1. Give the purpose of program stack and with an example discuss how it is used to keep track of program execution and state of the program

The memory space of the program is split into many sections. One of those is the program stack. The program stack in x86 is used to store temporary data for functions during the runtime. It is an area of memory that grows downward and acts as a last in first out data structure.

It can be used to store local variables, return addresses and other function specific information. It can also store program state and thus keep track of program execution.

When a function is called, some information relevant to that function is pushed onto the stack and when the function returns, this information is popped off. This is known as the function's stack frame.

This is how the stack is used to keep track of program execution. It handles the following

- Calling a function
- Space for local variables during function execution
- Returning from the function

Calling a Function

- The calling function first pushes the parameters needed for the call.
- Then the return address, which is the address of the next instruction in the calling function is pushed.
- Base Pointer(frame pointer) ebp is a register that points to the address on the stack where the stack frame of a function starts. Then the value of the ebp of the calling function is pushed onto the stack and then the ebp register is updated to hold the esp value(the called function's stack frame pointer).

Reserving Space for Local Variables

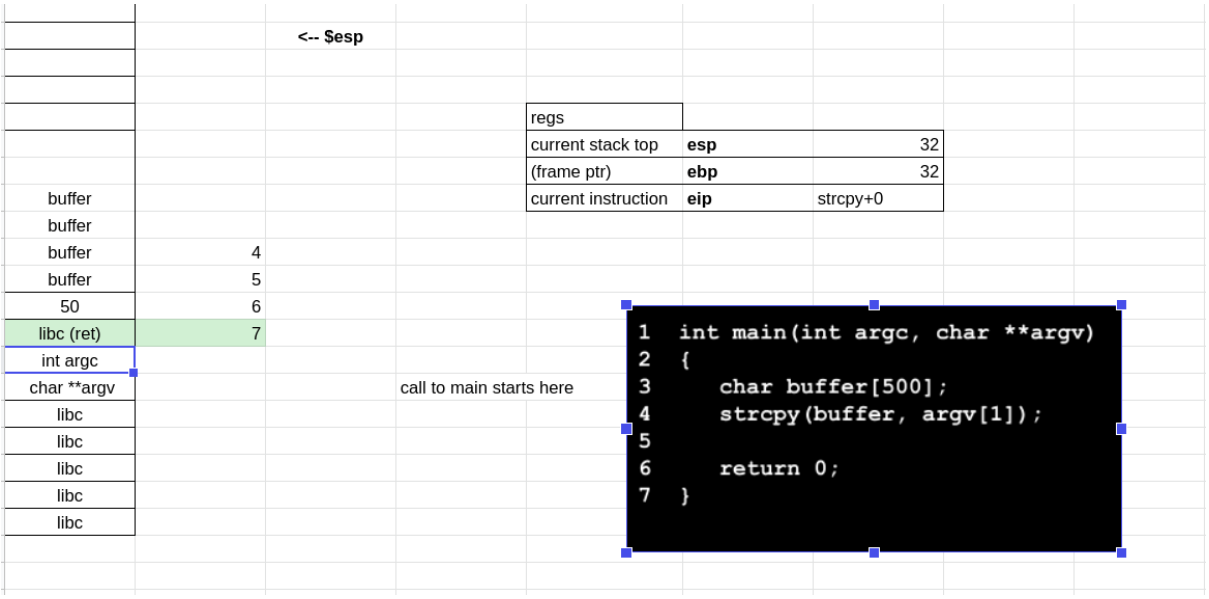
- The the stack top pointer is decreased by a certain amount to make space for local variables in the called function. It is decreased because the stack grows downward.

Returning from a function

- The stack frame is reset by setting the esp register to the value in the ebp register. That is, we move the stack top back to the start of the stack frame.
- Then the value of the saved ebp is put in the ebp register. Now the ebp register points to the stack frame of the calling function.

- Then the eip register is set to (4 + esp) which is the return address.

