Cyber Security Worm Lab

Due

Thursday, April 1, 2010

Goal

Explore worm building techniques.

Things you need to know

You should work independently on this assignment. You will be given code for a trivial Linux server and client (magic8d and magic8 on the class web site). Magic8d listens on TCP/10001 and responds to connections with a very simplistic "magic eightball" response. Magic8d also has a very obvious buffer overflow error.

You must make sure that the linux machines have stack randomization turned off. The ones you need to change on a FC12 system are:

```
/proc/sys/kernel/exec-shield
/proc/sys/kernel/randomize_va_space
```

They both default to 1, but you won't get any buffer overflows to work unless you set them to 0. As with the IP-forwarding, you can add lines to /etc/sysctl.conf like:

```
kernel.exec-shield = 0
kernel.randomize_va_space = 0
```

You can use the shellcode used in testsc2.c of the phrack writeup to exercise the overflow by creating a root shell on the server machine assuming the server program is running as root (either directly or by setting the program setuid root). The phrack shell code is already typed into the eggshell and testsc programs that come with this assignment.

You are provided an eggshell program to help you figure out the offset to create the appropriate return value in your attack packet. The usage of the eggshell program is:

eggshell <buff size> <offset> [-f <out file>]

Where *buff size* is the size of the resulting poison packet. *Offset* is the guess of the offset from the eggshell program's stack to the target buffer address. If *out file* is not specified, the eggshell program returns the attack packet in the $EGG environment variable in the returned shell. You can use that value directly in the magic8 program to try the packet, e.g., magic8 192.168.1.107 "$EGG". If out file is specified, the poison packet is placed in that file, and you can cat files to test it in the client program, e.g. Magic 192.168.1.107 `cat egg.file` (notice we're using back quotes here). I'd suggest using the output file form. Creating extra shells can be confusing.

In either case, you can look at the output of your poison packet by redirecting it through “od”, e.g. echo $EGG | od -x or cat egg.file | od -X.

Once you get a root shell to appear on the server machine, you will want to tie the stdin, stdout, stderr of the shell to the client's TCP socket. The shelldupasm.c code augments the basic shell

attack with the dup logic. It guesses that the socket was the last generated file descriptor. It dup's to create a new fd, and assumes that the socket is that new file descriptor minus one. Compile the shelldupasm.c code with the "-static" argument. Then use "objdump -d shelldupasm" to extract the shellcode. You may also use Steve Hanna's odfhex.c program from http://vividmachines.com/shellcode/shellcode.html.

At this point you will change the default magic8 (the client program). Since the shell is non-interactive, you will get no response, so your client will need to send a command before waiting for responses (via the select and read). Have the client sleep for a couple seconds before sending the command, so the command does not get absorbed into the attack request. Be sure to terminate your commands with the newline characters, so the spawned shell will wake up and process your commands.

In previous years a number of students used metasploit to create the poison packet, and they used the netcat utility (nc) to deliver the packet. This approach will also result in a satisfactory solution.

Regardless of the method, create an attack client that:

- Checks for previous infection
- Fetches password files
- Installs a place marker to prove infection, e.g. something like kilroy was here.
- Downloads itself
- Launches attacks on other machines in the lab. Since we are constraining our attack to the lab, you can use a very simplistic target address function.

Use a unique name for your files, so the attacks do not collide.

Downloading the attack binary can be tricky. You could just fetch the binary from a fixed location as the first versions of the LION worm did, but it would be harder to track and more resilient if your attack binary could pull the binary from the source of the attack. In this case, the attacking socket is the only guaranteed source of communication between the two machines. There are a number of options in the shell to pass bulk information via standard input. Knowing when to stop reading standard in is the problem. Here documents can help solve that problem (http://en.wikipedia.org/wiki/Heredoc).

Hand-in Items

- Describe your worm design. How did you end up propagating the attack files? How does your worm propagate itself? What strategy would you use to really find potential victim addresses.
- Source code for attack client and any other relevant scripts.
- A file containing the poison packet. Either in binary form that I can run od over or already coverted to ascii.
- If you didn't have source code access to the magic8d server, how would you have gone about figuring out the return address for your poison packet?