CS4375 Operating Systems Concepts
Fall 2024. Midterm I
10/14/2024

No documents allowed

You need to turn in the theoretical part of the exam (Sections 1 and 2) by 10/14/2024 11:50AM MDT on paper, directly to your instructor. You need to turn in the practical part of the exam (Section 3) by 10/14/2024 11:59PM MDT by email to `utep-os-fall-2024-midtermI@christoph-lauter.org`.

1 "Hello world" - the hardcore way (25 pts)

As we have seen in class, a system call to `write` can be obtained on a x86-64 Linux system with just a couple of lines of assembly:

```
        .text
        .globl  main
        .type   main, @function

main:
        movl    $1, %eax
        movl    $1, %edi
        movl    $12, %edx
        leaq    lbl(%rip), %rsi
        syscall
        movl    $0, %eax
        ret

lbl:
        .ascii  "Hello world\n"
```

- Explain where Linux knows from that this system call is a call to `write`.
- Explain where Linux knows from that the system call is to print out the message on the standard output file descriptor.
- Explain in detail which steps the Linux operating system goes through starting at the point where the `syscall` instruction is executed till the point where the system schedules the process again to execute the `movl` instruction.

2 Virtual Instructions (25 pts)

Older ARM processors had an instruction named `SWP` to atomically test and swap two values between memory and registers. That instruction was used in certain special cases for parallel computations with threads, but it was very costly to implement in hardware. Its inner workings do not matter for the purpose of this exam.

Due to the hardware cost it induced, ARM decided to deprecate the `SWP` instruction in later implementations of the ARM instruction set architecture. On recent ARM processors, the execution of the `SWP` provokes a trap into the Operating System. This means an interrupt is generated and the Operating System gets run at the appropriate interrupt vector address with an indication that an *undefined instruction* got executed. The Operating System of course knows the value of the program counter when such a trap occurs.

Already compiled, old legacy threaded user code using the `SWP` instruction, nevertheless continues to run on Linux systems running on these newer ARM processors that lack the instruction. To enable this kind of "magic", Linux implements an extended machine that emulates the instruction where needed.

Describe[1] which mechanism the Linux Operation System can use to perform such software emulation. Detail the different steps it takes from the moment it takes an *undefined instruction* trap until user code resumes after the emulated `SWP` instruction.

[1] You are of course not supposed to know the precise workings of Linux to perform this task. Just use your imagination.

3 Practical part: Cracking passwords (50pts)

On the `dandelion` server you have been given access to, reachable with `ssh` at `dandelion.cs.utep.edu`, you find an executable named `naivelogin` in the directory `/usr/bin`. That executable has been obtained compiling the following code with
`gcc -O3 -Wall -DLOGIN_SECRET='"tequila"' -o naivelogin naivelogin.c`,
of course with a secret password string stronger than `"tequila"`.
Here is the (flawed) `naivelogin` program:

```
#include <unistd.h>

#if !defined(LOGIN_SECRET)
#define LOGIN_SECRET "password"
#endif

static int read_character(char *c) {
  char buf;
  ssize_t res;

  res = read(0, &buf, (size_t) 1);

  if (res == ((ssize_t) 1)) {
    *c = buf;
    return 0;
  }
  return -1;
}

int main(int argc, char **argv) {
  char secret[] = LOGIN_SECRET;
  int i;
  char c;

  for (i=0; secret[i]!='\0'; i++) {
    if (read_character(&c) < 0) return 1;
    if (c != secret[i]) return 1;
  }
  return 0;
}
```

The `naivelogin` program checks if the user types in the correct password on standard input. It returns `0` in case of success and `1` in case the password is wrong. In `bash`, you can use `echo $?` to display the return value of a program, also called the status code. In C, the `wait()` and `waitpid()` system calls provide a means to recover the status (exit) code of a program.

Your task is to write a program that cracks the secret password compiled into `naivelogin`. That secret password is 1 to 16 characters in length and has been made out of the character set `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%&*_-+<>/.`

Base your code on the boilerplate code `crack.c` provided. Proceed as follows:

1. Analyze the `naivelogin.c` source code and try to understand its interactions with the Linux Operating System.

2. Analyze the function `try_password()` in the provided boilerplate code. Write a short `main()` that tries out some password guess, like `"mezcal"`.

3. Come up with an algorithm that leverages the flaw in `naivelogin` that you have discovered.

Do not try to brute force the problem! There are 75 possible choices for each of the 1 to 16 characters in the password. This means there are

$$\sum_{i=1}^{16} 75^i = 1015803624082960792489953943200$$

possible choices. If you can brute-force check passwords with a frequency of 1kHz, this means you will need $1015803624082960792489953943.200$ seconds, i.e. $3.22 \cdot 10^{22}$ years, to check all combinations!

The instructor's solution cracks the password on `dandelion` in 0.116 seconds.