CSE127: Computer Security **Winter 2026**

PA2 — Buffer Overflows

Project Release: January 16, 2026 PT
Deadline: January 29, 2026 by 11:59:59pm PT

Introduction

Alice's company **Security4All** is having one of their periodic security audits taking place today. The software used for this purpose has flagged a bunch of code snippets across various Security4All projects as potentially being unsafe. Unfortunately, seven of the flagged threats belong to projects under her ownership. However, before Alice can patch the security bugs, she wants to verify that the threats detected are indeed exploitable, and not false positives.

Alice was very happy with your assistance helping her fix the compiler bug last week, and asks for your help again. Your task is to help Alice develop working exploits for each of the threats flagged by the software tool. Happy Hacking!

Goal

This project will introduce you to control-flow hijacking vulnerabilities in application software such as buffer overflows. We will provide a series of vulnerable programs and a virtual machine environment in which you will develop exploits.

target0–target6 are required. target7 is extra credit!

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 other's 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! You are required to respect the privacy and property rights of others at all times, or else you will fail the course.

Forming a Group

This is a group project; you can work in a team of size at most two and submit one project per team. You are not required to work with the same partner on every project. You and your partner should collaborate closely on each part.

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

Solutions must be submitted to Gradescope.

Index

- Submission Details
- FAQ

Setup

Buffer-overflow exploitation depends on details of the target system. You must develop and test your attacks inside the CSE127 PA2 VM, as it has been configured to disable certain security features that would complicate your work.

1. Download the appropriate VM image for your platform and run it.
   Windows, Linux, and Intel Macs: Download the .ova file and import it into VirtualBox.
   Mac users with Apple Silicon: Download the .zip to get a UTM file and use UTM to open it.

   The username and password are both: `cse127`

   Once the VM is running, you may SSH in: `ssh -p 2222 cse127@localhost`

   Copy files using SCP: `scp -p 2222 -r /path/to/files/ cse127@localhost:/home/cse127`

   You can also use VS Code via SSH see the Microsoft docs.

2. Download the assignment starter code inside the VM with the command

   `wget https://cseweb.ucsd.edu/classes/wi26/cse127-a/resources/pa2-starter.tar.gz`

   and

   `tar -xf pa2-starter.tar.gz`

   to unzip it. You should see a `targets` directory—enter that directory.

3. Run `./build.sh`. It will prompt you for usernames. Use the usernames registered in your gradescope accounts. For example, if you're gradescope email is `kumarde@ucsd.edu`, you would give input `kumarde`.

   If you are in a group, enter both usernames separated by a space. The order of username doesn't matter. For example, if the usernames are `kumarde` and `savage`, running `./build.sh` should generate a cookie file. Take a look at its contents.

   ```
   cse127@cse127:~/targets$ cat cookie
   kumarde savage
   6351
   ```

   If you upload that to gradescope, you can confirm that the cookie you generate is the one we expect. If not, an error is generated noting the mismatch.

   Please make sure you have correctly generated your cookie file before you start hacking!

4. If you are working in a team of 2, make sure to add both you and your teammate on your Gradescope submission using the "Add Group Member" button (the cookie test case will only pass once both have been added).

5. If you need to recompile your targets, you will need to run `./build.sh clean` before you run `./build.sh` again.

Resources and Guidelines

No attack tools allowed!

Except where specifically noted, 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-flow hijacking tutorials

Before you begin this project, review the slides from the buffer overflow lectures and attend discussion for additional details. You can find more details in the article "Smashing the Stack for Fun and Profit."

GDB

You will make use of the GDB debugger for dynamic analysis within the VM, hopefully leveraging your learnings from PA1. This quick reference on GDB may help refresh your memory.

x86 assembly

These are many good references for x86, but note that our project targets use the 32-bit x86 ISA with AT&T syntax. The stack is organized differently in x86 and x64. If you are reading any online documentation, ensure that it is based on the x86 architecture, not x64.

If you are getting a segfault

A segfault means that you're either jumping execution to or dereferencing an address that is incorrect. This means you're probably on the right track because you've overwritten something and changed the program's behavior!

If you are stuck as to where to start looking, try to identify the addresses the exploit has changed and work from there, i.e. make sure the addresses you intended to change have actually been changed and nothing else.

