Operating Systems Security – Assignment 6

Version 1.0.0 – 2015/2016

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands.

1 Compartmentalization with `chroot`

Each process in UNIX knows the root of the filesystem (typically denoted /). The **`chroot`** system call changes this root and the **`chroot`** utility starts a process with a different filesystem root. The **`chroot`** mechanism can be used for *compartmentalization*: A process that is running in a **`chroot`** environment can only access files that are below its filesystem root. It can also be used to test new systems. For example, one can install the development branch of a Linux distribution in a **`chroot`** environment and test it without having to reboot. One can also use it as a development environment; for example, it is possible to run a Debian Linux system in a **`chroot`** environment on an Android phone (after rooting the phone; see, for example, https://cryptojedi.org/misc/nexuss-debian.shtml). The **`chroot`** mechanism for compartmentalization can be used to add a certain level of security, but it has various limitations, which we will investigate in the following.

Prerequisites

A process running in a **`chroot`** environment needs various files (libraries etc.) accessible. The easiest (but not necessarily most secure) way to achieve this is to make a whole UNIX environment available. Debian Linux and derivatives allow to "install" the whole environment in a directory, for example, in **`/tmp/debian`**, as follows:
$ **`debootstrap --arch amd64 jessie /tmp/debian/ http://ftp.nl.debian.org/debian/`**
This is going to take a while; afterwards you can (as root) **`chroot`** into this enviroment by running
$ **`chroot /tmp/debian`**
The environment is a quite minimal UNIX environment, so you might want to install additional software, for example (inside the **`chroot`** environment):
$ **`apt-get install gcc`**
The **`chroot`** compartmentalization does, by design, not prevent root to break out of the **`chroot`** "jail". The way to break out of the jail, for root, involves the following steps:

1. Create a subdirectory in the current **`chroot`** environment (**`mkdir`** standard C library function);
2. (open the current working directory using the **`open`** syscall);
3. use the **`chroot`** syscall to **`chroot`** into the subdirectory created in step 1;
4. (change the working directory back to the original working directory with the **`fchdir`** syscall);
5. perform **`chdir("..")`** syscalls to change to the actual (non-**`chroot`**) root of the filesystem.

The two steps in parantheses are only required if the **`chroot`** system call also changes the working directory to the **`chroot`** directory. Note that after step 4, the process has a working directory outside the current root directory; this is what allows the process to change the working directory further up to the actual root.

Objectives

a) Create a symbolic link from somewhere inside **`/tmp/debian/`** to somewhere outside **`/tmp/debian`**. Can you follow the symbolic link when using **`/tmp/debian`** as a **`chroot`** jail?
b) Create a hard link from somewhere inside **`/tmp/debian/`** to somewhere outside **`/tmp/debian`**. Can you follow the symbolic link when using **`/tmp/debian`** as a **`chroot`** jail?

c) Write a program that, when executed inside the **/tmp/debian chroot** jail with root rights, reads the file **/tmp/outside**, which is *outside* the **chroot** jail and outputs its contents. Submit the source code of the program.
   Note: The program will first have to escape the **chroot** jail using the above sequence.
d) Does the program also function without root permissions? Explain why or why not. **Note:** You can use **chroot --userspec USERNAME** to try this.

2 Return to libc

The standard mechanism to circumvent a non-executable stack in a buffer-overflow attack is to use return-oriented programming. This exercise is a classical return-to-libc attack on the AMD64 architecture.

Prerequisites

We will (at first) make our life easy and attack a textbook vulnerable program, which additionally prints the address of a buffer:

```
#include <stdio.h>

int main(void)
{
  char name[256];
  printf("%p\n",&name);
  puts("What's your name?");
  gets(name);
  printf("Hello, %s!\n", name);
  return 0;
}
```

Let us assume that this program is running with suid-root; the target of the attacker is to use a buffer overflow of the `name` buffer to obtain a root shell.
The idea of the attack is the following: make sure that the code eventually returns into the `system` function of libc with a pointer to the string "`/bin/sh`" in register `rdi`. This assumes that the attack is running on a 64-bit Linux system (AMD64 architecture); on this architecture, the first argument of a function is passed through register `rdi`. The attack needs the following building blocks:

- Put the string "`/bin/sh`" somewhere into the address space of the program, e.g., into the buffer `name`;
- find a gadget that consists of the instruction `pop %rdi` followed by `retq`;
- overwrite the return address of the function with the address of this gadget;
- write behind this gadget the address of the string "`/bin/sh`" (this is what is going to be popped into `rdi`);
- write behind the address of the string the address of `system` in libc. This is what is finally going to be called, giving you the root shell.

An excellent walkthrough of this attack is given by Ben Lynn on http://crypto.stanford.edu/~blynn/rop/.
Remark 1: Some parts of the assignment may depend on various aspects of your Linux system (in particular, the version of libc). If you have trouble with some parts, then try on `lilo.science.ru.nl` (where it has been tested). Obviously on this machine you cannot run the program suid-root, but you can still get a (non-root) shell and confirm that the attack works.
Remark 2: Note that in our exercise the size of the buffer changed; take this into account when mounting the attack.
Remark 3: It is important to compile the program with gcc flag `-fno-stack-protector`.
Remark 4: It is important to disable ASLR (either by using `setarch 'arch' -R` as in Lynn's tutorial for each call or by running `echo 0 > /proc/sys/kernel/randomize_va_space` once as root).

2.1 Objectives

a) Run the attack and obtain a root shell (you might want to try first without suid-root to not allow too much disaster if things go wrong). Now automate this attack in a bash script. The bash script should be robust, i.e., it should handle the case that offsets in libc are different or offsets between the `name` buffer and the return address are different than on your machine. You can test this by running the script on a different machine, e.g., on `lilo.science.ru.nl`. Submit this script.

b) The attack does not work against the "original" version of the program in Lynn's tutorial with a buffer size of 64. Use gdb to find out why not In particular, answer the following questions:
   - Does the attacked program jump (return) to the `pop %rdi, retq` gadget? If not, why not?
   - Does the attacked program put the right address into `rdi`? If not, why not?
   - Does the attacked program jump (return) to `system`? If not, why not?
   - Does the attacked program issue the correct syscall? If yes, which one? If not, why not?
   - Summarize and explain why the attack does not work.

   Note: It is of course perfectly fine to compile the program with the `-g` flag).
   Note: Addresses (for example, of `name`) are probably slightly different when running the program in gdb.

c) Can you think about a way to make the attack work with a buffer of size 64?
   Hint: Where else can you find the string `/bin/sh` or similar?

d) **Bonus task**: Make the attack work against a buffer of size 64 and against a buffer of size 4.