

Smashing the Stack for Fun and Credit¹

1 Overview

In this assignment you will learn about security vulnerabilities in software. In particular, you will find and exploit memory corruption vulnerabilities in order to better understand how simple programming errors can lead to whole system compromise. You will also examine some common mitigation techniques and analyze exploit code (i.e., shellcode). This assignment is due by no later than 11:59pm on Tuesday, September 21, 2021. See Section 6 for submission details.

The code and other answers you submit must be entirely your own work. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone else. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Objectives

- Be able to identify and avoid buffer overflow vulnerabilities in native code
- Understand the severity of buffer overflows and the necessity of standard defenses
- Gain familiarity with machine architecture and assembly language

Read this First

This project asks you to develop attacks and test them in a virtual machine you control. Attempting the same kinds of attacks against others' systems without authorization is prohibited by law and university policies and may result in *fines, expulsion, and jail time*. **You must not attack anyone else's system without authorization!** Per the course ethics policy, you are required to respect the privacy and property rights of others at all times, *or else you will fail the course*.

2 Getting started

Buffer-overflow exploitation depends on specific details of the target system, so we are providing a Kali Linux VM in which you should develop and test your attacks. This will also promote a consistent experience for everyone in the class and facilitate the grading process. We've slightly tweaked the configuration to disable security features that are commonly used in the wild but would complicate your work. We'll use this precise configuration to grade your submissions, so do not use your own VM instead. You must download the targets into a folder owned by the VM, running targets in a shared folder will cause incorrect behavior.

To get started, first download and install a copy of *Oracle VirtualBox* for Windows, GNU/Linux, or Mac OS X.² Next, download the GNU/Linux virtual machine image that we will use for this project and the Project 1

¹Thanks to Sam Small for creating parts of this assignment. Thanks as well to the University of Michigan for letting me borrow other parts of this project.

²The software and license information is available via the Oracle VirtualBox website.

materials³ from <https://www.cs.purdue.edu/homes/clg/CS526/>. Project 1 materials are also available via Brightspace.

The login information for the two available users is:

Username: cs526

Password: cs526

Username: root

Password: root

1. Download VirtualBox from <https://www.virtualbox.org/> and install it on your computer. VirtualBox runs on Windows, Linux, and Mac OS.
2. Get the VM file at <https://www.cs.purdue.edu/homes/clg/CS526/projects/526-kali.ova>. This file is 2 GB, so we recommend downloading it from campus.
3. Launch VirtualBox and select File > Import Appliance to add the VM.
4. Start the VM. The username and password required to login are cs526 and cs526.
5. Download <https://www.cs.purdue.edu/homes/clg/CS526/projects/CS526.Fall2021.Project1.tgz> from inside the VM. This file contains the target programs you will exploit.
6. `tar -xzvf CS526.Fall2021.Project1.tgz`
7. `cd targets/`
8. Each of your targets will be slightly different. Personalize the targets by running:
`./setcookie username`
Make sure your username is your Purdue Career Username (and correct)!
9. `sudo make` (The password you're prompted for is cs526. *You must run this command as sudo*)

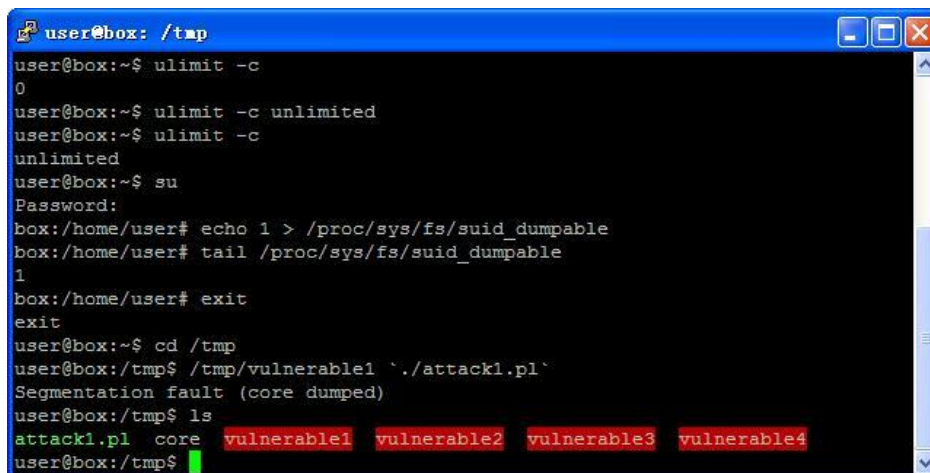
3 Resources, Guidelines, and Helpful Information