Explain the same with an example



2. Discuss the countermeasures against SQL injection attacks.

In SQL injection attacks, the problem is the treatment of code and data in the same way. Mechanisms should be introduced to treat data different from code.

Preventing SQL Injection

Input Validation

Validate the input before using it. **Check** the input to make sure it has the expected form. Only allow exact data. **Sanitise** the input by modifying it or using it such that the result formed by construction is of the expected form.

Sanitization

Blacklisting: Delete characters that we don't want. Like ;, –- or '. Except sometimes, these characters may be required too.

Escaping: Change problematic characters with safe ones. Add a backslash before these characters. \;, \', \\, \-. But, sometimes, these characters maybe needed in SQL. mysql_real_escape_string() and magic_quotes_gpc = On are examples.

Prepared-statements: Bind variables. Bind variables are typed. Treat data according to its type. Decouple code and data. Decoupling lets us compile before binding the data.

Example:

```
$db = new mysql("localhost", "user", "pass", "DB");
$stmt = $db.prepare("select * from users where(name=? and pass=?);");
$stmt->bind_param("ss", $user, $pass);
$stmt->execute();
```

Checking

Whitelisting : Check that the user input is safe. Maybe use regex. Uses the principle of fail-safe defaults. Safer to reject a invalid input than to fix it. Hard to do for rich-input. Need to cover a lot of edge cases.

Mitigation

Mitigate the effects of an attack. Should do input validation regardless.

Limit Privileges: reduces power of exploitation. Can limit commands and/or tables a user can access.

Encrypt Sensitive data. Less useful, even if stolen.

3. How an attacker constructs the needed functionality using return oriented programming to perform an attack and discuss the challenges involved in it?

The idea behind ROP is to string together pieces of existing code, called gadgets rather than use a single libc function to run shellcode.

Gadgets are instruction groups that end with ret that the attacker can chain together.
Here the idea is that the stack serves as the code.

- The %esp acts like the program counter
- Gadgets are invoked via the ret instruction
- Gadgets get their arguments through instructions like pop (data on the stack)

The challenges for the attacker include

- Finding the gadgets
- Stringing them together

Once the gadgets are found, the attacker uses other techniques(buffer overflow, format string) to change the data on the stack such that the return addresses point to the next gadget to execute. The data on the stack is also changed to serve as arguments for the next gadget.

To find the gadgets

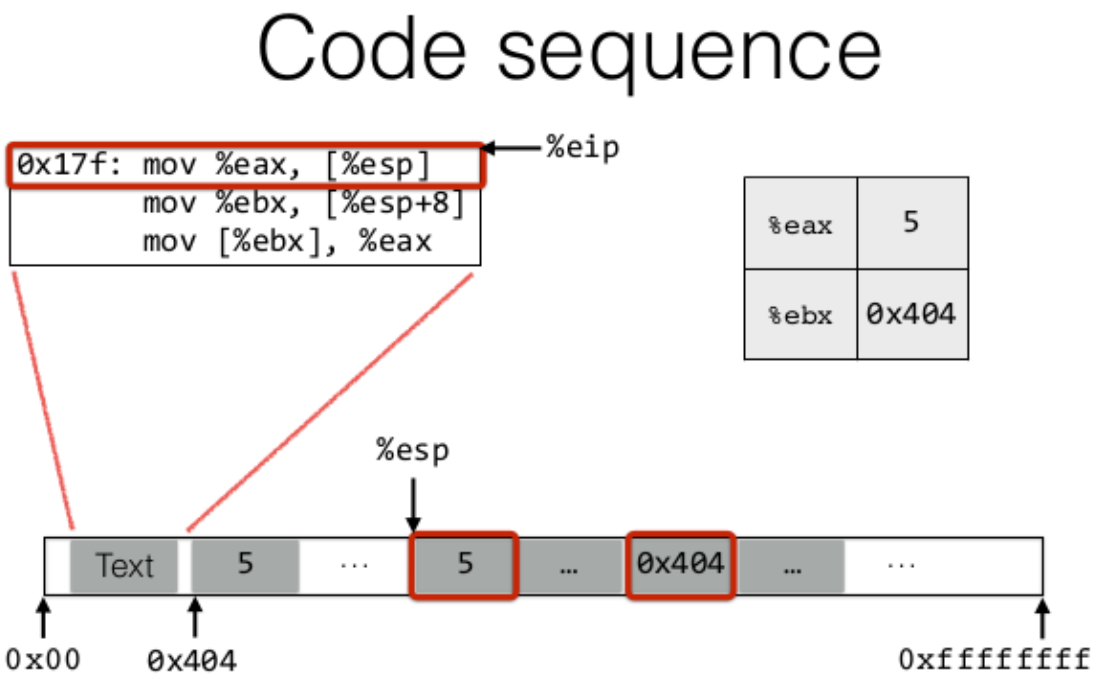
- We can automate a search over the target binary for gadgets. We look for all ret instructions and work backwards.
- It has been shown that there are sufficient gadgets because of x86s dense instruction set. Gadgets are turing complete.

