user you want to log in as. The `-X` option may not be needed, depending on your setup. If you are using a PC you will need an X Windows server running. If you want one for your own computer, Xming is a decent (free) option.

Setting up the virtual machines

To set up the virtual machines for this lab, run

```
/data/kurs/adit/bin/setup -s tddc90/pong
```

This will launch a script that generates and configures the virtual machines for you. You only need to do this once. You will be asked which group you belong to; if you don't know your group number, ask the lab assistant. You should see something like this after a few minutes:

```
============================================================================
Initializing MLN project specification tddc90/pong.
Which group do you belong to (1-16)? 1

Building UML instances for pong. This may take several minutes.
Output from the build process is in /home/jzrmf941/mln/logs/pong.log.

At the end of the build process there may be one or more important
messages, such as the root password assigned to your UML instances.

Regular username is: jzrmf941
Password for jzrmf941 is: q6HXnUPw
Directory /home/jzrmf941/mln/files available on UML as /host

/data/kurs/adit/bin/mln start -p pong  starts all UML instances for pong.
/data/kurs/adit/bin/mln stop -p pong   stops all UML instances for pong.
----------------------------------------------------------------------------
```

Make note of the username and password that is shown in your output (they will be different than the one shown above). You'll need them to log in to the virtual machines.

The setup script creates a directory named mln in /users/*userid*, where *userid* is your user name, which in turn contains four subdirectories: conf, files, pong, and logs. The conf subdirectory contains `mln` configuration files generated by setup; files can be used to transfer files to or from the virtual machines; pong contains the scripts and filesystem images generated by `mln`, and logs contains log files from when `mln` was run.

Neither UML nor mln are perfect. For some common problems and solutions, see the course home page.

Introduction to UML and mln

This lab will be done using virtual machines: virtual computers that run as processes in the operating system. Virtual machines make it possible to run multiple operating system instances on the same physical hardware. The different instances are for all intents and purposes independent computers; they just happen to run on the same hardware.

Our virtual machines are implemented using user-mode Linux (UML), which is a port of Linux to the Linux system call interface (the hardware-dependent bits of Linux have been replaced by bits that make calls to the host operating system), and allows users to run any number of virtual systems ("UML instances", "UMLs", or "guests") under a normal Linux system (the "host") without the need for special privileges. The UML system also includes facilities for networking virtual machines.

From a user standpoint, a UML instance is just like a real machine. The work environment very much resembles a situation where a number of machines are connected to a console server and only accessible through a single text console or through the network.

To simplify setup of networks of UML instances, we use a tool called `mln`.

Exercise 1: Using the mln tool

1-1 Run /data/kurs/adit/bin/mln start –p pong to start the UML instances.

1-2 Run /data/kurs/adit/bin/mln stop –p pong to stop the UML instances.

1-3 Start the UML instances again.

1-4 Run /data/kurs/adit/bin/mln stop –p pong -H to halt your UML instance(s) instantly.

Report: No report is required for this exercise.

Accessing UML instances

When you start a UML instance using `mln`, a window is displayed containing the console of the UML. This console is effectively isolated from the host the UML is running on: you can't expect a GUI-based program to display on the host when running it in the UML console. Accessing the UML through the console is sufficient when you want to run text-based programs or commands, but not when you want to run a program with a GUI.

Exercise 2: Using user-mode Linux

2-1 Start your UML instances using `mln` and log in to each UML instance using the username and password that was shown when you ran `setup`.

2-2 Run `/sbin/ip addr list` to see what IP address(es) each UML instance has. The address you can use to access the UML instance remotely should start with "10".

2-3 Connect to your UML instance from marsix using `ssh -X` *`ADDRESS`*, where *ADDRESS* is an address of the UML instance.

2-4 Run `xlogo` in the `ssh` session you just started and check that a window containing an X-Window System logo appears on your screen. You can use this method to run any graphical program on the UML instance.

2-5 Exit the ssh session by issuing the command `exit` in the ssh session.

2-6 Shut down your UML instances by issuing the `mln stop` command (as before).

Report: No report is required for this exercise.

Transferring files between the UML and the host

The easiest way to copy files from a UML instance to the host (and vice versa) is by mounting a directory on the host in the UML instance. The UML instances in this lab automatically mount the mln/files directory in your home directory as /host on the UML. Copying a file to this directory on the host makes it available on the UML instance, and vice versa. This might not always work due to the fact that the host and UML don't necessarily share the same user IDs. If this *doesn't* work for you, you can use the `scp` command to transfer files to and from the UML via the network.

Exercise 3: Transferring files between the UML and the host

3-1 Start your UML instances again using `mln`.

3-2 On marsix, run `touch /users/`*`USER`*`/mln/files/testfile`. (Replace *USER* with your username.)

3-3 On any UML, do `ls /host`; the file testfile should be shown.

3-4 On any UML, do `cp /etc/passwd /host`; on marsix, do `cat /users/`*`USER`*`/mln/files/passwd` to display the contents of the password file you just copied. (Replace *USER* with your username.)

Report: No report is required for this exercise.

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:

1. At the login prompt, log in as root instead of your normal user.
2. 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.