No Attack Tools! You may not use special-purpose tools meant for testing security or exploiting vulnerabilities. You must complete the project using only general purpose tools, such as gdb.

Control Hijacking Before you begin this project, review the lecture slides from software security lectures. Read “Smashing the Stack for Fun and Profit,” available at <http://phrack.org/issues/49/14.html>.

GDB You will make use of the GDB debugger for dynamic analysis within the VM. Useful commands that you may not know are “disassemble”, “info reg”, “x”, and “stepi”. See the GDB help for details, and don’t be afraid to experiment! This quick reference may also be useful: <https://web.eecs.umich.edu/~sugih/pointers/Gdb-reference-card.pdf>.

³I recommend using `wget` or `scp` to transfer the Project 1 materials to your virtual machine. To use `wget` for example, use the command `wget https://www.cs.purdue.edu/homes/clg/CS526/projects/CS526.Fall2021.Project1.tgz` from the shell in your virtual machine.



```
user@box: /tmp
user@box:~$ ulimit -c
0
user@box:~$ ulimit -c unlimited
user@box:~$ ulimit -c
unlimited
user@box:~$ su
Password:
box:/home/user# echo 1 > /proc/sys/fs/suid_dumpable
box:/home/user# tail /proc/sys/fs/suid_dumpable
1
box:/home/user# exit
exit
user@box:~$ cd /tmp
user@box:/tmp$ /tmp/vulnerable1 `./attack1.pl`
Segmentation fault (core dumped)
user@box:/tmp$ ls
attack1.pl  core  vulnerable1  vulnerable2  vulnerable3  vulnerable4
user@box:/tmp$
```

Figure 1: If you use ssh to connect to your virtual machine and want to enable core dump files, you must execute `ulimit -c unlimited` and for setuid executables execute `echo 1 > /proc/sys/fs/suid_dumpable` as root.

x86 Assembly There are many good references for Intel assembly language, but note that this project targets the 32-bit x86 ISA. The stack is organized differently in x86 and x86_64. If you are reading any online documentation, ensure that it is based on the x86 architecture, not x86_64.

3.1 Enabling core dump files

Core dump files can be useful for analyzing the cause of an execution crash. Once enabled, the operating system will create a file named `core` in your current working directory when a process crashes (e.g., due to a segmentation fault). Used in conjunction with `gdb`, the core file can give you a complete view of the process state when it crashed. To analyze a crashed process, simply launch `gdb` with the names of the executable file and its core file as arguments.

You can enable core dump files for processes that crash by using the command `ulimit -c unlimited` from the command prompt. However, to enable core dump for setuid executables, you should use an additional command (`echo 1 > /proc/sys/fs/suid_dumpable`). If you need assistance enabling core files, see Figure 1 or ask for help via the course discussion board.

