Nebula Challenges

– TD (**2**) –

1 Getting started in Nebula

Nebula takes the participant through a variety of common (and less than common) weaknesses and vulnerabilities in Linux. It takes a look at: SUID files, permissions, race conditions, password cracking, shell meta-variables, `$PATH` weaknesses, scripting language weaknesses, binary compilation failures, . . .

At the end of Nebula, the user will have a reasonably thorough understanding of local attacks against Linux systems, and a cursory look at some of the remote attacks that are possible.

1.1 Setting and Starting the VM

Setting the VM is easy, just use the given "`*.iso`" image with QEMU with the following command line:

```
#> qemu-system-i386 -enable-kvm -cdrom ./nebula.iso
```

The previous commands will run an instance of the virtual image and allow you to log-in. But, if you want to access through an `ssh` connection on your machine you have to first run an instance of the machine and, then, connect on it:

```
#> qemu-system-i386 -net user,hostfwd=tcp::2222-:22 -net nic,model=e1000 \
                   -enable-kvm -cdrom ./nebula.iso
#> ssh -p 2222 levelXX@localhost
```

Note that you need to copy the `*.iso` file to `/tmp` in order to access locally to your VM and avoid going through the NFS. It will give you a quicker access to it and lower the number of access error during your work session.

1.2 Getting logged on the VM as Administrator

Login on the machine as Admin can be done through the `nebula` account (**login**: `nebula`, **password**: `nebula`). In case you need root access to change something on the system, you can use the "`sudo`" command from the `nebula` account (as in Ubuntu). Also, if you want to change the keyboard layout as a French keyboard, you should log as `nebula` and type the following command: “`sudo loadkeys fr`”.

1.3 Levels

There are twenty levels (from `00` to `19`), to complete each of these levels you have to log in as "`levelXX`" (login: `levelXX`, password: `levelXX` with "`XX`" the number of the level) and try to spawn a shell under the login "`flagXX`". Once you think you have completed the level, run “`getflag`” to get the confirmation of your success (or check your identity through the `id` command).

2 Challenges

Questions

0. **Level00**: This level requires you to find a Set User ID program named, “`flag00`”, that will run as the “`flag00`” account. You could also find this by carefully looking in top level directories in `/` for suspicious looking directories.

   To access this level, log in as `level00` with the password of `level00`.

1. **Level01**: There is a vulnerability in the below program that allows arbitrary programs to be executed, can you find it?

   To do this level, log in as the `level01` account with the password `level01`. Files for this level can be found in `/home/flag01`.

```c
#define _GNU_SOURCE
#include <stdlib.h>
#include <unistd.h>

int main() {
  gid_t gid = getegid();
  uid_t uid = geteuid();

  setresgid(gid, gid, gid);
  setresuid(uid, uid, uid);

  system("/usr/bin/env echo and now what?");

  return EXIT_SUCCESS;
}
```

2. **Level02**: There is a vulnerability in the below program that allows arbitrary programs to be executed, can you find it?

```c
#define _GNU_SOURCE
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main() {
  char *buffer = NULL;
  gid_t gid = getegid();
  uid_t uid = geteuid();

  setresgid(gid, gid, gid);
  setresuid(uid, uid, uid);

  asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
  printf("about to call system(\"%s\")\n", buffer);

  system(buffer);
  free(buffer);

  return EXIT_SUCCESS;
}
```

3. **Level03**: Check the home directory of `flag03` and take note of the files there.

   A crontab is called every two minutes to execute the script `/home/flag03/writable.sh`.

4. **Level04**: This level requires you to read the token file, but the code restricts the files that can be read. Find a way to bypass it.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <err.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv) {
  char buf[1024];
  int fd, rc;

  if (argc == 1) {
    printf("%s [file to read]\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  if (strstr(argv[1], "token") != NULL) {
    printf("You may not access '%s'\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  fd = open(argv[1], O_RDONLY);
  if (fd == -1) {
    err(EXIT_FAILURE, "Unable to open %s", argv[1]);
  }

  rc = read(fd, buf, sizeof(buf));
  if (rc == -1) {
    err(EXIT_FAILURE, "Unable to read fd %d", fd);
  }

  write(1, buf, rc);

  return EXIT_SUCCESS;
}
```

5. **Level05**: Check the `flag05` home directory. You are looking for weak directory permissions

6. **Level06**: The `flag06` account credentials came from a legacy UNIX system (see: `/etc/passwd`).

7. **~~Level07~~**: The `flag07` user was writing their very first perl program that allowed them to ping hosts to see if they were reachable from the web server.

