CS 161 Fall 2005 — Computer Security — Joseph/Tygar/Vazirani/Wagner — HW 3

Due Friday, December 2 at 11am

Please include the following at the top of the first page of your homework solution:

Your full name
Your login name
The name of the homework assignment (e.g. hw3)
Your TA's name

Staple all pages together, and drop them off in drop box #2 (labeled CS161/Fall 2005) in 283 Soda by 11am on the due date.

Homework exercises:

1. **(5 pts.) Any questions?**
   What's the one thing you'd most like to see explained better in lecture or discussion sections? A one-line answer would be appreciated.

2. **(20 pts.) Zero knowledge**
   Alice wishes to prove to Bob that she really does know the private key (N,d) corresponding to her RSA public key (N,e). They decide to use the following protocol:

   1. Bob chooses a random $r \bmod N$ and sends its encryption $s = r^e \bmod N$ to Alice.
   2. Alice decrypts s by computing $s^d \bmod N$ and returns the result to Bob.
   3. Bob accepts iff the returned message is $r \bmod N$.

   Note that if Bob accepts, then Bob should be convinced that Alice knows how to take eth roots—from which it seems reasonable to expect that Alice must know d.

   (a) Prove that the protocol is zero-knowledge under the assumption that Bob is honest. In other words, under the assumption that Bob is honest, prove that there is a simulator that outputs fake transcripts whose distribution is exactly the same as the distribution of the real transcripts obtained each time Alice and Bob interact.

   (b) Argue that the protocol is not zero-knowledge if Bob is not honest. What kind of information can a dishonest Bob learn that he could not have discovered on his own? Give an example of a value that a dishonest Bob could learn by interacting with Alice, that he could not have computed on his own.

3. **(75 pts.) Exploiting buffer overflows**
   The purpose of this question is to learn how to exploit buffer overruns. We have provided two exploitable programs, and your goal is to write an exploit for each one that results in shell access.

The first thing to do is read Aleph One's "Smashing The Stack For Fun And Profit" (`http://www.shmoo.com/phrack/Phrack49/p49-14`).

We have provided two exploitable programs (`target1` and `target2`). Your goal is to write an exploit for each one that results in a shell on the system. In particular, you will write a program (`exploit1` and `exploit2`) which sets up arguments and environment variables that trigger exploitation of the buffer overrun vulnerability in the corresponding target when it is executed.

If the targets were installed setuid-root, then running your exploit program should result in a root shell. For obvious reasons, we have not installed the targets as setuid-root. Therefore, running your exploit program should give a shell prompt ("`$ `"), like this:

```
po% ./exploit1
$
```

You will know you have succeeded when you get the `$` shell prompt.

The source code and the executable programs are available in `/home/ff/cs161/hw3-f05/` on instructional machines. We have also provided templates for the exploit programs. You should modify `exploit1.c` and `exploit2.c` with your exploit code. We have provided shellcode for Solaris x86 machines in `shellcode.h`. The shellcode is the malicious code you are trying to inject into the target; when it is run, it will execute `/bin/sh`. Therefore, your goal is to try to inject this malicious code into the target and then cause the target to start executing that malicious code.

You should turn in the files `Makefile`, `exploit1.c`, `exploit2.c`, and `shellcode.h`. Make sure that they running `gmake` will compile all of them successfully, and that executing each exploit program will successfully exploit the corresponding target program. You may optionally provide a README file with comments about this assignment or feedback about how to improve the assignment, if you like.

You will need to do all of your coding and experiments on an x86 machine running the same operating system, compiler, etc., as we use to grade your submissions. Therefore, we require that your solution must work on `po.eecs.berkeley.edu`. Make sure that you do all your work on that machine, as exploiting buffer overruns is highly platform-dependent. We will grade your programs on `po.eecs`, so your exploit programs had better work reliably on that machine.

You will find it helpful to run the code with `gdb` and single-step through the program. Some especially useful commands: `break` (to set breakpoints; try `break foo` to cause `gdb` to stop when execution reaches the function `foo()`), `x` (to examine memory; try `x/24w buf` to see 24 words of memory starting at the address of `buf`), `info frame` (to see some information about the current stack frame), `next` (to execute one line of code and then stop), `cont` (to continue execution), and `disassemble`.

A good starting point would be to inspect the stack frame of the target code that you are trying to exploit. Understand as much of it as you can. Draw a picture.

A tip for making `gdb` work with `execve()`: `gdb` will trap when the `execve()` command executes. At that point, you can tell `gdb` about the symbol table for the new program with the command `symbol-file newprog` (assuming that `newprog` is the program that has been executed). Then you can set breakpoints in the new program, continue execution with the `continue` command, and so on.

You will probably want to use a different exploit technique than that shown in the Aleph One tutorial. Aleph One's tutorial assumes a Linux x86 machine; you will use a Solaris x86 machine, so a few details are different. For instance, the shellcode is different. Also, I found it much easier to place the shellcode on the heap rather than on the stack (and then overwrite the return address to point to the shellcode), though either strategy can be made to work. You will need to figure out the address of your shellcode, and this is where judicious use of `gdb` can be very helpful. The second half of Aleph One's tutorial mainly deals with how to construct shellcode; since we are giving you shellcode, you can ignore that part of the tutorial.

Caution: Aleph One's code calculates addresses on the target program's stack by looking at addresses on the exploit program's stack, and blindly assuming that the two will be the same. This is fragile and we definitely do not recommend it. Addresses may vary according to how the exploit program is executed (e.g., which directory, arguments, environment, etc. are used to start the exploit program). Since we may execute the exploit program is executed with a different context than when you wrote it, this means that Aleph's One technique for calculating stack addresses will not work. You should hardcode addresses for the target program's stack addresses in your exploit programs, so that your exploit will still work when we execute it. (Don't use `get_sp()` or its equivalents.)

In our experience, it was not necessary to create a NOP slide or use any other fancy techniques.