Targets

All the flagged programs are short C programs with (mostly) clear security vulnerabilities. Furthermore, we have provided source code and a build script that compiles all the targets. Your exploits must work against the targets as compiled and executed within the provided VM.

target0: Overwriting a variable on the stack (2 points) (Easy)

This program takes input from stdin and prints a message. Your job is to provide input that causes the program to output: "Hi {username}! Your grade is A+." (You can use either group member's username.) To accomplish this, your input will need to overwrite another variable stored on the stack.

Here's one approach you might take:

1. Examine target0.c. Where is the buffer overflow?
2. Disassemble _main. What is its starting address?
3. Set a breakpoint at the beginning of _main and run the program.
4. Using GDB from within the VM, set a breakpoint at the beginning of _main and run the program.

   ```
   (gdb) break _main
   (gdb) run
   ```

5. Draw a picture of the stack. How are name[] and grade[] stored relative to each other?
6. How could a value read into name[] affect the value contained in grade[]? Test your hypothesis by running ./target0 on the command line with different inputs.

Be careful about null terminators!

What to submit

Create a Python 3 program named sol0.py that prints a line to be passed as input to the target. Test your program with the command line:

```
$ python3 sol0.py | ./target0
```

Hint: In Python 3, you should work with bytes rather than Unicode strings. To construct a byte literal, use this syntax: b"\xnn", where nn is a 2-digit hex value (e.g., b"\x70"). To repeat a byte m times, you can do: b"\xnn" * m. To output a sequence of bytes, use:

```
import sys
sys.stdout.buffer.write(b"\x61\x62\x63")
```

Don't use print(), because it automatically encodes whatever is being printed with the default encoding of the console. We don't want our payload to be encoded, so we use `sys.stdout.buffer.write()`.

target1: Overwriting the return address (3 points) (Easy)

This program takes input from stdin and prints a message. Your job is to provide input that makes the program output: "Your grade is perfect." Your input will need to overwrite the return address so that the function `vulnerable()` transfers control to `print_good_grade()` when it returns.

1. Examine target1.c. Where is the buffer overflow?
2. Examine the function `print_good_grade`. What is its starting address?
3. Using GDB from within the VM, set a breakpoint at the beginning of `vulnerable` and run the program.

   ```
   (gdb) break vulnerable
   (gdb) run
   ```
4. Disassemble `vulnerable` and draw the stack. Where is `input[]` stored relative to `ebp`? How long would an input have to be to overwrite this value and the return address?
5. Examine the `esp` and `ebp` registers:

   ```
   (gdb) info reg
   ```
6. What are the current values of the saved frame pointer and return address from the stack frame? You can examine two words of memory at `ebp` using:

   ```
   (gdb) x/2wx $ebp
   ```

   Essentially, this command says "e(x)amine the memory at location $ebp. Give me two words (4 bytes per word, so 8 bytes in total) and put the result in hexadecimal.
7. What should these values be in order to redirect control to the desired function?

What to submit

Create a Python 3 program named sol1.py that prints a line to be passed as input to the target. Test your program with the command line:

```
$ python3 sol1.py | ./target1
```

When debugging your program, it may be helpful to view a hex dump of the output. Try this:

```
$ python3 sol1.py | hd
```

Remember that x86 uses little endian ordering. Use Python's `to_bytes` method to output 32-bit little-endian values like so:

```
import sys
sys.stdout.buffer.write(0xDEADBEEF.to_bytes(4, "little"))
```

target2: Redirecting control to shellcode (3 points) (Easy)

Targets 2 through 7 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 several of 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. 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.

1. Examine target2.c. Where is the buffer overflow?
2. Create a Python 3 program named sol2.py that outputs the provided shellcode:

   ```
   from shellcode import shellcode
   import sys
   sys.stdout.buffer.write(shellcode)
   ```

3. Disassemble vulnerable. Where does buf begin relative to ebp? What is the offset from the start of the shellcode to the saved return address?
4. Set up the target in GDB using the output of your program as its argument:

   ```
   $ gdb --args ./target2 "$(python3 sol2.py)"
   ```

5. Set a breakpoint in vulnerable and start the target.
6. Identify the address after the call to strcpy and set a breakpoint there:

   ```
   (gdb) break *<address>
   ```

   Continue the program until it reaches that breakpoint:

   ```
   (gdb) cont
   ```

7. Examine the bytes of memory where you think the shellcode is to confirm your calculation:

   ```
   (gdb) x/32bx 0x<address>
   ```

8. Disassemble the shellcode:

   ```
   (gdb) disas/r 0x<address>,+32
   ```

   How does it work?

