1. Row hammer attacks are monstrously technical. They include deliberately executing a program again and again on a row of transistors in a PC's memory chip. The thought is to "hammer" that row, until it releases some power into the contiguous line. That spillage can cause a piece in the objective line to "flip" starting with one position then onto the next, somewhat modifying the information put away in memory. A talented Row hammer attacker would then be able to begin to misuse these minor information changes to acquire more system access.

   Meltdown and Specter attacks basic vulnerabilities in present day processors. These equipment vulnerabilities permit projects to take information/data which is being prepared on the PC. While programs are regularly not allowed to use information/data from different projects, a noxious program can abuse Meltdown and Specter to get hold of insider facts put away in the memory of other running projects. This may incorporate your passwords put away in a secret key administrator or program, your own photographs, messages, texts and even business-basic archives. Meltdown and Specter take a shot at PCs, cell phones, and in the cloud. Contingent upon the cloud supplier's foundation, it may be conceivable to take information from different clients. The worst part is that it doesn't leave any traces in the log files, so it is hard to prevent this type of attack

2. **In this example:**

```
int main(int argc, char *argv[]) {
        unsigned short s; int i; char buf[80];
        if (argc < 3){ return -1; }
        i = atoi(argv[1]); s = i;
        if(s >= 80) {  printf("No you don't!\n"); return -1; }
        printf("s = %d\n", s);
        memcpy(buf, argv[2], i);
        buf[i] = '\0'; printf("%s\n", buf); return 0;
}
```

   These will be the inputs,

   ./file 5 hello
   s = 5
   hello
   ./file 80 hello
   Oh no you don't!
   ./file 65536 hello
   s = 0
   Segmentation fault (core dumped)

   The length argument is taken from the command line and stored in the integer "i". At the point when this is moved into the short integer s, it is shortened if the value is too big to even consider fitting into s (for example on the off chance that the value is more prominent than 65535). So, it could happen that to bypass the limits check and overflow this buffer. After this, standard strategies can be utilized to misuse the procedure.

In this example:

```
const long MAX_LEN = 20K;
Char buf[MAX_LEN];
short len = strlen(input);
if (len < MAX_LEN) strcpy(buf, input);
```

Yes, the buffer overflow attack can occur. Here in the code, the variable MAX_LEN is being compared to the user inputted len. We also notice that, the reference is being copied in the Char buf object, which has the size of MAX_LEN. If the input for the variable "len" is bigger than MAX_LEN, the "strcpy" will not be able to successfully copy the reference to the Char buf, and hence buffer overflow will happen. In short, if the inputted value is bigger than MAX_LEN then the overflow will happen

In this example:

```
int ConcatBuffers(char *buf1, char *buf2, size_t len1, size_t len2) {
        char buf[0xFF];
        if ((len1 + len2) > 0xFF) return -1;
        memcpy(buf, buf1, len1);
        memcpy(buf+len1, buf2, len2);
        return 0;
}
```

We see in this example that memcpy function is used. This function, when the first time it is called, will copy from len1 bytes from buf1 to buf. The only way to have buffer overflow happen is when the size of buf1 is larger than buf. In the next line, len2 bytes are being copied from buf2 to buf+len1. Again, the only way to have buffer overflow happen is when the size of buf2 is larger than buf+len1

In this example:

```
bool IsValidAddition(unsigned short x, unsigned short y) {
        if (x+y < x)
                return false;
        return true;
}
```

This x and y are unsigned and unsigned variables do not overflow (at least in C). So yes, this program will work fine

3. For a simple printf statement like these,

```
int func(char *user) {
    fprintf( stdout, user);
}
```

Here, if variable “user” is “%s%s%s%s%s%s%s” then the format string attack will be performed, as it will print the memory contents, which can reveal privacy

```
#include <stdio.h>

 int maintint arge, char **argv)
 {
     int number = 5;

     printf(argv[l]);
     putchar('\n');
     printf{"you typed in: (%p)which is equal to %d\n", value);
}
```

Since we can control the arguments to a specific format string function, we can make random qualities be overwritten to determined locations with the utilization of the %n format string character. To really overwrite the estimation of pointers on the stack, we should indicate the location to be overwritten and use %n to keep in touch with that specific location. Now, in the first place, we realize that while conjuring the helpless program with a contention of the length of 10, the variable is situated at 0xbffffcl8 on the stack. We would now be able to endeavor to overwrite the variable number.

