Ordinary Wizarding Level Examination
Defense against the Dark Arts
Fall, 2015

As a young wizard, you are required to take Ordinary Wizarding Level Examination (O.W.L.) to earn your wizard title and hone your skills for the battle you are sure to face. In the O.W.L, you must overcome four challenges. If you complete all four challenges you will be awarded a grade of 100 (out of 100) for the final examination grade. Glory is within your grasp! Complete details of the grading of the O.W.L. are provided at the end of this document. There is an extra challenge which is worth 10 extra points. This extra challenge must be finished if you want to compete for Wizard's Cups.

Due Date

O.W.L is due on Tuesday, 12/15/2015, at 10:00am

O.W.L Rules

The five challenge problems **must be** completed in the order presented. Thus, you must complete Challenge 1 before moving on to Challenge 2. You must complete Challenge 3 before moving on to Challenge 4, etc. You may not work together. The O.W.L. is a pledged activity. You may use any non-human resource. For example, you may use any resource available on the Internet. (Remember: Google is a Wizard's friend.)

You should make sure that ASLR (Address Space Layout Randomization) is turned off. The command

```
setarch i386 -RL bash
```

will disable ASLR for the current shell. It does not affect any other shells.

NOTE: To keep the necessary files segregated according to the Challenge, the files are under Resources in the Collab.

IMPORTANT: Use the executables provided. **DO NOT** compile the sources provided to produce new executables and then attack those. The Ministry of Magic will be using the executables they provided to test your spells.

Challenge 1: Arc injection via buffer overrun (easy)

This first challenge is very similar to Assignment 4. The executable to attack is *challenge1.bin*. The source code is *challenge1.c*. Again to emphasize, please used the executable provided by the Ministry of Magic as that executable is what the one the Ministry will use to test your attack code generator. Do not recompile the source code to produce a new executable. As should be obvious, the code contains a buffer overrun vulnerability in the *readString* function. You are to exploit this vulnerability by performing an arc-injection spell to change your grade from a "D" to an "A".

As before, you should write a program to generate the attack string. Save the attack string into a file named *attack_spell1.txt*. Here are two sample runs that demonstrate the execution of the program. Assume the file *name.txt* only has the name Gregory Goyle.

```
$ .challenge1.bin <name.txt
Gregory Goyle: You have not completed this challenge. Try again.
$
```

Now we generate an attack string ans save it into *attack_spell1.txt* and running the program.

```
$ ./challenge1.bin <attack_spell1.txt
Hermione Granger: Congratulations, you have completed this
challenge!.
$
```

If you complete this challenge, submit your attack string. It must be named *attack_spell1.txt*.

Challenge 2: Arc injection via buffer overrun (medium)

This second challenge is very similar to Challenge 1 (but it has a slight twist that you must understand and handle). Examine *challenge2.c* and *challenge2.bin*. Again, as should be obvious, the code contains a buffer overrun vulnerability in the *readString* function. You are to exploit this vulnerability by performing an arc-injection spell to change your grade from a "D" to an "A".

As before, you should write a program to generate the attack string. Save the attack string into a file named *attack_spell2.txt*. Here are two sample runs that demonstrate the execution of the program. Assume the file *name.txt* has the name Millicent Bulstrode.

```
$ ./challenge2.bin <name.txt
Thank you, Millicent Bulstrode.
I recommend that you get a grade of D on this assignment.
$
```

Now we generate an attack string and save it into *attack_spell2.txt* and running the program.

```
$ ./challenge2.bin <attack_spell2.txt
Thank you, Hermione Granger.
I recommend that you get a grade of A on this assignment.
$
```

If you complete this challenge, submit your attack string. It must be named *attack_spell2.txt*.

Challenge 3: Code injection via buffer overrun

For this challenge, you will use the code *challenge3.c* and *challenge3.bin*. Again, your goal is to give yourself a grade of "A". Unfortunately, an arc-injection attack is not effective against this slightly modified version of the program. You must do a code-injection attack (i.e., inject code on the run-time stack and cause it to be executed). You must inject code that will change your grade and then cause that code to be executed.

The Ministry of Magic has prepared an executable that will permit code injection attacks. As before, you should write a program to generate the attack string. Save the attack string into a file named

attack_spell3.txt. Here are two sample runs that demonstrate the sample runs of the program. Assume the file *name.txt* has the name Vincent Crabbe.

```
$ ./challenge3.bin <name.txt
Thank you, Vincent Crabbe.
Unfortunately, you have not completed this challenge. Try again.
Exiting
$
```

Now we generate an attack string and saved it into *attack_spell3.txt* and running the program.

```
$ ./challenge3.bin <attack_spell3.txt
Thank you, Ginny Weasley.
Congratulations, you have completed this challenge.
Exiting
$
```

If you complete this final challenge, submit your attack string. It must be named *attack_spell3.txt*.

Challenge 4: Format String attack

For this challenge, you will use the code *challenge4.c* and *challenge4.bin*. Again, your goal is to give yourself a grade of "A". For this challenge, the weakness is a format string vulnerability (see function vulnerable()).