9. Modify your solution to overwrite the return address and cause it to jump to the beginning of the shellcode.

What to submit

Create a Python 3 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:

```
$ ./target2 "$(python3 sol2.py)"
```

If you are successful, you will see a root shell prompt (`#`). Running `whoami` will output "root". Running `exit` will return to your normal shell.

If your program segfaults, you can examine the state at the time of the crash using GDB with the core dump: `gdb ./target2 core`. To enable creating core dumps, run `ulimit -c unlimited`. 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 ./target2` in order for the core dump to be created.

target3: Overwriting the return address indirectly (3 points) (Medium)

In this target, the buffer overflow is restricted and cannot directly overwrite the return address. You'll need to find another way. Your input should cause the provided shellcode to execute and open a root shell.

What to submit

Create a Python 3 program named sol3.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
$ ./target3 "$(python3 sol3.py)"
```

target4: Beyond strings (3 points) (Medium)

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 (all little endian). Create a data file that causes the provided shellcode to execute and opens a root shell.

Hint: First figure out how an attacker can cause a buffer overflow in this program. Note that the `read_elements` function breaks the for-loop once the end of the file is reached, so the 32-bit count does not need to be truthful.

What to submit

Create a Python 3 program named sol4.py that outputs the contents of a data file to be read by the target. Test your program with the command line:

```
$ python3 sol4.py > tmp; ./target4 tmp
```

target5: Bypassing DEP (3 points) (Medium)

This program resembles `target2`, but it has been compiled with data execution prevention (DEP) enabled. DEP means that the processor will refuse to execute instructions stored on the stack. You can overflow the stack and modify values like the return address, but you can't jump to any shellcode you inject. You need to find another way to run the command `/bin/sh` and open a root shell.

What to submit

Create a Python 3 program named sol5.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
$ ./target5 "$(python3 sol5.py)"
```

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

Warning: Do not try to create a solution that depends on you manually setting environment variables. You cannot assume that the autograder will run your solution with the same environment variables that you have set.

target6: Variable stack position (3 points) (Medium)

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 target2, 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 3 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:

```
$ ./target6 "$(python3 sol6.py)"
```

Warning: If you see any output before the root shell is opened, you have not done this target correctly and your solution will not be accepted by the autograder.

target7: Return-oriented programming (4 points) (Extra Credit) (Hard)

This target is identical to target2, but it is compiled with DEP enabled. Implement a ROP-based attack to bypass DEP and open a root shell.

It will be helpful to use a tool such as ROPgadget; this is an exception to the "no attack tools" policy. The ROPgadget command is already installed on the provided VM. View its usage by running ROPgadget -h. The –binary, –badbytes, –multibr, and –ropchain flags will be particularly helpful.

1. Though there are a number of ways you could implement a ROP exploit, for this target you should use the setuid syscall to become root, followed by the execve syscall to run the /bin/sh binary. This is equivalent to:

   ```
   setuid(0);
   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. If the EAX register contains the number 23, the syscall will be setuid; if it contains 11, the syscall will be execve. You need to figure out what values you need for EBX, ECX, and EDX, and set them using ROP gadgets!

3. We recommend that you start by getting the execve call to work on its own, without setuid. When you do this correctly, it will open a shell, but you won't be root. Then modify your solution to make it call setuid first, and you'll get a root shell.

What to submit

Create a Python 3 program named sol7.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
$ ./target7 "$(python3 sol7.py)"
```

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

Note: Tutors/TAs will not offer help for the extra credit problem.

Submission Details

1. If working in a group of 2, add your teammate using the "Add Group Member" option on Gradescope. Either you or your teammate can submit to gradescope and that would count for the both of the you.

2. Submit the following files as a group submission on Gradescope.

   - `sol[0-6].py`
   - `sol7.py` (only needed for EC)
   - `cookie`

3. The Gradescope autograder will not test your submission when you submit, only check for the Python solution files and whether the provided cookie is what we expect.

   You will need to make sure you are satisfied with the correctness of your submission when you submit.

   Again, the score on Gradescope is not your score for the assignment.

   8.0/8.0 does not guarantee any points on the assignment, but you do need to pass the sanity checks to have any possibility of getting points.

   If you do not pass the Cookie validity test, you won't pass any of the actual tests.

Your files can make use of standard Python 3 libraries and the provided `shellcode.py`, but they must be otherwise self-contained. Do not modify or include the targets, build script, `helper.c`, `shellcode.py`, etc. Be sure to test that your solutions work correctly in an unmodified copy of the provided VM, without installing or updating any packages or changing any environment variables.