INPUT:

./test “printf “\xl8\xfc\xff\xbf”~%x%x%n

OUTPUT:

bffffc3840049fl840135e48
you typed in: (0xbffffcl8) which is equal to 10

4.

A) User in a Linux/UNIX system can change these configurations in order to attack,
In Linux, a setuid program has a system call: system(ls). The user can set his PATH to be . (current directory) and place the program “ls” in this directory. The user can now execute arbitrary code as the setuid program, which is a huge system flaw.

Another thing is an attacker can attack by resetting the IFS. IFS is the characters that the system considers as white space. If the IFS variable is not reset then the user may add “s” to the IFS and thus, system(ls) becomes system(l). It can then place a function “l” in the directory.

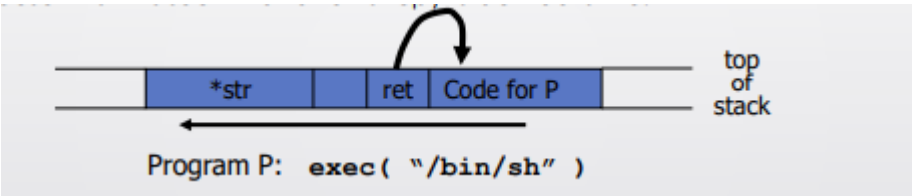
Another way of attacking modern OS is by resetting the “LD_LIBRARY_PATH” variable. Here is an example for that, assume you have a setuid program that loads dynamic libraries. We know that UNIX searches the environment variable LD_LIBRARY_PATH for libraries. Now, a user can set LD_LIBRARY_PATH to /tmp/bigAttack and places his own copy of the libraries here and exploit the system. Most modern C runtime libraries have fixed this by not using the LD_LIBRARY_PATH variable when the EUID is not the same as the RUID or the EGID is not the same as the RGID

B) In order to perform the buffer overflow attack, we simply will have to break the strcpy function by passing in a value greater than Char buf[128]. So, the first input will be to pass a string that is 140 bytes long. That way, it will result in exploiting the buffer and attack will be successful. Since, there is no range checking in the strcpy function, it is easier to break it. Here is an example,

INPUT: /execcmd 123456

OUTPUT: Segmentation fault (core dumped)

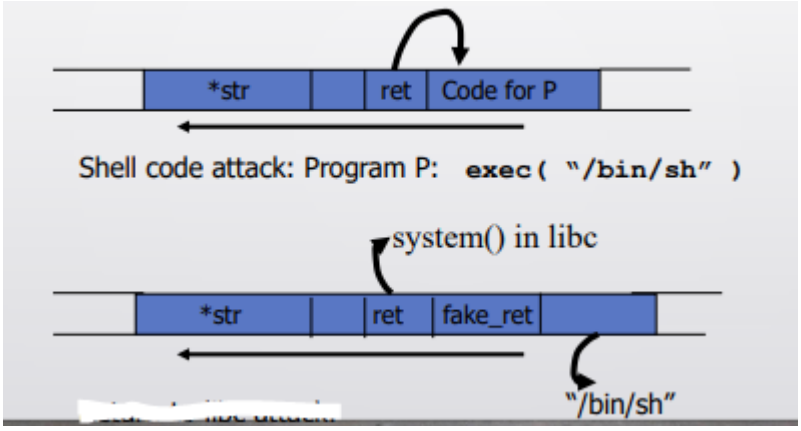
C) Here is what the stack will look like. Note that the code will run on the stack,



