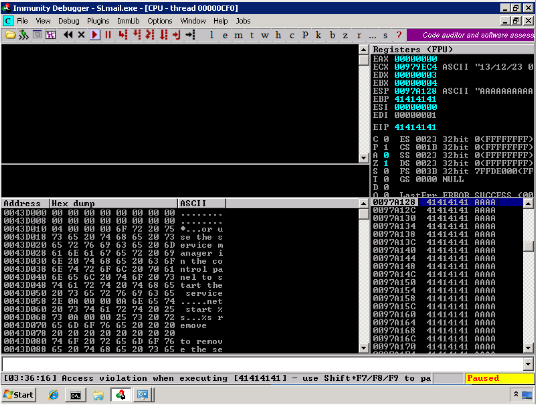
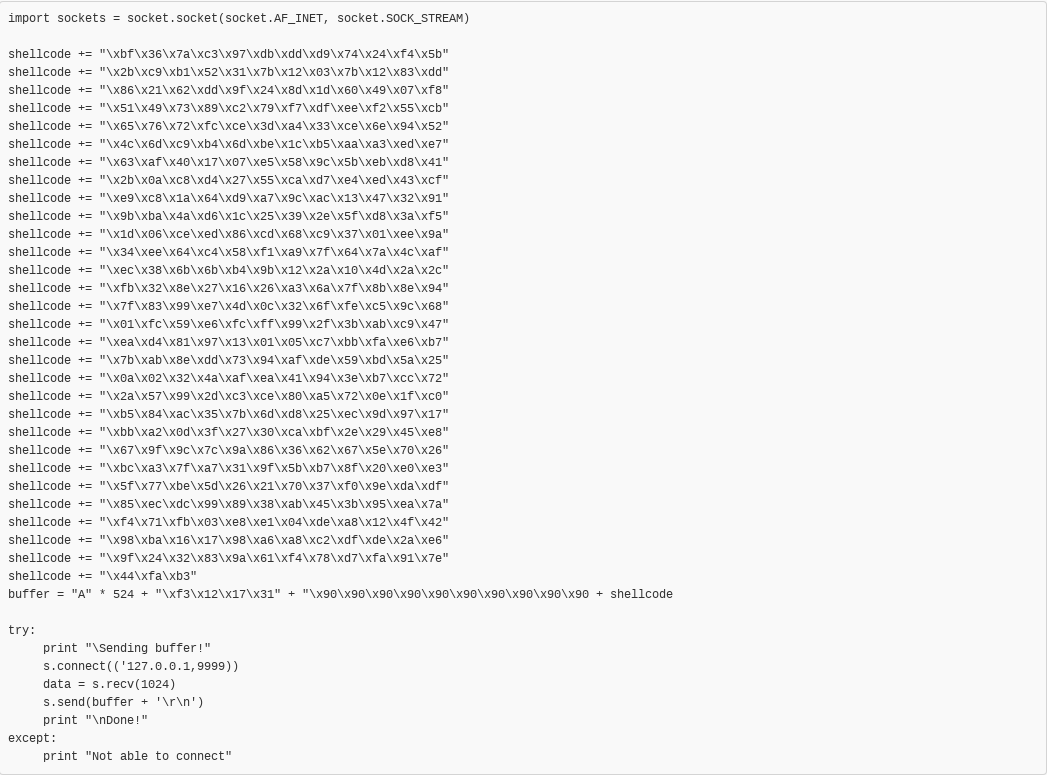
Misuse case:

Base on my personal experience with reverse engineering BoF (buffer overflow) attacks, I knew this would be a glaring misuse case. Even with secure code, the fact that the usage of the application was: **./build/file <STRING\_TO\_SEARCH>** could implement some problems. This allows a user to send a string over a network, with unlimited characters. The first step to probing for a BoF, besides reviewing source code, is to send a long string of \x41 and open a debugger, to see how the application behaves. An example of a vulnerable application is listed below:



What this image means, is that a string of A’s was received by a server. The server took this string and passed it through the program. If you look above to the error- the **EIP** (Extended Instruction Pointer) was filled with A’s. This means, the EIP was controlled. The EIP controls the flow of execution for the stack. If an attacker can control where an EIP will execute in memory, malicious shellcode/hexadecimal code can be pointed to, from EIP, allowing for code execution. This would have been easy to test for, because any string can be sent to the server, and the server expects it. I needed to figure out a way to mitigate this concern.

Here is an example of a Buffer Overflow I created for a vulnerable application VERY similar to the application used in our project, in preparation for my OSCP certification. This application listens on port 9999. After reverse engineering, here is what I created. Notice that because of the vast amount of data being sent, it can overflow buffers:

Mitigation:

To mitigate this concern, I limited the input of the user. I figured there are not many words greater than 15 characters, so I limited input to this size. That means, a malicious attacker cannot just start probing for overflow vulnerabilities. In addition, a buffer overflow would be very hard to create without the source code or being able to send large data to test for overflows.