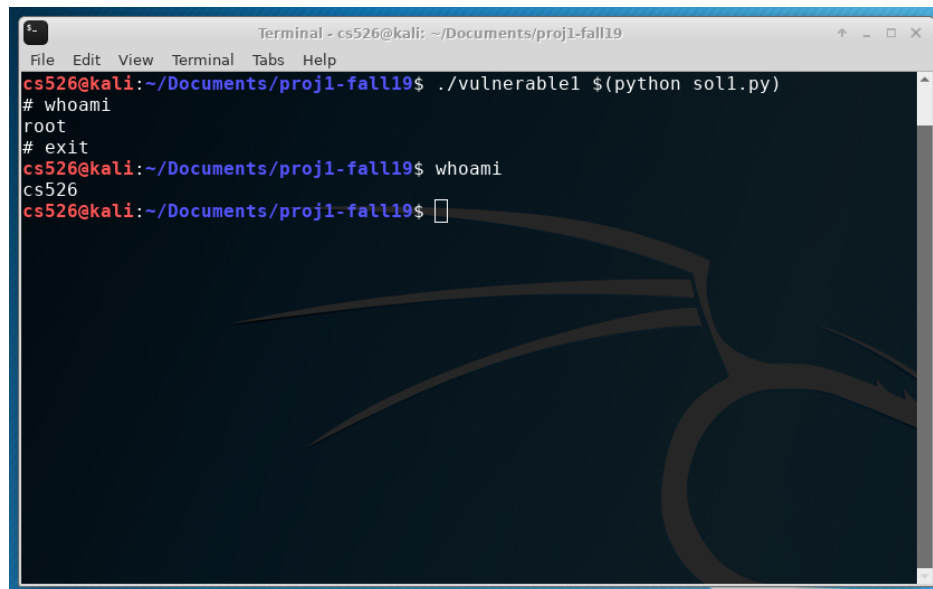
3.2 Miscellaneous

You may find it helpful to write an additional program or script that will assist in figuring out the correct input to exploit any of the vulnerabilities. Note that this can be separate from the script/program you will write to exploit the vulnerabilities. If you're unsure about what is acceptable for this assignment, please don't hesitate to email us. Be sure to turn in the code for any additional program or script that you write.

4 Targets

The target programs for this project are simple, short C programs with (mostly) clear security vulnerabilities. We have provided source code and a Makefile that compiles all the targets. Your exploits must work against the targets as compiled and executed within the provided VM.

These targets are owned by the `root` user and have the `suid` bit set. Your goal is to cause them to launch a shell, which will therefore have root privileges. This and the following targets all take input as command-line arguments rather than from `stdin`. Unless otherwise noted, you should use the shellcode we have provided in `shellcode.py`⁴ for you to use. Successfully placing this shellcode in memory and setting the instruction pointer to the beginning of the shellcode (e.g., by returning or jumping to it) will open a shell. A successful run is shown below.

A terminal window titled "Terminal - cs526@kali: ~/Documents/proj1-fall19" shows the execution of a program. The user runs `./vulnerable1 $(python sol1.py)`. The program outputs `# whoami` and `root`, indicating a successful privilege escalation. The user then runs `# exit` and `whoami`, which outputs `cs526`, indicating the program has exited and the user is back at the prompt.

For each program that we have provided, we ask that you explicitly:

1. Briefly describe the behavior of the program.
2. Identify and describe the vulnerability as well as its implications.
3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
4. Provide your attack as a self-contained program written in Python.
5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

For `vulnerable5.c`, provide answers to the following questions in addition to those listed above.

1. What is the value of the stack canary? How did you determine this value?

⁴For reference, this shellcode is based off of Aleph One's shellcode. See Aleph One's seminal article *Smashing the Stack For Fun and Profit*, which is posted on the course webpage and all over the Internet.

2. Does the value change between executions? Does the value change after rebooting your virtual machine?
3. How does the stack canary contribute to the security of `vulnerable4`?

vulnerable1: Redirecting control to shellcode

What to submit Create a Python program named `sol1.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./vulnerable1 $(python sol1.py)
```

If you are successful, you will see a root shell prompt (`#`). Running `whoami` will output “root”.

If your program segfaults, you can examine the state at the time of the crash using GDB with the core dump: `gdb ./vulnerable1 core`. The file `core` won’t be created if a file with the same name already exists. Also, since the target runs as root, you will need to run it using `sudo ./vulnerable1` in order for the core dump to be created.

vulnerable2: Overwriting the return address indirectly

What to submit Create a Python program named `sol2.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./vulnerable2 $(python sol2.py)
```

vulnerable3: Beyond strings

This target takes as its command-line argument the name of a data file it will read. The file format is a 32-bit count followed by that many 32-bit integers. Create a data file that causes the provided shellcode to execute and opens a root shell.