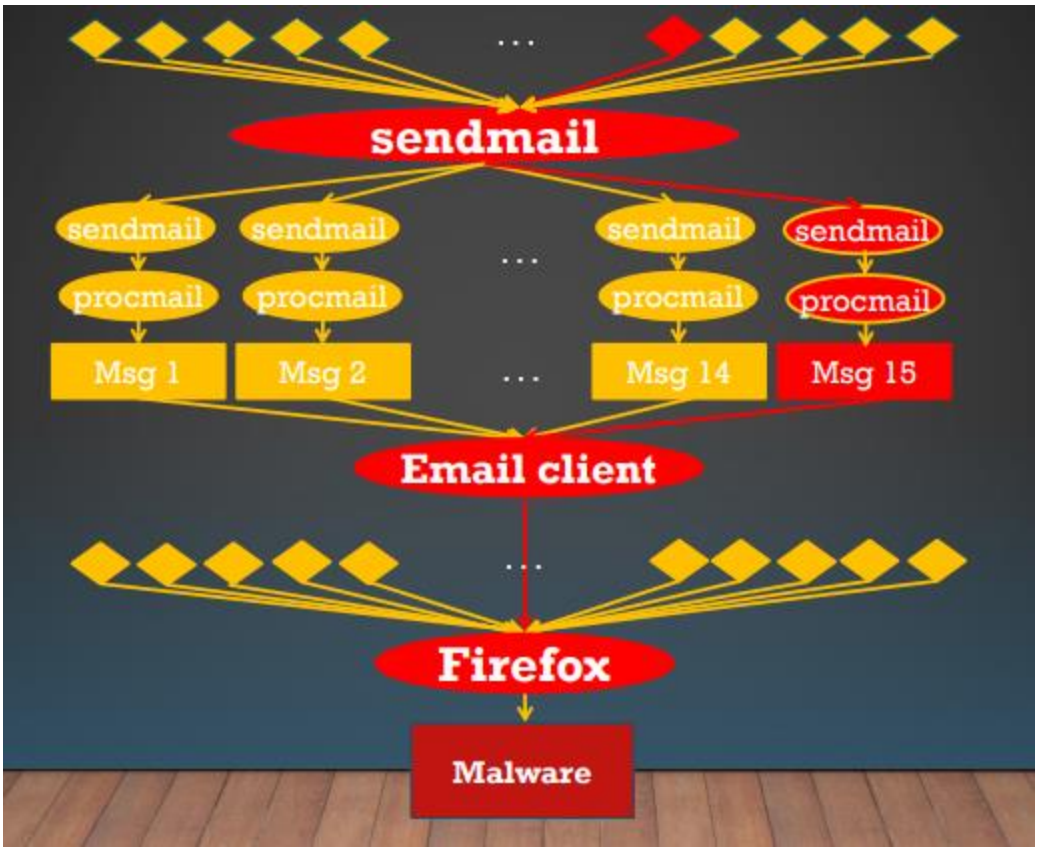
When the function exits, user will be given a shell, which is the main goal of this kind of attack.

D) Basically, all the overflow attacks are input oriented and can be done without injecting any code. Almost all instructions already exist in the process's address space, the configurations mentioned in part a will help to piece them together to do what the attacker wants

These configurations are important as they help the attacker to exploit the system to the fullest. The setuid configuration will help to bypass non executing stack during exploitation and the attacker can set the parameter in a way it gets a shell. It will also help in overwriting the return address to point to functions in libc. Here is an illustration on how this will help the attacker. Because of the mentioned configuration, attacker can follow through with the attack this way,



5. Here is an example of how defensive explosion attack works (from the slides)



By looking at the above image, we can see that the malware has been transmitted via the email service. The way it worked is, the network socket sent some malware and the “sendmail” process was affected by it. The “sendmail” process sent the mail to the user which was then accepted by the email client. Email clients are generally run by opening them up on a web browser (Firefox, in our case). Once the web browser gets the virus it can easily be transferred to the user’s computer. The problem with this attack is that it is really hard to trace because of there are many numbers of sockets that were open, many processes initiated the process of sending the mail, and hence, it is hard to find the root cause of it, and without it, it is almost impossible to trace and stop the malware.

One of the methods to stop this attack, is to reduce the number of operations that this attack is generating. Meaning, we need to find a way to stop the amount of process it is creating, amount of file it is using, number of tabs it is opening in a web browser etc. That way, we will know what exactly started the spread of the malware and eventually, we will be able to completely remove it or stop the spread of it, once we get to know its root cause. One of the methods is called BEEP (BINARY-BASED EXECUTION PARTITION), where we dynamically partition the execution of a process into autonomous execution segments, called Units, that are not always independent, which helps us detect causality between units.

It is also a good idea to have some high-level back tracking device, that can back track processes in a cluster-free manner. That way, despite of having multiple ports, ip addresses, browser tabs, we will know what exactly cause the issue.

Another method that I can think of is to alarm users if a large amount of processes is run for a relatively simple task. If the users are notified when any abnormal behavior is noticed, then they can maybe stop the process, and it may also help them figure out what caused the malware spread if the system is affected. Hence, it will help them back trace as well.

A fine and simplest way will be to close the process once it is done executing. Long running process is one of the easiest ways how this kind of attack is spread. Sometimes users let the process run for several hours and do not pay any attention to it and thus, if any malicious activity is noticed, they won't be aware of that.