# Analysis of NX with Peekaboo

## Pre-requisites

1. Peekaboo
2. udcli

## Setting up of vuln binary

```

cd ~/CS5231/binary

gcc -fno-stack-protector -z execstack -no-pie -o nodef/vuln vulnerable.c

gcc -fno-stack-protector -no-pie -o nx/vuln vulnerable.c

```

Also, make sure the binaries are of the expected security defences by using `checksec`

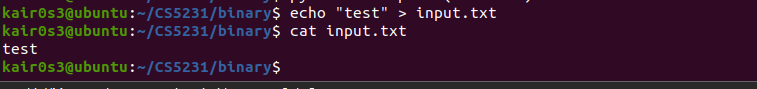
Text

Description automatically generated

Now, we can move to the actual analysis.

## 1. Using normal expected input

First adding some simple test data into `input.txt`



Running the binary with the input, shows us that the following instructions seen are the same.

Text

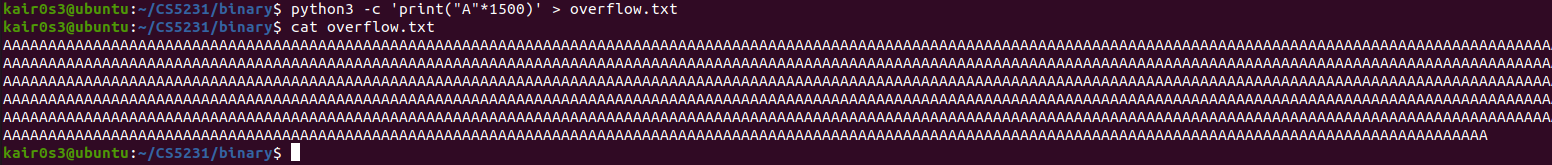
Description automatically generated

What if we force it to overflow, given the assumption that we know the code?

## 2. Using intended overflowing input

Similarly, let’s generate the input file, this time with ridiculously large input that we know will overflow 1337 bytes.

And, this time, we will use python3 to generate that overflow input.



With the newly created overflow input, let’s try running the program with peekaboo again.

Running peekaboo on the respective vulnerable programs shows us that the instructions seen are the same again.

Text

Description automatically generated

Interestingly, there is no additional instructions being called at all for both the non-overflowing input and the overflowing input.

Perhaps, the instruction for NX is only called when an instruction on the stack marked as non-executable is called? This will make more sense as it relates to the design of efficiency and performance, as additional redundant checks will reduce performance.

Let’s find out if that is true, by explicitly modifying the return address to within the stack. If this indeed returns additional instructions, then our deduction is likely to be correct.

## 3. Using an overwrite RIP input

Using a manual script `generatePayload.py`, we can generate the `payload` file which enables to exploit the vuln without any defences on. This script is intended to exploit it by printing out the `/etc/passwd` file.

Text

Description automatically generated

However, since the addresses are hardcoded and the binary is run from a different directory, the address we overwrite in RIP will likely not be the shellcode but within the stack region.

Thus, this gives us the opportunity to test the `payload` as our input without worrying about accounting for the additional instructions used to print the `/etc/passwd` since the RIP would most likely not be going to the location of the intended shellcode.

Running the peekaboo command with the `payload` as the input now does show some interesting differences.

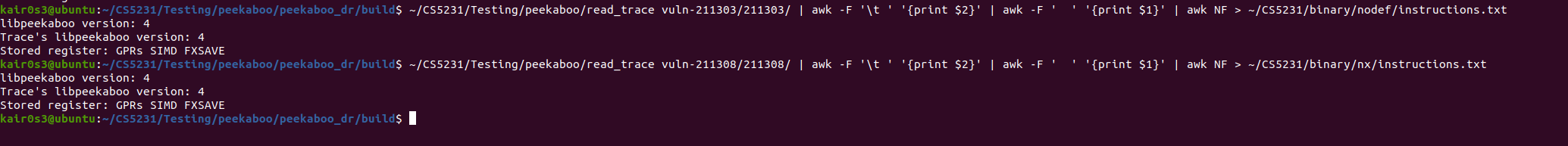
Text

Description automatically generated

For the binary with no defences, 188722 instructions were seen while 188727 instructions were seen when NX was up.

But what are these additional instructions? Taking note of the PIDs, we have PID 211303 for the binary with no defences and PID 211308 for the NX.

Now, let’s try to trace and obtain the instructions, specifically only the instruction bytes.



With that, we can obtain the instructions of the respective runs.

Text

Description automatically generated

To find out the differences between the 2 instruction files, we can use the utility `diff`.

Text

Description automatically generated

Here, we see some of the differences. Specifically, from instruction 188721 and onwards. Let’s try to translate the respective instructions to something more readable/understandable.

For the instructions of no defences,

Text

Description automatically generated

For the instructions of NX,

Text

Description automatically generated