What to submit Create a Python program named `sol3.py` that outputs the contents of a data file to be read by the target. Test your program with the command line:

```
python sol3.py > tmp; ./vulnerable3 tmp
```

vulnerable4: Variable stack position

When we constructed the previous targets, we ensured that the stack would be in the same position every time the vulnerable function was called, but this is often not the case in real targets. In fact, a defense called ASLR (address-space layout randomization) makes buffer overflows harder to exploit by changing the starting location of the stack and other memory areas on each execution. This target resembles `vulnerable1`, but the stack position is randomly offset by 0–255 bytes each time it runs. You need to construct an input that always opens a root shell despite this randomization.

What to submit Create a Python program named `sol4.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./vulnerable4 $(python sol4.py)
```

A word of caution If you see any output from the program before a root shell is opened, you have not done vulnerable4 of the project correctly and your solution will not be accepted by the grader.

vulnerable5: Bypassing DEP

This program has been compiled with data execution prevention (DEP) enabled.

What to submit This program is a little bit different. Create two text files for us:

- `sol5_input.txt` The input file you used in targeting vulnerable5
- `sol5_commands.txt` A text file containing the user commands used for targeting vulnerable5

Test your files with the command line:

```
./vulnerable5 sol5_input.txt
```

Warning: do not try to create a solution that depends on you manually setting environment variables. You cannot assume that the grader will run your solution with the same environment variables that you have set. Please make sure that your program exits gracefully when you close the spawned shell as well (it should not segfault).

Note that in this vulnerable program *you will not achieve a root shell with just a basic attack*. This is okay! This is due to protections in newer versions of `bash/dash` that are specifically intended to drop `setuid` privileges when a shell is spawned through `/bin/sh`.

If your program spawns a shell in the default environment and a root shell in `zsh`, then you have successfully completed the exploit for this class.

If you would like to know more about this, please read on...

From the documentation for the `system` call⁵:

”`system()` will not, in fact, work properly from programs with `set-user-ID` or `set-group-ID` privileges on systems on which `/bin/sh` is `bash` version 2, since `bash` 2 drops privileges on startup.”

`bash` is explicitly programmed to check for `setuid`⁶:

```
if (running_setuid && privileged_mode == 0)
    disable_priv_mode ();
```

Basically, if `dash/bash` detects that it is executed in a `setuid` process, it immediately changes the effective user ID to the process’s real user ID, essentially dropping the privilege. This was not always true in previous versions.

⁵<https://linux.die.net/man/3/system>

⁶<http://git.savannah.gnu.org/cgit/bash.git/tree/shell.c?id=a0c0a00fc419b7bc08202a79134fcd5bc042707>
n503

To see that this is actually just an artifact of the type of shell we are running, the VM also has `zsh` installed, which does not include these protections. We can switch the shell to `zsh` and run the exact same exploit with a very different result. See the example or the commands below for how to do this.

```
cs526@kali:~/Documents/proj1-fall19$ sudo rm /bin/sh
cs526@kali:~/Documents/proj1-fall19$ sudo ln -s /bin/zsh /bin/sh
cs526@kali:~/Documents/proj1-fall19$ zsh
kali% exit
cs526@kali:~/Documents/proj1-fall19$ sudo rm /bin/sh
cs526@kali:~/Documents/proj1-fall19$ sudo ln -s /bin/dash /bin/sh
cs526@kali:~/Documents/proj1-fall19$
```

To switch to `zsh`:

```
sudo rm /bin/sh
sudo ln -s /bin/zsh /bin/sh
```

Open a `zsh` shell by running:

