CS 155 Spring 2018 Homework 1

SUNet ID: 05794739
Name: Luis Perez
Late Days: 0

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Problem 1

Elizabeth can make the vulnerable application write the value 0x2222 to memory address 0x8888 by setting the values as specified in the tables below:

Table 1: Registers

ecx	0x3000
edx	0x9000
eip	0x4000

Table 2: Stack

0x9000	
0x9004	0x2222
0x9008	0x5000
0x900c	0x4000
0x9010	0x8888
0x9014	0x6000

Problem 2

The proposal is not safe. If GDB will simply run any program that calls setuid(0) with root privileges, then we have a relatively direct privilege escalation attack. Essentially, simply write an attack program (that, for example, encrypts the entire hard-drive). Normally, such a program would not succeed unless it is run as root. However, with the proposed modification, we can simply include 'setuid(0)' at the beginning of our program, and then execute the program using GDB, which will cause the code to run as root. Even worse, we don't even have to write the attack program ourselves. We simply need to find *some* program which we know calls 'setuid(0)' somewhere it its execution, and run it under the debugger GDB with a breakpoint. At this point, the program will be running with root privilege, and using GDB, we can directly manipulate the stack/heap/executable code and modify the EIP register. Essentially, with the tools GDB provices, we can make this root program do anything we want.

Problem 3

The lines were added to prevent an integer overflow vulnerability present in the original code (ie, since we add a non-zero value to a size_t and store the result in a size_t, we can overflow the size_t where the resulting sum is truncated and is actually less than the intended value). A possible attack would be one that causes a segmentation fault. If we pass a large enough value to _MALLOC, instead of returning NULL (to inform the program that the allocation failed), it will instead allocate some relatively small amount of bytes and return a pointer. However, the caller will have expected a large number of bytes to have been allocated, so it will very likely attempt to overwrite memory it does not actually own, causing a Segmentation Fault and possibly leading to a DoS attack.

Problem 4

(a) The new forked process has the same euid, ruid, and suid as the parent process.

(b) (a) In this case, the process can continue to run with euid n or run with its suid/ruid of m. There are no other possible options.

(b) In this case, the process can run with any value of euid it wants.

(c) By assigning seperate uids, we can restrict the resources each Android application can access (based on its distinct uid). If we use this in conjunction with the Principle of Least Privilege, then each Android application will have access only to the resources it needs to run. Generally, this means that if an application is vulnerable to attack and successfully hijacked, the attacker will only have access to limited system resources, thereby preventing a full system takeover.

(d) By the response in (1), we must call setuid after forking in order to decrease the privilage of the forked process (otherwise it would run as root!). This follows the Principle of Least Privilage and serves the purpose of isolating the system resources from vulnerable sub-processes. Only the zygote process runs as root - all other run as other users with restricted resources.

(e) (a) The setuid bit should be set on the passwd program (so that the euid is set to the user ID of the owner of the file, which should be root). By having this set, the passwd program can thereby run as root (even though the user calling the program is not necessarily root), and can change the Unix password file.

(b) This makes it *extremely* important write the source code of the passwd program carefully. The program escalates privilage (from non-root to root), and therefore is a target for attack. If the source code is not written carefully and a vulnerability is discovered, it is likely that the vulnerability can be used to gain root privileges to a system from an unprivileged account, thereby gaining full control of the system.

Problem 5

(a) Consider an attacker that is aware of this program running on the machine. While the program sleeps, the attacker can create a symbolic link linking "./file.dat" to "/etc/passwd" or some other system file. When the program writes "Hello world!", it will then also write it into the linked location. This is a very obvious vulnerability.

(b) The problem can still occurs since the OS is constantly context-switching between processes. So while the sleep(10) makes the problem easier to exploit, it is still possible to exploit without it.

(c) We would need to make the exists check and the file open/write atomic operations. This would require using OS-level features, and is typically quite difficult. Hoever, we would essentially need to lock access to the file handle before we perform the existence check, and hold this lock until after we finish our write.