Chapter 3

Reinforcement

Problem R-3.3

If a process forks two processes and these each fork two processes, how many processes are in this part of the process tree?

Solution There are seven processes in this part of the process tree: the root, its two children, and their four children.

Problem R-3.12

If a password is salted with a 24-bit random number, how big is the dictionary attack search space for a 200,000 word dictionary?

Solution The search space size is $2^{24} \times 200000$.

Problem R-3.13

Eve has just discovered and decrypted the file that associates each userid with its 32-bit random salt value, and she has also discovered and decrypted the password file, which contains the salted-and-hashed passwords for the 100 people in her building. If she has a dictionary of 500,000 words and she is confident all 100 people have passwords from this dictionary, what is the size of her search space for performing a dictionary attack on their passwords?

Solution The size of the search space is $100 \times 500,000$, which is 50 million. This is due to the fact that the attacker needs to do the dictionary attack against each person separately, because of the password salt.

Creativity

Problem C-3.1

Bob thinks that generating and storing a random salt value for each userid is a waste. Instead, he is proposing that his system administrators use a cryptographic hash of the userid as its salt. Describe whether this choice impacts the security of salted passwords and include an analysis of the respective search space sizes.

Solution Note that in traditional salt, a password, p, for a user, u, is stored as $h(p||s)$, where s is the random number associated with u. Under this plan, the salted password would be stored as $h(p||h(u))$. Assuming that an attacker knows the userid in this scenario, then he immediately knows the salt. Thus, this scheme makes the attacker's life easier, since he would no longer need to try all possible salt values (or perform an additional intrusion to capture the password file containing the salt values for each user. In particular, an attacker who only knows userid's now has a search space equal to the dictionary size for each individual user, instead of a search space of the dictionary size times 2^b, where b is the number of bits.

Problem C-3.2

Alice has a picture-based password system, where she has each user pick a set of their 20 favorite pictures, say, of cats, dogs, cars, etc. To login, a user is shown a series of pictures in pairs—one on the left and one on the right. In each pair, the user has to pick the one that is in his set of favorites. If the user picks the correct 20 out of the 40 he is shown (as 20 pairs), then the system logs him in. Analyze the security of this system, including the size of the search space. Is it more secure than a standard password system?

Solution There are two pictures for each choice, one on the left and one on the right. Thus, with 20 pairs, there is a search space of 2^{20}, which is roughly 1,000,000. This is roughly as secure than a standard password, for which there are dictionaries with 500,000 passwords that can be used in dictionary attacks. If the number of picture pairs is increased to 40, however, then the security is much higher. Of course, even picking one of 20 picture pairs requires 20 mouse clicks, which would take longer than typing in a traditional password.

Problem C-3.3

Charlie likes Alice's picture-password system of the previous exercise, but he has changed the login so that it just shows the user 40 different pictures in random order and they have to indicate which 20 of these are from their set of favorites. Is this an improvement over Alice's system? Why or why not?

Solution Note that there are two pictures for each choice in the original plan, one on the left and one on the right. Thus, with 20 pairs, there is a search space of 2^{20}, which is roughly equal to 1 million. If the search space is defined by picking 20 out of 40, then the size of the search space is the combinatorial function "40 choose 20," which is $40!/(20!)^2$, i.e., it is over 137 trillion.

Problem C-3.5

On Unix systems, a convenient way of packaging a collection of files is a ***SHell ARchive***, or ***shar file***. A shar file is a shell script that will unpack itself into the appropriate files and directories. Shar files are created by the shar command. The implementation of the shar command in a legacy version of the HP-UX operating system created a temporary file with an easily predictable filename in directory /tmp. This temporary file is an intermediate file that is created by shar for storing temporary contents during its execution. Also, if a file with this name already exists, then shar opens the file and overwrites it with temporary contents. If directory /tmp allows anyone to write to it, a vulnerability exists. An attacker can exploit such a vulnerability to overwrite a victim's file. (1) What knowledge about shar should the attacker have? (2) Describe the command that the attacker issues in order to have shar overwrite an arbitrary file of a victim. Hint: the command is issued before shar is executed. (3) Suggest a simple fix to the shar utility to prevent the attack. Note that this is *not* a setuid question.

Solution (1) The attacker should know the fact that the victim is about to unpack a shar file and the temporary file name, f, to be used by shar. (2) The attacker creates a symbolic link with name f in /tmp that points to the personal file, p, of the victim.

When the victim unpacks a shar file, shar finds that the temporary file f already exists and overwrites it with temporary contents. Since f is a symbolic link to file p, the original content of p is lost. Note that the attacker causes the victim to write to file p. (3) Instead of a predictable temporary file name, use a randomly-generated file name that is difficult to guess. Alternatively, generate a sequence of candidate names for the temporary file until a name is found that does not correspond to an existing file in /tmp.

Problem C-3.7

Dr. Blahbah has implemented a system with an 8-bit random canary that is used to detect and prevent stack-based buffer overflow attacks. Describe an effective attack against Dr. Blahbah's system and analyze its likelihood of success.

Solution An 8-bit canary only provides 256 possible canary values, so an attacker can try all possibilities in a fairly short amount of time. As soon as one of them has succeeded, he has compromised the system, and each one has a pretty good, one out of 256 chance, of succeeding.

Problem C-3.8

Consider the following piece of C code:

```
int main(int argc, char *argv[])
{
char continue = 0;
char password[8];
strcpy(password, argv[1]);
if (strcmp(password, "CS166")==0)
continue = 1;
if (continue)
{
*login();
}
}
```

In the above code, *login() is a pointer to the function login() (In C, one can declare pointers to functions which means that the call to the function is actually a memory address that indicates where the executable code of the function lies). (1) Is this code vulnerable to a buffer-overflow attack with reference to the variables password[] and continue? If yes, describe how an attacker can achieve this and give an ideal ordering of the memory cells (assume that the memory addresses increase from left to right) that correspond the variables password[] and continue of the code so that this attack can be avoided. (2) To fix the problem, a security expert suggests to remove the variable continue and simply use the comparison for login. Does this fix the vulnerability? What kind of new buffer overflow attack can be achieved in a multiuser system where the login() function is shared by a lot of users (both malicious and nonmalicious) and many users can try to log in at the same time? Assume for this question only (regardless of real systems' behavior) that the pointer is on the stack rather than in the data segment, or a shared memory segment. (3) What is

the existing vulnerability when login() is not a pointer to the function code but terminates with a return() command? Note that the function strcpy does not check an array's length.

Solution (1) The attacker gives a password longer than 8 characters and overflows buffer password[]. Thus, if variable continue is stored in memory after variable password[], the attacker can overwrite variable continue with value 1 and login without knowing the password. To defend against this attack, variable continue should be stored before variable password[]. In this way, overflowing buffer password[] will not affect the value of continue. (2) This does not fix the problem since the address that points to ∗login() can be overwritten by a buffer overflow and hence point to malicious code. After this exploit, a non-malicious user will correctly login but he will not execute the real login code, but the malicious code. (3) The return() address can be overwritten with similar results as before.