AI Attestation

Submission

Submit your attestation to using AI on this PA (Yes or No) to "Assignment 2: AI Attestation" on Gradescope.

Frequently Asked Questions

Q: HELP! I'm getting a scary message when I try to ssh into the machine at cse127@localhost. It says "WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!" Am I getting hacked!

A: If you used `ssh -p 2222 student@localhost` to get into the previous assginments's virtual machine, SSH may falsely flag this as a security issue due to the changed signature. The PA1 virtual machine and the PA2 virtual machine are two different machines, but SSH doesn't know this, all it knows is that it connected to a server called `'localhost'` before, and the signature provided by this `'localhost'` server changed since last time. To get rid of this false alarm, go to the `.ssh/known_hosts` file, find the line that starts with localhost, and delete it.

Q: I'm a Mac user using UTM and can't ssh into my VM after I start it, when I type in the ssh command it gives me a "connection refused" error.

A: The network configuration for the .utm file may be set up incorrectly. You can fix it with the following steps:

1. Stop the virtual machine, if it is still running.
2. In the upper right hand corner of the UTM window, click on the settings icon (it should look like three horizontal sliders on top of each other).
3. A "Settings" window should pop up. Go to Network $->$ Port Forwarding. There should be a list of port forwarding configurations.
4. Go to the line with Guest Port = 22. Edit that line so that the Host Port = 2222.

After these steps, you should be able to ssh into your VM using port 2222.

Q: I'm getting an "ignored null byte in input" but I didn't put any null bytes in my input.

A: Targets 2, 3, 5, 6, and 7 require you to pass in a value as a command-line argument, but arguments in Unix cannot contain null bytes. (The other targets, which read data from stdin or a file, don't face this challenge.)

Using target 2 as an example, you can examine the exact bytes of your solution using this command:

```
$ python3 sol2.py | hd
```

Do you see a null byte? Something in your Python code like this example may be causing it:

```
(0xFF).to_bytes(4, little)
```

This will format the integer as 4 bytes in little endian. If the value is too small, it will be padded with zeros, such that the line produces bytes 0xFF, 0x00, 0x00, and 0x00. It is also possible that an address you're trying to use happens to have a null byte. In that case, try to find an alternative way to accomplish what you're trying to do by, for example, using a copy of the data located at a different address, or overwriting a different function's return address.

Q: I get a root shell when I run `sudo ./targetX "$(python3 solX.py)"`. Am I done?

A: No! You should only run `./targetX "$(python3 solX.py)"` without sudo. If you run it under sudo, then your shells will always be spawned as the root user, whether you have accomplished the task of opening a root shell or not.

Q: My solution works in GDB but not from the command line.

A: The most likely explanation is that you're referencing data from `argv[]`. Since `argv[]` comes from outside of `_main`'s stack frame, its position can vary depending on the size of the environment and arguments, which can be slightly different when running under `gdb`. The best solution is to find the data you need in the stack frame of the vulnerable function, rather than from `argv[]`.

Q: I'm tired of starting gdb over and over with my new input, is there a way to just run it again with new args without restarting it?

A: Instead of running gdb every time, you can just run `run $(python3 solX.py)` in the same gdb instance, and it should restart with the output of your updated python script (or use `run` or `r` without arguments, as this will use the last arguments given).

Q: I'm tired of starting gdb over and over with my new input, is there a way to just run it again with new args without restarting it?

A: Instead of running gdb every time, you can just run `run $(python3 solX.py)` in the same gdb instance, and it should restart with the output of your updated python script (or use `run` or `r` without arguments, as this will use the last arguments given).

Q: When I try to rebuild the cookie, I get an error like:

```
Traceback (most recent call last):
File "/home/cse127/targets/./build.py", line 120, in module
generate(workdir, offsets, rootdir, 0xffff_0000 - cookie)
File "/home/cse127/targets/./build.py", line 96, in generate
with open(output_file, 'wb') as out:
PermissionError: [Errno 13] Permission denied: '/home/cse127/targets/target0'
```

A: Remember to run ./build.sh clean before you run ./build.sh again.

Q: I'm sure my cookie is right, and it looks identical to the one in the autograder, but the autograder isn't accepting it!

A: The cookie generated by the script doesn't have an end of file newline. Many IDEs automatically add this newline, so it's a very subtle issue. Maybe your IDE has added a newline at the end of your file without you knowing about it. See here on how to resolve this using vim.