```
#!/usr/bin/perl

use CGI qw{param};
print "Content-type: text/html\n\n";

sub ping {
  $host = $_[0];

  print("<html><head><title>Ping results</title></head><body><pre>");

  @output = `ping -c 3 $host 2>&1`;
  foreach $line (@output) { print "$line"; }

  print("</pre></body></html>");
}

# check if Host set. if not, display normal page, etc
ping(param("Host"));
```

8. **Level08**: World readable files strike again. Check what that user was up to, and use it to log into `flag08` account.

9. **~~Level09~~**: There's a C setuid wrapper for some vulnerable PHP code. . .

```
<?php
  function spam($email) {
    $email = preg_replace("/\./", " dot ", $email);
    $email = preg_replace("/@/", " AT ", $email);

    return $email;
  }

  function markup($filename, $use_me) {
    $contents = file_get_contents($filename);

    $contents = preg_replace("/(\[email (.*)\])/e", "spam(\"\\2\")", $contents);
    $contents = preg_replace("/\[/", "<", $contents);
    $contents = preg_replace("/\]/", ">", $contents);

    return $contents;
  }

$output = markup($argv[1], $argv[2]);
print $output;
?>
```

10. **Level10**: The setuid binary at `/home/flag10/flag10` binary will upload any file given, as long as it meets the requirements of the `access()` system call.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#include <errno.h>
#include <fcntl.h>
#include <string.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  char *file, *host;

  if (argc < 3) {
    printf("%s file host\n"
           "\tsends file to host if you have access to it\n",
           argv[0]);
    exit(EXIT_FAILURE);
  }

  file = argv[1];
  host = argv[2];

  if (access(argv[1], R_OK) == 0) {
    int fd;
    int ffd;
    int rc;
    struct sockaddr_in sin;
    char buffer[4096];

    printf("Connecting to %s:18211 .. ", host); fflush(stdout);
    fd = socket(AF_INET, SOCK_STREAM, 0);

    memset(&sin, 0, sizeof(struct sockaddr_in));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = inet_addr(host);
    sin.sin_port = htons(18211);

    if (connect(fd, (void *)&sin, sizeof(struct sockaddr_in)) == -1) {
      printf("Unable to connect to host %s\n", host);
      exit(EXIT_FAILURE);
    }

#define HITHERE ".oO Oo.\n"
  if (write(fd, HITHERE, strlen(HITHERE)) == -1) {
    printf("Unable to write banner to host %s\n", host);
    exit(EXIT_FAILURE);
  }
#undef HITHERE

  printf("Connected!\nSending file .. "); fflush(stdout);
  ffd = open(file, O_RDONLY);
  if (ffd == -1) {
    printf("Damn. Unable to open file\n");
    exit(EXIT_FAILURE);
  }

  rc = read(ffd, buffer, sizeof(buffer));
  if (rc == -1) {
    printf("Unable to read from file: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  write(fd, buffer, rc);
  printf("wrote file!\n");
  } else {
    printf("You don't have access to %s\n", file);
  }
}
```

11. **Level11**: The `/home/flag11/flag11` binary processes `stdin` and executes a shell command.

```
#define _GNU_SOURCE
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/mman.h>
#include <sys/types.h>

#include <err.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>

#define CL "Content-Length: "

/* Return a random, non predictable file, and return the file
 * descriptor for it. */
int getrand(char **path) {
  srandom(time(NULL));
  asprintf(path, "%s/%d.%c%c%c%c%c%c", getenv("TEMP"), getpid(),
           'A' + (char) (random() % 26), '0' + (char) (random() % 10),
           'a' + (char) (random() % 26), 'A' + (char) (random() % 26),
           '0' + (char) (random() % 10), 'a' + (char) (random() % 26));

  int fd = open(*path, O_CREAT|O_RDWR, 0600);
  unlink(*path);
  return fd;
}

void process(char *buffer, int length) {
  unsigned int key = length & 0xff;

  for (int i = 0; i < length; ++i) {
    buffer[i] ^= key;
    key -= buffer[i];
  }
  system(buffer);
}

int main() {
  char line[256], buf[1024];
  char *mem, *path;

  if (fgets(line, sizeof(line), stdin) == NULL)
    errx(1, "reading from stdin");

  if (strncmp(line, CL, strlen(CL)) != 0)
    errx(1, "invalid header");
  unsigned int length = atoi(line + strlen(CL));

  if (length < sizeof(buf)) {
    if (fread(buf, length, 1, stdin) != length)
      err(1, "fread length");
    process(buf, length);
  } else {
    int blue = length;
    int fd = getrand(&path);
    while (blue > 0) {
      printf("blue = %d, length = %d, ", blue, length);

      int pink = fread(buf, 1, sizeof(buf), stdin);
      printf("pink = %d\n", pink);

      if (pink <= 0)
        err(1, "fread fail(blue = %d, length = %d)", blue, length);
      write(fd, buf, pink);
      blue -= pink;
    }

    mem = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
    if (mem == MAP_FAILED)
      err(1, "mmap");

    process(mem, length);
  }
  return EXIT_SUCCESS;
}
```

