Assignment: Final Project

Description:

I've never been a huge fan of multiple-choice quizzes, and I don't feel that's a great way to assess your software exploitation knowledge anyhow. So, rather than a final *exam* let's do a final *project*! I hope this will be more valuable, and more fun.

For your final project I would like you to build a software exploitation challenge of your own. :)

Specifically, I'd like you to create, and solve, a binary exploitation lab problem similar to those I have challenged you to solve throughout the semester. I will leave all of the creative details up to you, but will provide the following minimum requirements…

1. This should be a binary exploitation problem, i.e. a memory corruption vulnerability exploited at the assembly-code level, like we have been working on in class. For instance, a stack or heap buffer overflow, an uninitialized variable bug, a double-free, a use-after-free, etc. I would also accept a race-condition bug, as an exception to that rule. Higher level bug classes such as XSS, SQL injection, command injection, etc are not eligible.
2. As this is a binary exploitation problem, your program should be written in a compiled language such as C or C++. I suppose that technically others like Rust, Go, or even raw Assembly could work, but I would recommend C/C++.
3. Your problem should have some basic theme or plausible story around it. For instance a three-line program that reads one line of input from the user causing a stack buffer overflow is not sufficient. However you might create an "interview application" (just an example) which asks the user some job-related questions, one of which results in an overflow. Doesn't need to be complicated or well polished, just make it feel a little bit like a "*real*" program, as I have done with your lab problems.
4. You should provide a short write-up and working solution (exploit) for your problem, proving it works and explaining the details. Ideally this should use Python3 and pwntools, but you're welcome to use another language if you prefer.

I have provided some tips and tricks for developing software exploitation problems on the following page(s). Feel free to use these for inspiration!

Deliverables:

Please turn in…

1. The problem itself. This should probably include...
    1. The source code to your binary exploitation problem (i.e. a "*.c*" file)
    2. The compiled binary version of your problem (i.e. an executable file)
    3. A build script or instructions for compiling your the binary.
2. A short written description of the problem. Please address...
    1. What software exploitation mitigations are enabled, or not. (i.e. cookies, ASLR, NX, etc)
    2. What/where/why the vulnerability is.
    3. The intended exploitation steps/methodology.
3. A working solution to the problem
    1. A script which demonstrates the solution.
    2. A screenshot of you running the script, showing it works.
4. Would you like this problem to be included in a "community problems" repository for other students? (With attribution of course!) If not that is totally 100% ok.

Tips and Tricks:

The simplest and most popular vulnerability type is a buffer overflow, where too much user-controlled data is written to an array. This might happen on the stack or the heap. These are most commonly exploited by overwriting something else in memory beyond the buffer such as a return address, a pointer, a credential, or a string which the program uses for a sensitive operation. Here is a non-exhaustive list of ways a buffer overflow can occur.

- The `gets()` function may overflow the destination buffer
- A `"%s"` format specifier (string input) may overflow the destination buffer when used with any of `scanf()`, `sscanf()`, `sprintf()`, `fscanf()`, etc.
- Functions which copy data from one buffer to another may overflow the destination buffer, i.e. `strcpy()`, `strcat()`. This can ever occur with size-bounded functions if the "size" parameter is user-controlled, for instance `memcpy()`, `strncpy()`, `strncat()`, etc.
- Loops may copy beyond the end of a destination buffer if the size is not properly calculated. For instance `for (i=0; i<tooBig; i++) { dest[i] = input[i]; }`

Here are a few other vulnerability types, and ways in which they might lead to an exploitable condition.

- Uninitialized variable bugs can be exploitable when the user is able to control the uninitialized memory prior to the program using it, and the variable represents some sensitive value such as a pointer, a size/length, a credential, a command, a filename, etc.
- Double-free bugs can be exploitable when the user is able to allocate controlled data of the same size as another secret/sensitive allocation, potentially resulting in both being allocated in the same place.
- Use-after-free bugs can be exploited to perform a t-cache poisoning attack, which might allow the user to make the program "allocate" a controlled address outside the heap.

When designing your challenge, it would be smart to consider the following questions.

- Will the user need to know the location of anything in memory? If so, you may need to disable ASLR (PIE). Alternatively though you could provide a mechanism for the user to leak, calculate, or guess the necessary address, even with ASLR enabled. The `-no-pie` compiler option can be use to disable PIE.
- Is the goal to execute raw shellcode, use a ROP chain, or just to jump to some "win()" function? If shellcode, DEP/NX will probably need to be disabled. If either of the others, you can likely leave DEP/NX enabled. NX can be disabled with the `-zexecstack` option.
- If the goal is to use a ROP chain, ensure there is enough code in the program to result in a sufficient list of gadgets. Statically compiling the program is an easy way to ensure this! That can be accomplished with the `-static` compiler option.
- If you are using a buffer overflow vulnerability, will the stack cookie prevent the user from overwriting the target? If so, you may wish to disable it. In other situations this may not matter, for instance heap overflows, or overflows which hit other stack variables prior to the return address. Cookies can be disable with the `-fno-stack-protector` option.

I would STRONGLY recommend starting with the vulnerability/exploit idea itself. Build a simple prototype and prove that the idea works. Then construct a theme or scenario around the prototype. For instance, for the "heap" modules I first decided you should be able to create/edit/delete arbitrary chunks of memory. I then themed the program as a "notes" application with that functionality.

Here is another nice list of thoughts and ideas! https://github.com/Naetw/CTF-pwn-tips