EE312 Fall 2014

Exercise 1B, September 12, 2014

The purpose of this exercise is to reinforce our understanding of the stack and to exercise our problem-solving skills. We'll also become familiar with one of the most well-known security vulnerabilities in C programs, the stack-smashing vulnerability. I ask that you complete this exercise in small groups of roughly three-to-five students. There's a lot going on in this exercise, and more eyes on the problem will help discover the "tricks" and more hands will help get the answers hacked out more quickly. Your TA is encouraged to run a small "competition" of sorts, where each team is competing against the other teams to complete the exercise the quickest. Have fun!

Background:

A stack smashing attack is a hacking technique enabled by poor programming, particularly in C/C++ programs. The attack takes advantage of a buffer overflow (indexing an array out of bounds) error in the software. By putting the right values into memory outside the bounds of the array, it is possible to (among other things) overwrite the return address on the stack with a value of the hacker's choosing. Probably the most famous stack smashing attack of all time was the Morris Worm's (developed by Robert Morris, Jr. at Cornell University in 1988). Morris had access to the source code for Unix, and knew that the fingerd program had a buffer overflow defect. Fingerd ran with elevated privileges on most unix systems at that time, and spent its time waiting for input to come over the internet. When used correctly, fingerd would be provided with a person's name or unix login as its input. Fingerd read the input, stored it in an array (a local array, on the stack), and then would search for information about that person's last known activity on the system. The bug in fingerd was that it would read as much character input data as the user provided to it. If the user provided enough character data, the array allocated in the fingerd program, would overflow. Overflow the array by the exactly correct amount, and you would overwrite the return address.

Morris' technique was very sophisticated. He actually transmitted the machine code for his own very short program in lieu of character data. i.e, instead of sending "Craig" to fingerd, he write a program (in assembly, I believe), compiled the program to machine code and then transmitted the sequence of bytes that made up his program. In addition, he transmitted enough data to overflow the buffer and overwrite the return address. Since he'd just transmitted his own program (and since that program was just written into fingerd's local memory), he made sure he replaced the return address with the address of the local variable! When the function eventually returned (after printing out error messages about "no such user"), instead of it returning to its caller, it actually loaded the PC with the address of its own data – data that Morris had populated with a bootstrap loader for his worm. The boostrap loader then opened up a network connection, downloaded a full copy of the worm and ran it.

Your Challenge

There are three parts to this exercise, you won't reproduce every aspect of Morris' attack, but you'll repeat enough of it to really get a handle on the technique and why we as programmers must be careful to avoid the type of security vulnerability that fingerd possessed. Before we jump in there, it's important to know that, specifically because of stack smashing attacks, both GCC and Linux change the way they compile and run programs to make this harder. As we work through this exercise, we'll disable these security features – taking a trip in the way-back machine, to a simpler time…. Anyway, GCC likes to insert a special value (referred to as the "canary") onto the stack near the return address. If you're overflowing a buffer as part of a stack smash, you'll probably overwrite memory all the way up to and over the return address. That means, you'll overwrite the canary. When GCC creates machine code for the function return, it also creates code that looks at the canary. If the canary has been clobbered, the program aborts rather than attempting to return. This feature won't protect us from our "STAGE 1" attack, since in stage 1, you'll be skipping over memory and only smashing the return address (stage 1 isn't a buffer overflow – won't kill the canary). But GCC is smart enough to defeat our attempts to reproduce the Morris Worm's behavior in Stage 2 and Stage 3 of this exercise. So, we'll turn that security check off with the GCC flag "-fno-stack-protector".

When Morris wrote his program he needed to make sure that he put precisely the correct address into memory. Since he was going to overwrite the return address with the address of the fingerd local variable, he needed to know what address (on the stack) fingerd was using for its local variable. That knowledge was only possible because every time fingerd ran, it was assigned exactly the same memory locations to use for its stack (one of the wonders of "virtual memory"). Modern linux systems actually try to make this hard. For no other reason than to provide security, linux chooses a random address as the initial value of the stack pointer. Each time a program is run, the stack pointer is at a different location, preventing an attack as simple as Morris' from being able to know what the "magic input" will be. Fortunately, we can disable this security feature in linux. By launching the program using the setarch command (instead of launching it directly), we can tell linux not to randomize the addresses. To run ./prog do the following:

```
setarch `uname –m` -R ./prog
```

The uname –m needs to be enclosed in single "back quote" characters. There are other ways to launch setarch, but this method is reasonably portable across both 64-bit and 32-bit linux systems.

In between each stage, I'd like the TA to reveal one possible solution to the stage. So, each stage is "timed". If no one wins before the time is expired, that's "score one for the professor!"

Stage 1:

For test1, you write a function doit1. That function can do anything you want, provide that (a) you write the function in C, and (b) it must actually return. The pass this test, the test1 function must call doit1 and then print "you win". Note that test1 is written so that it invokes the "die()" function after calling doit1. If it actually calls die, it will abort and you will not win. So, the goal for test1 is to overwrite the return address in doit1 so that when it returns, it "skips" die. One hint I'll give, in x86 machine code, the "call" opcode is one byte plus the address of the function you're trying to call (which is four bytes). So, the instruction to call die() is exactly 5-bytes long.

Stage 2:

For stage 2 (and stage 3), you no longer get to modify the program. Instead, you must accomplish your hacking solely by providing input data to the program. In stage 2, this input data is in the form of decimal integer data. The test2 program invokes doit2(), then invokes die() and then prints "you win". Once again, your objective is to not invoke die. Hopefully your experience in Stage 1 gives you a good plan of attack. The getData function in stage 2 asks you to provide input data. This very poorly written program has an obvious buffer overflow defect. If you don't terminate the input data, it just keeps reading more and more data. If the program is compiled with –fno-stack-protector then the buffer overflow can be "undetected" and just about anything can happen. To make stage 2 especially easy, getData() displays the current value of each array element before asking if you want to provide a replacement value. Good luck!

Stage 3:

The third and final stage of this exercise simulates much of Morris' attack vector. We won't be uploading any assembly code, but we will be providing input to the program that is nominally "character data", but in actuality bears little resemblance to any actual characters. As a hint, since some of the "characters" we'll want to provide are not ASCII, you will want to run test3 where the input to the program is redirected from a file. When I did stage 3, I ran the program this way

```
setarch `umame –m` -R ./prog < input_file.txt
```

The goal, as with all the previous stages is to not die() and win. You may want to insert print statements and/or run the program in the debugger to develop your "attack" input file. In fact, I've taken the liberty of inserting two print statements into doit3. You are welcome to change these print statements if you'd like, but they have to remain merely print statements (i.e., they can read, but not write memory locations – no trickery here). You are permitted to, but not required to remove the print statements from the program to win. You may also want to create the input file by using a C program to create this file (that's what I did, that's how I got the non-ascii characters I needed).