12. ~~**Level12**~~: There is a backdoor process listening on port 50001.

```
local socket = require("socket")
local server = assert(socket.bind("127.0.0.1", 50001))

function hash(password)
  prog = io.popen("echo "..password.." | sha1sum", "r")
  data = prog:read("*all")
  prog:close()

  data = string.sub(data, 1, 40)
  return data
end

while 1 do
  local client = server:accept()
  client:send("Password: ")
  client:settimeout(60)
  local line, err = client:receive()
  if not err then
    print("trying " .. line) -- log from where ;\
    local h = hash(line)
    if h ~= "4754a4f4bd5787accd33de887b9250a0691dd198" then
      client:send("Better luck next time\n");
    else
      client:send("Congrats, your token is 413**CARRIER LOST**\n")
    end
  end

  client:close()
end
```

13. **Level13**: There is a security check that prevents the program from continuing execution if the user invoking it does not match a specific user id.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>

#define FAKEUID 1000

int main(int argc, char **argv, char **envp)
{
  int c;
  char token[256];

  if (getuid() != FAKEUID) {
    printf("Security failure detected. UID %d started us, we expect %%d\n",
           getuid(), FAKEUID);
    printf("The system administrators will be notified of this violation\n");
    exit(EXIT_FAILURE);
  }

  // snip, sorry

  printf("your token is %s\n", token);
}
```

14. **Level14**: This program resides in `/home/flag14/flag14`. It encrypts input and writes it to standard output. An encrypted token file is also in that home directory, decrypt it.

15. **Level15**: `strace` the `/home/flag15/flag15` binary and see if you spot anything out of the ordinary. You may wish to review how to "compile a shared library in Linux" and how the libraries are loaded and processed by reviewing the `dlopen` manpage in depth.

16. **~~Level16~~**: There is a perl script running on port 1616.

```
!/usr/bin/env perl

use CGI qw{param};
print "Content-type: text/html\n\n";

sub login {
  $username = $_[0];
  $password = $_[1];

  $username =~ tr/a-z/A-Z/; # convert to uppercase
  $username =~ s/\s.*//;    # strip everything after a space

  @output = `egrep "^$username" /home/flag16/userdb.txt 2>&1`;
  foreach $line (@output) {
    ($usr, $pw) = split(/:/, $line);

    if ($pw =~ $password) {
      return 1;
    }
  }
  return 0;
}
```

```
sub htmlz {
  print("<html><head><title>Login resuls</title></head><body>");
  if ($_[0] == 1) {
    print("Your login was accepted<br/>");
  } else {
    print("Your login failed<br/>");
  }
  print("Would you like a cookie?<br/><br/></body></html>\n");
}

htmlz(login(param("username"), param("password")));
```

17. ~~**Level17**~~: There is a python script listening on port 10007 that contains a vulnerability.