As before, you should write a program to generate the attack string. Save the attack string into a file named *attack_spell4.txt*. Here are two sample runs that demonstrate the sample runs of the program. Assume the file *name.txt* contains two lines where each line has the name Draco Malfoy.

```
$ ./challenge4.bin <name.txt
Draco Malfoy
Thank you, Draco Malfoy.
I recommend that you get a grade of D on this assignment.
$
```

Now we generate an attack string and save it into *attack_spell4.txt* and running the program.

```
$ ./challenge4.bin <attack_spell4.txt
(?0 Tp ` ? ? ? ? @ ? H ?d 1073859552 1073856500 1073945512
Thank you, Luna Lovegood.
I recommend that you get a grade of A on this assignment.
$
```

Notice the garbled output before the Thank you, Luna Lovegood message. This output is being generated by the format string that is being passed to function vulnerable(). This type of output is a downside of a format string attack. If you complete this third challenge, submit your attack string. It must be named *attack_spell4.txt*.

Ordinary Wizarding Level Examination

Bonus Challenge 5: ROP attack via buffer overrun.

A ROP-attack, or return-oriented programming attack, is a type of arc-injection attack where the attacker causes "gadgets" already existing in the code to be executed. If sufficient gadgets exist, the attacker can cause arbitrary programs to be executed. The goal of our attack is to exec a shell and execute a shell command (*hacked.sh*). You first need to download and install a tool, called *ROPgadget*. I have provided a tarball in the Collab resources directory for this Challenge. Grab the tarball and extract the files in the directory where you will be doing your work. Move into the *ROPgadget* directory and issue the following commands:

```
$ make clean
$ make
```

Make sure the program is built by issuing the following command:

```
$ ./ROPgadget –help
```

You will need to read the document to understand how to use *ROPgadget*. There are some YouTube videos demonstrating its use.

You will need to download the following files from the Collab: *challenge5.c*, *challenge5.bin*, *ROPgadget.tgz*, and *hacked.sh*. You will notice that *challenge5.c* is identical to the one used in Challenge 1 (with onevery slight differences). So you should know how to craft an attack string that will kick off the execution of a sequence of gadgets. The file *hacked.sh* needs to be in the directory where you carry out the attack (i.e., same directory as the binary you are attacking).
You first need to build *ROPgadget*. Untar the package in your directory with the challenge files.

```
$ tar –zxvf ROPgadget.tgz
<list of files extracted>
$ cd ROPgadget
$ make clean
$ make
```

The tool *ROPgadget* will generate a program that will create most of the attack string. Here is the command:

```
$ ./ROPgadget/ROPgadget –csyn  challenge5.bin /bin/sh hacked.sh >attack_spell5.c
```

You will need to add some code to the program that ROPgadget generates (in this case attack_spell5.c) where it says "Padding goes here". Basically, this is the beginning of the attack string – your name and some padding to fill up the buffer so that the first address that ROPgadget generated overwrites the return address in function readString's activation. We are essentially doing an arc-injection attack to the first gadget and then ROPgadget has set up successive addresses to invoke the appropriate gadgets. Call this modified program (where you have inserted the necessary padding), attack_spell5.c. The following illustrates an attack run assuming the attack string generated by ROPgadget was written to a file called attack_spell5.c and it was compiled to produce attack_spell2.bin.

```
$ ./ROPgadget/ROPgadget -csyn challenge5.bin /bin/sh hacked.sh >attack_spell5.c
<edit attack_spell5.c to put in padding>
$ gcc -m32 -o attack_spell5.bin attack_spell5.c
$ ./attack_spell5.bin >attack_spell5.txt
$ ./challenge5.exe <attack_spell5.txt
You passed your OWLs. You are a true Wizard! Congratulations!
$
```

Notice that because the attack code causes a shell to be exec'ed (causing the shell script contained in *hacked.sh* to be executed), the normal output of the program does not appear. If you complete this extra challenge, submit your attack string. It must be named *attack_spell5.txt*.

Wizards of Distinction

The first two wizards to finish all five challenges will be deemed the champions to signify their stellar achievement. At the Ministry of Magic's direction, the champions will receive Wizard's Cups and at least an "A" for the final course grade.

Grading

For grading purposes, completion of challenges will be scored as indicated in the following table.

Challenges Completed	Final Examination Grade
1: Arc-injection via buffer overran (easy)	70
2: Arc-injection via buffer overran (medium)	80
3: Code-injection via buffer overran	90
4: Format string attack	100
5: ROP attack via buffer overran	110

Miscellaneous Notes

1. The O.W.L ends on Tuesday (12/15/2015) at 10:00 A.M. **No exceptions and no extensions.**

2. You must use the binaries provided on the Collab. The source code is provided to help you craft your attacks. All testing will be done on the binaries provided. You must follow the

3. If you are competing for the Wizard's Cup, send me an e-mail saying that you have completed all five challenges. The Collab time stamps will be used to determine the finish times.

4. You must make your submissions via the Collab. To recap,

Challenges Completed	File to submit
1: Arc-injection via buffer overran (easy)	attack_spell1.txt
2: Arc-injection via buffer overran (medium)	attack_spell2.txt

3: Code-injection via buffer overran	attack_spell3.txt
4: Format string attack	attack_spell4.txt
5: ROP attack via buffer overran	attack_spell5.txt

5. There will be a 40% penalty on the total grade if any submitted file's name is incorrect. There will be a 40% penalty on the total grade if you compile your own executable. There will be a 40% penalty on the total grade if you failed to follow the instructions of this document. No exceptions will be given.

6. Remember: Google is your friend. There are lots of resources out there to help you.