```
zsh
```

Once you have opened the new shell, try running your exploit again. This should give you a root shell.

If we want to reset back to the default shell:

```
sudo rm /bin/sh
sudo ln -s /bin/dash /bin/sh
```

There are potential workarounds for this dropping of privileges in the default shell⁷... However, if your program spawns a shell in the default environment and a root shell in `zsh`, then you have successfully completed the exploit for this project.

Extra credit: For extra credit, create a Python program that will spawn a root shell using the default environment and shell. If you complete the extra credit, please let us know in the `answers.pdf` file and submit this solution instead.

vulnerable6: Return-oriented programming [Extra credit]

(Difficulty: Hard)

This target is identical to `vulnerable1`, but it is compiled with DEP enabled. Implement a ROP-based attack to bypass DEP and open a root shell. It may be helpful to use a tool such as ROPgadget (<https://github.com/JonathanSalwan/ROPgadget>).

1. Though there are a number of ways you could implement a return-oriented program, your ROP should use the `execve` system-call to run the `"/bin/sh"` binary. This is equivalent to:

```
execve("/bin/sh", 0, 0);
```
2. For an extra push in the right direction, `int 0x80` is the assembly instruction for interrupting execution with a `syscall`, and if the `EAX` register contains the number 11, it will be an `execve`. Now all you need to figure out is what values you need for `EBX`, `ECX`, and `EDX`, and set them using ROP gadgets!

⁷for example, see the documentation for `bash` <https://linux.die.net/man/1/bash>

What to submit Create a Python program named `sol6.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./vulnerable6 $(python sol6.py)
```

You may find the `objdump` utility helpful.

For this target, it's acceptable if the program segfaults after the root shell is closed.

vulnerable7: Heap-based exploitation [Extra credit]

(Difficulty: Hard)

This program implements a doubly linked list on the heap. It takes three command-line arguments. Figure out a way to exploit it to open a root shell. You may need to modify the provided shellcode slightly.

What to submit Create a Python program named `sol7.py` that prints lines to be used for each of the command-line arguments to the target. Your program should take a single numeric argument that determines which of the three arguments it outputs. Test your program with the command line:

```
./vulnerable7 $(python sol7.py 1) $(python sol7.py 2) $(python sol7.py 3)
```

5 Autograding

We will be automatically grading your programs. Your program should not print any extra characters or white space to the output. The picture below includes sample input/output for the programs. Please note that we do not include sample output for Program 5 since it will partially release the solution. All your file names should follow the guidelines mentioned in Section 6.

The image displays four terminal windows, each showing a successful exploit of a different vulnerable program. In each case, the user runs a command like `./vulnerableX $(python solX.py)` or `python solX.py > tmp; ./vulnerableX tmp`, which results in a root shell being granted, indicated by the prompt changing from `cs526@kali` to `root`.

```
Terminal - cs526@kali: ~/Desktop/cs526
File Edit View Terminal Tabs Help
cs526@kali:~/Desktop/cs526$ ./vulnerable1 $(python sol1.py)
# whoami
root
# █

Terminal - cs526@kali: ~/Desktop/cs526
File Edit View Terminal Tabs Help
cs526@kali:~/Desktop/cs526$ ./vulnerable2 $(python sol2.py)
# whoami
root
# █

Terminal - cs526@kali: ~/Desktop/cs526
File Edit View Terminal Tabs Help
cs526@kali:~/Desktop/cs526$ python sol3.py > tmp; ./vulnerable3 tmp
# whoami
root
# █

Terminal - cs526@kali: ~/Desktop/cs526
File Edit View Terminal Tabs Help
cs526@kali:~/Desktop/cs526$ ./vulnerable4 $(python sol4.py)
# whoami
root
```


6 Deliverables, deadline, and other information

Please submit your project online via Brightspace.⁸ You can submit the components as many times as you like (n.b., subsequent submissions overwrite previous ones). The deadline is firm so please do not wait until the deadline to upload your assignment.