Some common problems and their solutions

Neither UML nor mln are perfect. These are some of the common problems and solutions:

Problem: The computer complains about "resource temporarily unavailable" a lot, and also about something concerning IRQs.

Solution: You can safely ignore these messages.

Problem: `mln` claims that I already have UML instances started, but I don't!

Solution: First of all, double-check that you really don't have any UML instances already running. Do `pgrep -l -u` *`USER`* `linux-x32`. If you have any UML instance running, then kill them using `pkill -u` *`USER`* `linux-x32`. (Replace *USER* with your username.)

If you really don't have any UML instances running, it is likely that you terminated your UMLs abruptly (e.g. by logging out without stopping them). In such cases, mln often leaves files behind, which are later used to determine if UML instances are running. To solve the problem, use the `mln clean` command (e.g. `/data/kurs/adit/bin/mln clean -p pong`).

If that doesn't help, clean up the `mln` files manually by locating the run-time files that `mln` uses (/users/*USER*/mln/pong/uml_dir/*INSTANCENAME*), and deleting them. The following command cleans up all mln files for this lab (replace *USER* with your username):

```
rm -rf /users/USER/mln/pong/uml_dir/*/*
```

Make sure you *don't* delete the directories directly under the uml_dir directory (there is one for each UML instance).

Problem: There is no eth0 device (and no network connectivity).

Solution: Some other process may be using your virtual port in the virtual switch. This can be caused by someone else running using your group number. If you know who it is (perhaps your lab partner), have them terminate their UMLs. Otherwise alert a lab assistant. Note that in this course, network connectivity is not essential to do the labs. Simply use the 127.0.0.1 address when testing `ping` or `pong`.

Problem: The windows for the UMLs come up with `mln start`, but then disappear immediately.

Solution: You probably have UMLs running in the background. Do `pkill linux-x32` to terminate all your UMLs (this will actually try to kill *all* UMLs on the system, but you'll only succeed in terminating yours).

⚠ If you update the start scripts (e.g. to add more memory) ensure that you clean up any backup files that are created and that you do not introduce any stray line breaks in the files. Stray backup files and line breaks tend to result in weird behavior when you try to start your UMLs again.

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 www.ida.liu.se
PING informatix.ida.liu.se (130.236.177.26) 56(84) bytes of data.
64 bytes from informatix.ida.liu.se (130.236.177.26): icmp_seq=1
ttl=254 time=0.231 ms
64 bytes from informatix.ida.liu.se (130.236.177.26): icmp_seq=2
ttl=254 time=0.175 ms
64 bytes from informatix.ida.liu.se (130.236.177.26): icmp_seq=3
ttl=254 time=0.140 ms
64 bytes from informatix.ida.liu.se (130.236.177.26): icmp_seq=6
ttl=254 time=0.144 ms

--- informatix.ida.liu.se 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 www.ida.liu.se, which is also known as informatix.ida.liu.se and has the IP address 130.236.177.26. The `ping` program has sent six ICMP echo request packets, and received four replies (number 4 and 5 are missing).

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 4: Familiarize yourself with ping

4-1 Run ping to test connectivity to one or more computers.

4-2 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.

On your UML system, in your home directory, 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 tools, which you will probably not use.

This directory is also available on /home/TDDC90/pong on marsix.

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. Your UML system has `emacs`, `vim`, and `nano` installed.

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. 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 that was installed with the UML 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.

Part 4: Manual code review

The source code for `pong` is available under /home/TDDC90/pong and in the pong directory on the UML systems you will be using for most of this lab. 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 5: Manual code review

5-1 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.

5-2 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.

5-3 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.

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 UML systems and on marsix.

Exercise 6: Use flawfinder and rats to analyze pong

6-1 Run `flawfinder` on the source code.

6-2 Run `rats` on the source code.

6-3 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.

Unix has long had a tool called `lint`, which is used to detect problematic C code. There is a similar tool available today called `splint`, which augments `lint` greatly. Unlike `lint`, `splint` is quite useful as a security analysis tool. However, `splint` requires the programmer to annotate the code in order for many features to be useful.

The `splint` manual is available on the course homepage.

Exercise 7: Use splint to analyze pong (OPTIONAL)

7-1 Run `splint` on the source code for `pong`. Initially, you should use the following arguments to `splint` (others may be appropriate as well, but without these, nothing will work):

```
splint +posixlib +gnuextensions +skipsysheaders ping.c
splint +posixlib +gnuextensions +skipsysheaders ping_common.c
```

Examine the warnings you get from `splint`. As you can see, there are a lot of them, and most have nothing to do with security, but are simply the result of the source code not containing any annotations.

7-2 Fix as many of the warnings as possible in ping.c and ping_common.c without annotating the code (i.e. fix the bugs in the source code – don't just disable the warnings).

7-3 Annotate the code so that `splint` produces more useful results.

Report: Submit your fixed and annotated code, as well as the output from `splint` when run on this code. You will be assessed on the quality of your fixes, the quality of your annotations, and the results of `splint` when run on the updated version of `pong`.

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.

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.

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 8: Acquire the address of ifr

8-1 In a terminal window, use `cd` to change working directory to one that contains the source code for `pong`.

