# Assignment Objectives

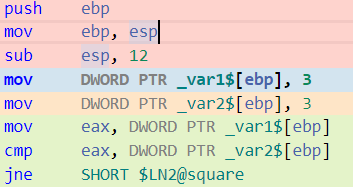
* See if any overflows
* See if nasty strings break
* Find Usernames
* Find Passwords
* Create modified binary to automatically get access

Issues

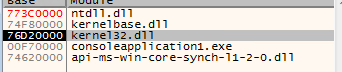
* Restarting exe/address changes
* Python script with pipes to call your program

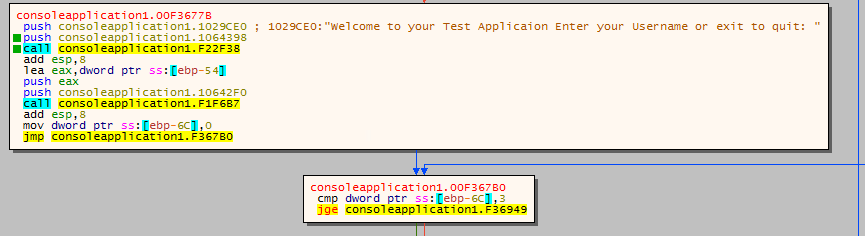
Questions:

* Is there a better way to “restart” the program, redragging the menu is annoying, and not sure if the addresses change.
* Fuzzing this application is time consuming, and we can’t easily deliver an exploit to others. Have can we write a script to interact with this application?
* How can we easily view the Heap?
  + You can search in the symbol table for “malloc” and then set a breakpoint on it, but there are tons of calls to malloc when printing out things and using other standard library functions, so that doesn’t help that much.
  + In x64dbg, go to the pointer referencing your allocated memory, right click, and chose “follow in dump”. This will give you the memory addresses that the pointer was keeping track of.
* How can we easily view things on the stack?
  + The program stack can be viewed in the bottom right
  + Below is an example where two variables were created on the stack, both assigned the number 3, compared to each other, and then jumped if not equal



* + You will often see things like [esp-4] to make room on the stack for integers
* How can we view Global/Static variables?
  + Looks like most number variables are in .data. just translate from hex to decimal.
  + Looks like read-only data, such as literal strings, constants, and debug directory information are in .rdata
* How can we view all custom made function calls?
  + The .pdb file would tell you all of the useful debugging info, but we don’t have that, we just have the executable. This is the Microsoft file type for debugging windows apps.
  + Function calls with lower addresses are user defined ones though:



See how the .dll’s are 7 addresses where as the custom.cpp application is at address x00F22F38… 

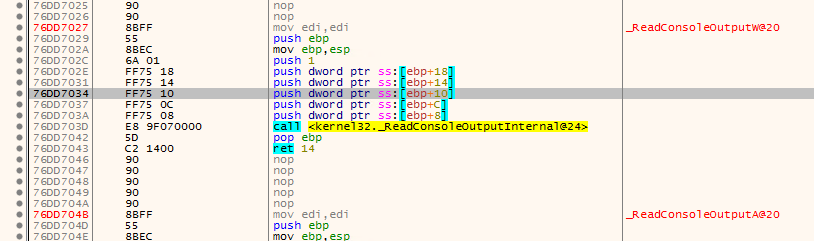
Well if we look at the memory map, these functions are in the .text section – which contains machine code of all functions we would have written in C sourcefile.

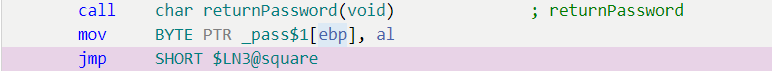


So did Professor Randall just write his own GetConsoleInput functions to screw with me? No.

If the Microsoft symbols are included, and the application is a standard exe, x64dbg should be able to identify any library calls like “malloc” or “cout”

* + For external libraries used in the application: you can right click when in the external module and chose “download symbols for this module”. This will then fillout the names of all of the functions in the “comment” section of the disassembler.



* + When the application prompts for input, we are in a x777~ memory address which means its an external library function. It always says “kernel32” by EIP
* Where do function calls store their results?
  + Assembly doesn’t have any operations that can call a function and store its return value. A function will change the value of a variable on the stack. If the function returns a value in c++, then the assembly code would: call the function, that function would store the result in one of the registers (like eax or al), and then the line after that function call can do something to that register

Call returnPassword(void) stored its result in “al”. The next line put “al”’s value on the stack to be used later.

* How can I find out how much space was allocated for the username and password? Feeding too many characters to username breaks the application
  + you can watch all calls to \_\_malloc by putting a breakpoint on it. Or you could find the pointer that is looking at the allocated memory, and put a memory breakpoint on that, so anytime that is assessed, it pauses
  + Yes the heap grows upward and has a specific address area, but it isn’t allocated like the stack. There can be pointers everywhere throughout the heap, some can be a couple bytes and others can be huge. So there isn’t a good way to visualize this. Just find the variables you care about, and put breakpoints on them if you want to see when they change or are compared.

# Walkthrough

## **Gathering Information**

## **Fuzzing the Application**

## **Controlling Execution**

### EIP

### Finding space for our code

### Calculating program offset

## **Patching**

### Hard Coded Credentials

### Allocation

### Bad Characters

# Extra Information

Execution Breakpoint:

This is the most common and most used type of breakpoint. When you toggle a breakpoint on a specific address, this tell the debugger to stop when that address is reached in the execution. To use this, simply press the F2 when over an address you would like to break on.

Memory Breakpoint:

A memory breakpoint is used to pause an application when a specific area of memory is either accessed, written to, or executed. This is handy when you want to determine when or if a specific area of memory is used by the program. This is available in the right click menu of the memory map window and dump pane.

Hardware Breakpoint:

A hardware breakpoint is used to pause an application when a particular address is either written to, read, or executed. This is specifically useful to determine when a particular variable is set. This can be used for byte, word, and dword reads and writes. This feature is available in the right click menu of the hex dump.

The Call Stack window:

Gives you an extensive list of the functions and procedures (routines) which brought you to your current location. You can use this window to analyze these routines and learn about your application's execution routine.

Memory Map:

This will show us all of the sections or regions of RAM allocated to and by our application and its dependencies.

Patch:

You can change the executable and then click the patch button to save the new modified exe. Otherwise, the exe wouldn’t work right because of the changing addresses.