```
#!/usr/bin/python
import os
import pickle
import socket
import signal
import time

signal.signal(signal.SIGCHLD, signal.SIG_IGN)

def server(skt):
  line = skt.recv(1024)
  obj = pickle.loads(line)

  for i in obj:
    clnt.send("why did you send me " + i + "?\n")

skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
skt.bind(('0.0.0.0', 10007))
skt.listen(10)

while True:
  clnt, addr = skt.accept()

  if (os.fork() == 0):
    clnt.send("Accepted connection from %s:%d" % (addr[0], addr[1]))
    server(clnt)
    exit(1)
```

18. **Level18**: Analyse the C program, and look for vulnerabilities in the program. There is an easy way to solve this level, an intermediate way to solve it, and a more difficult/unreliable way to solve it.

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <err.h>
#include <getopt.h>
#include <string.h>
```

```c
struct {
  FILE *debugfile;
  int verbose;
  int loggedin;
} globals;

#define dprintf(...) \
  if (globals.debugfile) \
    fprintf(globals.debugfile, __VA_ARGS__)
#define dvprintf(num, ...) \
  if (globals.debugfile && globals.verbose >= num) \
    fprintf(globals.debugfile, __VA_ARGS__)

#define PWFILE "/home/flag18/password"

void login(char *pw) {
  FILE *fp;

  fp = fopen(PWFILE, "r");
  if (fp) {
    char file[64];

    if (fgets(file, sizeof(file) - 1, fp) == NULL) {
      dprintf("Unable to read password file %s\n", PWFILE);
      return;
    }
    fclose(fp);
    if (strcmp(pw, file) != 0) return;
  }
  dprintf("logged in successfully (with%s password file)\n",
          fp == NULL ? "out" : "");
  globals.loggedin = 1;
}

void notsupported(char *what) {
  char *buffer = NULL;
  asprintf(&buffer, "--> [%s] is unsupported at this current time.\n", what);
  dprintf(what);
  free(buffer);
}

void setuser(char *user) {
  char msg[128];
  sprintf(msg, "unable to set user to '%s' -- not supported.\n", user);
  printf("%s\n", msg);
}

int main(int argc, char **argv, char **envp) {
  char c;
  while ((c = getopt(argc, argv, "d:v")) != -1) {
    switch(c) {
      case 'd':
      globals.debugfile = fopen(optarg, "w+");
      if (globals.debugfile == NULL) err(1, "Unable to open %s", optarg);
      setvbuf(globals.debugfile, NULL, _IONBF, 0);
      break;
      case 'v':
```

```
      globals.verbose++;
      break;
    }
  }

  dprintf("Starting up. Verbose level = %d\n", globals.verbose);
  setresgid(getegid(), getegid(), getegid());
  setresuid(geteuid(), geteuid(), geteuid());

  while (1) {
    char line[256];
    char *p;

    char *q = fgets(line, sizeof(line)-1, stdin);
    if (q == NULL) break;
    p = strchr(line, '\n'); if (p) *p = 0;
    p = strchr(line, '\r'); if (p) *p = 0;

    dvprintf(2, "got [%s] as input\n", line);

    if (strncmp(line, "login", 5) == 0) {
      dvprintf(3, "attempting to login\n");
      login(line + 6);
    } else if (strncmp(line, "logout", 6) == 0) {
      globals.loggedin = 0;
    } else if (strncmp(line, "shell", 5) == 0) {
      dvprintf(3, "attempting to start shell\n");
      if (globals.loggedin) {
        execve("/bin/sh", argv, envp);
        err(1, "unable to execve");
      }
      dprintf("Permission denied\n");
    } else if (strncmp(line, "logout", 4) == 0) {
      globals.loggedin = 0;
    } else if (strncmp(line, "closelog", 8) == 0) {
      if (globals.debugfile) fclose(globals.debugfile);
      globals.debugfile = NULL;
    } else if (strncmp(line, "site exec", 9) == 0) {
      notsupported(line + 10);
    } else if (strncmp(line, "setuser", 7) == 0) {
      setuser(line + 8);
    }
  }
  return EXIT_SUCCESS;
}
```

19. **Level19**: There is a flaw in the below program in how it operates.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <err.h>
#include <string.h>
#include <sys/stat.h>

int main(int argc, char **argv, char **envp) {
  char buf[256];
  struct stat statbuf;

  /* Get the parent's /proc entry, so we can verify its user id */
  snprintf(buf, sizeof(buf) - 1, "/proc/%d", getppid());

  /* stat() it */
  if (stat(buf, &statbuf) == -1) {
    printf("Unable to check parent process\n");
    exit(EXIT_FAILURE);
  }

  /* check the owner id */
  if (statbuf.st_uid == 0) {
    /* If root started us, it is ok to start the shell */
    if (argc > 1)
      execve("/bin/sh", argv, envp);
    err(1, "Unable to execve");
  }
  printf("You are unauthorized to run this program\n");

  return EXIT_SUCCESS;
}
```