CS458 - Assignment 1

Name: Lawson Fulton
UserId: ljfulton
Student#: 20381453

Written Answers

1. i) The fingerprint-based access control system is protecting access to a private resource behind a locked door. In this case, an interception attack could be achieved by simply following an authorized party through the door without the attacker scanning his or her own finger. This assumes that the system does not implement any measures to ensure that only one person enters per scan.
An interruption attack could be as simple as vandalizing the fingerprint scanners. This could be achieved by covering the scanning surface with a substance like super glue which would be extremely difficult to remove without replacing the component. The access system would likely be interrupted until a repair is possible.
Assuming that an attacker could gain access to the central server, a modification attack could be made by modifying the database to escalate the privileges of someone registered in the system already. This could happen if for example the central server also hosts the company's website which was compromised to give the attacker root access.
Most fingerprint scanners have some degree of error which allows for fabrication attacks. This can be achieved by making a replica fingerprint of an authorized user.

ii) The badge-based access control system is also vulnerable to interception attacks in the same way that the fingerprint-based system is vulnerable. However, there is the additional opportunity to launch an attack by stealing the card of an authorized user. This could happen if the authorized user was careless enough to leave their badge lying out in the open in a public place.
Similar to the fingerprint scanner, the barcode reader could be vandalized for an interruption attack.
The same approach for a modification attack would also be possible.
In terms of fabrication attacks, the barcode system is even easier to spoof than the fingerprint system. All that would be required is a photocopy or picture of an authorized user's badge to gain access. Assuming a typical user wears their badge on their waist, an attacker could approach a user in public while they were wearing their badge and take a picture with their smartphone without anyone noticing.

iii) Based on the previous discussion, it is clear that the fingerprint based system is more secure. This is because the barcode system is vulnerable to all of the same attacks as the fingerprint system with the additional fact that it is easier to fake or steal a barcode badge than it is to fake a fingerprint or steal a finger.

2. i) When Alice is stopped by a police officer, the officer identifies her by looking at her name on her driver's licence. The first step to authenticating the identity is to ensure that she looks like the picture on the card. Furthermore, the police officer checks on his cruiser computer that the card Alice provided is registered and all the information matches. This ensures that the card is not a fake. After this step the officer can be confident that Alice is who she says she is.

ii) The Android Smart Lock feature has three different methods of authentication. The first method is location based. The feature first identifies Alice by periodically checking the current GPS coordinates of the phone and contacting Google Maps to get the location. The location is then compared against a pre-defined set of 'Trusted Places'. If the current location of the phone is within one of the trusted places, Alice is authenticated and the phone automatically unlocks.

The next feature is based off of proximity to known devices. This feature identifies Alice by periodically checking for devices within range of the bluetooth and NFC sensors. If a device is found, it is compared against a pre-chosen set of 'Trusted Devices'. If the device is in the set of trusted devices, then Alice is authenticated and the phone unlocks.

The third and final feature is the facial recognition lock. The identification step takes place when Alice presses the unlock button of the phone and the front camera takes a picture of her face. To authenticate Alice the face is then compared to a pre-chosen set of 'Trusted Faces'. If the face is determined to be sufficiently similar to one of the trusted faces, Alice is authenticated and the phone is unlocked.

sploit3.c - Buffer Overflow

This is a buffer overflow exploit that takes advantage of an assumption in the backup program that files will never be larger than 3072 bytes.

The vulnerability is from the buffer on line 25 in the copyFile function of backup.c. The buffer is specified to be a fixed size of 3072 bytes. The function fills the buffer by opening the source file and reading its contents into the buffer one byte at a time until EOF. No bounds checking is performed on the buffer, so if the file is larger than 3072 bytes, the contents of the file will be written into adjacent memory on the stack. This behaviour can be exploited to overwrite the return address of the function. However, there is one complicating factor, there is a counter variable that is used to index the buffer that is stored between the buffer and the return address. It is important to avoid corrupting the counter while over-flowing the buffer.

The goal of sploit3.c is to exploit the backup program by asking it to back up a file a little larger than 3072 bytes that contains a shellcode and suffixed with repeating modified return addresses. It works as follows: First a file named 'foo' is created on the disk in the current directory. Then NOP instructions are used to fill the entire file saving enough room to put the shellcode right at the very end of the 3072 byte range. Then the 4 byte integer 3072 is written to the file in order not to corrupt the counter when it gets to the end of the buffer and it is overwritten. Finally, the address 0xffbfd1f0 is repeated ten times at the end of the file. This address, which was determined through manual inspection with gdb to point at the beginning of the buffer, will overwrite the original return address.

When the sploit is run, a shell with root privileges should be launched immediately.

This exploit could be fixed by reusing the same buffer if the file is larger than 3072 bytes. What I mean by this is that once the buffer is full, the 3072 bytes should be flushed to disk, and the counter set back to 0, and repeat the process until the entire file has been copied. This way, the counter would never be able to index out side of the array. The corrected code is shown below.

```
#define BUFSIZE 3072
unsigned int len, i;
char buffer[BUFSIZE]; /* 3K is not enough for anyone*/
FILE *source, *dest;
int c;

source = fopen(src, “r");
if (source == NULL) {
  fprintf(stderr, "Failed to open source file\n");
  return -1;
}
dest = fopen(dst, "w");
if (dest == NULL) {
  fprintf(stderr, "Failed to open destination file\n");
  return -1;
}

i = 0;
c = fgetc(source);
while (c != EOF) {
  buffer[i % BUFSIZE] = (unsigned char) c; //mod for extra safety
  c = fgetc(source);
  i++;
  if(i == BUFSIZE) {
    fwrite(buffer, 1, i, dest);
    i = 0;
  }
}
fwrite(buffer, 1, i, dest);
```

sploit4.c

This is a format string exploit that takes advantage of a buffer overflow that can occur when sprintf is called in backup.c:133

If the backup program does not receive any commands, it calls the usage() function that is supposed to print out the correct usage of the program. However, the function assumes that the value passed into args[0] will always be the name of the executable. This is not always the case, as is demonstrated in sploit4.c where the args[0] is set to a custom exploit string that uses the sprintf to cause a buffer overflow.

When sprintf(output, buffer) is called, it starts iterating over the buffer one character at a time copying the results into output. However, when the special format characters %44d are encountered, 44 bytes of output are filled with whitespace and the end of the range is filled with data from the sprintf stack frame interpreted as an integer. This itself is innocuous enough, but now as sprintf continues copying data out of buffer, output will overflow by 44 bytes, since both arrays are the same size. Buffer itself contains a shellcode immediately following the format string characters, followed by a repeating address of the shellcode in the buffer array. The result is that sprintf writes past the end out the output array, overwriting the return address of the

function with the address of the shellcode. So when the function returns we start a new shell as root.

The fix for this vulnerability is very simple. All we would need to do is replace sprintf(output, buffer) with sprintf(output, “%s”, buffer) to prevent any special format characters in buffer from being expanded.