For this project you will submit the following files:

Final submission

Filename	Description
answers.pdf	A PDF file containing your responses to <u>all</u> project tasks and questions
cookie	Generated by setcookie based on your username
sol1.py	Your exploit targeting vulnerable1
sol2.py	Your exploit targeting vulnerable2
sol3.py	Your exploit targeting vulnerable3
sol4.py	Your exploit targeting vulnerable4
sol5.input.txt	The input file you used in targeting vulnerable5
sol5.commands.txt	A text file containing the user commands used for targeting vulnerable5
sol6.py	Your exploit targeting vulnerable6 [Optional extra credit]
sol7.py	Your exploit targeting vulnerable7 [Optional extra credit]
other files	Any additional files or programs created to support your efforts

Table 1: Your final submission tarball should include the following files. Please name it `<username>.proj1.tgz`

Guidelines

The file `answers.pdf` should contain your responses for each project question or prompt. You may use any word processor or typesetting environment of your choosing in creating this file, but you must submit a valid PDF file. Note that you are not allowed to collaborate with anyone on this project. Also note: we will deduct points from your project for excessive violation of reasonable organization, good grammar, or proper spelling.

If there is anything non-standard involved in compiling or running your attack files (or anything that you think it would make the TA's life easier to know when grading your assignment), please include a README as well with any additional information.

Your files can make use of standard libraries and the provided `shellcode.py`, but they must be otherwise self-contained. Do not include `shellcode.py` with your submission. Be sure to test that your solutions work correctly in an unmodified copy of the provided VM (ie: without installing any additional packages or setting environment variables).

You must upload your assignment as a compressed tar archive.⁹ First, make a directory with a name that is formatted in the style `<username>.proj1` using your Purdue Career Username in place of `<username>`.

⁸See the *Project Submission* link on Brightspace.

⁹See [http://en.wikipedia.org/wiki/Tar_\(file_format\)](http://en.wikipedia.org/wiki/Tar_(file_format)).

For example, if H. G. Wells were in our class, he might make a directory named `hgwells1.proj1`. Copy all of the Project 1 materials we will need for grading your assignment to this directory and then create a compressed tar archive of the directory formatted as `<username>.proj1.tgz`, e.g., `hgwells1.proj1.tgz`. For those of you unfamiliar with this process, we list the commands Mr. Wells would have used in Table 2.

Command	Description
<code>cd</code>	Change current directory to your home directory
<code>mkdir hgwells1.proj1</code>	Make directory <code>hgwells1.proj1</code>
<code>cd hgwells1.proj1</code>	Change directory to <code>hgwells1.proj1</code>
<code>cp ~/project1/answers.pdf .</code>	Copy the listed file to the current directory
<code>cp ~/project1/sol1.py .</code>	—
<code>cp ~/project1/sol2.py .</code>	—
<code>cp ~/project1/sol3.py .</code>	—
<code>cp ~/project1/sol4.py .</code>	—
<code>cp ~/project1/sol5_input.txt .</code>	—
<code>cp ~/project1/sol5_commands.txt .</code>	—
<code>cd</code>	Change current directory to your home directory
<code>tar zcvf hgwells1.proj1.tgz hgwells1.proj1</code>	Create a compressed tarball from your directory

Table 2: The commands H. G. Wells used to create his Project 1 tarball.

Once you have created your tar archive, you will likely need to copy it from your virtual machine to your host (or somewhere else) so that you can upload it to Blackboard. If you don't know how to do this or have problems creating your tar file, please post a message to the discussion board so that someone can assist you. Before submitting your project, we strongly advise that you copy your tar file to a temporary directory and unarchive it¹⁰ to personally verify that you've created a complete and valid tar file. We will deduct points if your tar file does not follow the conventions we have described.

¹⁰Mr. Wells would use the command `tar zxvf hgwells1.proj1.tgz` to unarchive his tarball.