8-2 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.

8-3 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.

8-4 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 130.236.189.1`, then run the program in `gdb` using the following command:

```
run -I eth0 130.236.189.1
```

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.

8-5 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.

8-6 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 9: Create an exploit

9-1 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. 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!

9-2 Compile the shellcode using `nasm`. The shellcode is located pong/exploit/shellcode.s. Explain, in your own words, what is does. To compile it, simply type the following command on your UML system:

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

The compiled shellcode will be located in current working directory.

9-3 Determine the size of your compiled shellcode. You can use the `wc` command to accomplish this:

```
wc -c shellcode
```

9-4 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? Why doesn't it matter in this case? What is the NOP sled for anyway?

9-5 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* 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 a decent-sized NOP sled, you can use an address higher than the address of `ifr`. This will probably make the exploit a little more reliable.

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?

9-6 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
```

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 RETURN into `gdb`. Now, examine the contents of the stack using the following commands.

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

The first command disassembles ten instructions in memory, starting at the location of `ifr`. Just hit RETURN to see more. The second command examines the stack starting at the address stored in register `ebp` (the frame pointer). 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 0x40yyyyyy. If you want to check if a particular address is a return address, use the `x` command. For example:

```
(gdb) x/x $ebp
0xbffffad8:  0xbffffb28
(gdb) RETURN
0xbffffadc:  0x40032ea8
(gdb) RETURN
0xbffffae0:  0x00000000
(gdb) x/i 0x40032ea8
0x40032ea8 <__libc_start_main+200>:     mov    %eax,(%esp)
```

Here, 0xbffffb28 is a stack address (it is the value `ebp` had when `main` was called), and 0x40032ea8 looks like a return address. Using the `x` command again, we see that this is at line 200 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 S 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 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 RETURN 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.

9-7 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:

```
sh-3.1# id
uid=0(root) gid=1000(user) groups=1000(user)
```

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:

```
sh-3.1# id
uid=1000(user) gid=1000(user) euid=0(root) groups=1000(user)
```

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 doesn't terminate at all. Why was your exploit unsuccessful?

(d) 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.

9-8 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.

Report: Submit your explanation of the shellcode (exercise 9-2), your answers concerning the NOP sled (exercise 9-4), the answer concerning alignment of the return address (exercise 9-5) and your analysis of the four ways execution can proceed (exercise 9-7). You will be assessed on the quality and correctness of your answers.

Exercise 10: Own the box

10-1 Use your working exploit to start a root shell.

10-2 Ensure continued access to the computer using any one of several methods:

(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) Create a copy of /bin/sh that is setuid root (change the owner using `chown` and make it setuid using `chmod u+s` on the copy).

(d) Install a rootkit. There are three to choose from in /home/TDDC90 (adore-ng, suckit2, and mood-nt. We haven't tested any one of them and don't know if they'll work.

(e) 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 regardless of which other option (if any) you choose, you also change the root password to something you know.

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 your UML system 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/gcc (on marsix, not the UML systems), there is a small program named sp.c that you will use for the following exercises.

Exercise 11: Exploring the gcc stack protector

11-1 Compile sp.c without generating machine code, once with and once without the stack protector enabled. You can use the following commands (assuming you have copied sp.c to your working directory):

```
gcc -o sp_with.s -mpreferred-stack-boundary=2 \
                 -masm=intel -fstack-protector -S sp.c
gcc -o sp_without.s -mpreferred-stack-boundary=2 \
                 -masm=intel -fno-stack-protector -S 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).

11-2 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.

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

Exercise 12: Use the compiler to prevent exploitation

12-1 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.

12-2 Install and attempt to exploit `pong`. What happens?

12-3 Re-compile pong without stack protection (still using the same version of `gcc`). You will need to change the Makefile again.

12-4 Install and attempt to exploit `pong`. You should now see a change in behavior (but the exploit will still not be successful, due to the way this version of `gcc` manages its stack).

12-5 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. Some versions of Linux, including the one powering your UML system, are capable of doing this.

Exercise 13: Exploring randomized stack addresses

13-1 Log in as root on your UML system and enable the kernel.randomize_va_space sysctl by issuing the following command:

```
sysctl -w kernel.randomize_va_space=1
```

When this sysctl is set, the Linux kernel randomizes that top of the stack when loading ELF binaries.

13-2 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?

13-3 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.

13-4 Turn stack 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 13-2 and 13-3. 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.

Exercise 14: Fix design-level vulnerabilities

14-1 Save a copy of ping.c named ping.c.org and a copy of ping_common.c named ping_common.c.org.

14-2 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).

14-3 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
```

14-4 Recompile and install `pong` and verify that the new version cannot be exploited.

14-5 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. Answer to 14-5.

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

Exercise 15: Fix code-level vulnerabilities

15-1 Save a copy of ping.c named ping.c.org and a copy of ping_common.c named ping_common.c.org.

15-2 Fix all the code-level vulnerabilities in `pong` (i.e. calls to unsafe functions, incorrect use of APIs etc).

15-3 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
```

15-4 Recompile and install `pong` and verify that the new version 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.