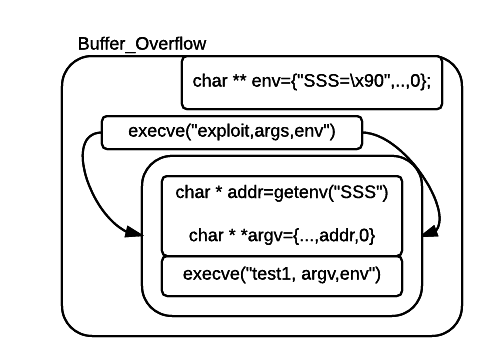
**Describe the problem that you are solving**

The purpose of this project was to develop a program that understand a human like description of a buffer overflow and automatically exploits that vulnerability, when given a vulnerable binary and the vulnerability’s description. Our goal was that our program should successfully run the given binary, overflow the binary’s vulnerable buffer and get the binary to execute code of our choosing.

**Describe the approach you took to implementing the project**

We used ‘execve()’ to run and pass arguments and environment to the binary. The main challenge was to make the binary’s execution jump to our code, as our program will be run on different machine and would have to work for different binaries. If the code we want to be executed was passed to the binary through the vulnerable its address would change every time we change binary or the computer, moreover the buffer might too small to fit our shell code in there.

Instead of using the overflown buffer to pass our shell code to the binary, we decided include it as a value of an environment variable, and use ‘getenv()’ to find where our shell code is located. The problem is ‘getenv()’ only get addresses for variable form the current environment, not the ones in the execve()’s environment. To overcome this we create a new program called “buf\_overflow” which will create the environment variable and pass it the second program called “exploit” that will then get the variable’s address and call vulnerable binary with the same environment.



**Describe what a developer or security expert must do to use your approach**

To use my program a developer must only have access to a Linux system, they must also know in advance the size of the input that will overwrite the instruction pointer (EIP) for the particular binary they want to exploit. They can pass the name of the binary name, size of the input that will overflow the buffer and total number of arguments and the input position using –test, -size, -args, -place respectively.

D**escribe the limitations of your tool: What types of applications can it be applied to?**

The tool supports stack overflows, so buffers that are kept on the stack can be exploited, for instance If the binary uses malloc to allocate space on heap, our tool would not be able to exploit that.

The binary should be compiled using flags that makes the stack executable namely “-z execstack” and “fno-stack-protector”, so that the shellcode that we used can be executed on the stack

We write the application in such a way that it can overflow binary applications that use char buffers

**Describe the future work that you can see which would improve your tool.**

The program can be enhanced by one of the following ways:

By pass non-executable stacks

Support other types of overflows such as integer and array index overflows