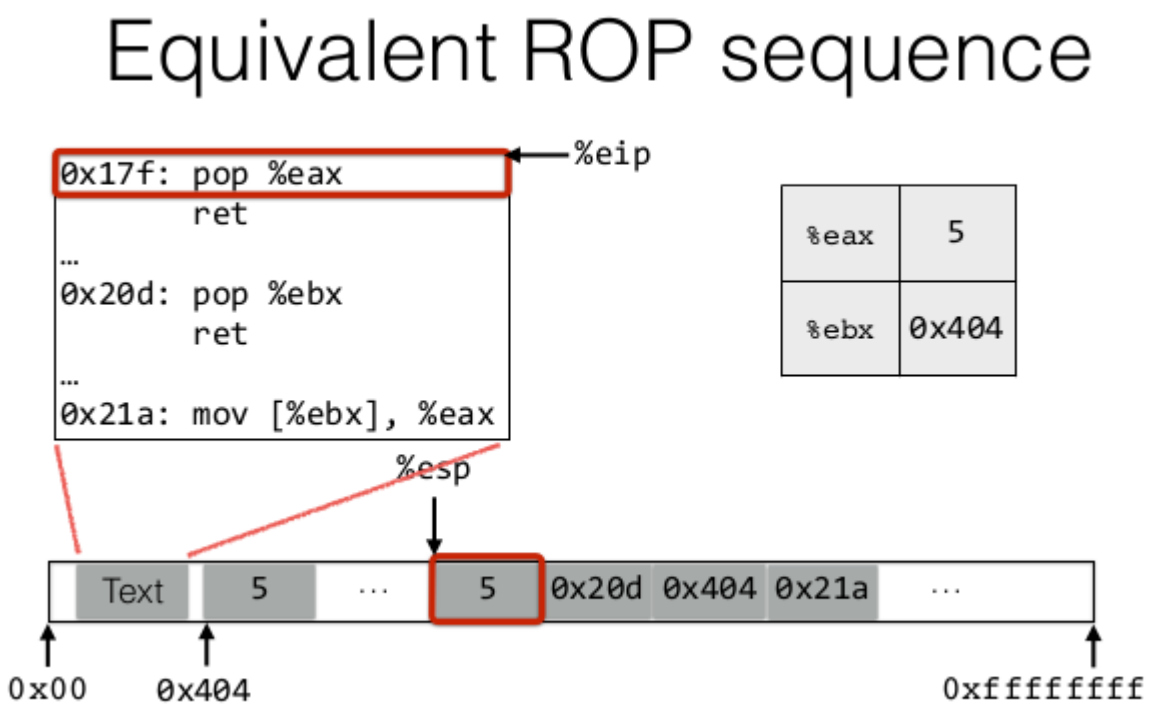
A defense against this technique is **Randomizing the location of the code**.
This is done by compiling as position-independent code. This makes attack very difficult on a 64-bit machine.

Attackers can get around this by using **Blind ROP**
If a server restarts on a crash, but does not re-randomize

- Read the stack to leak canaries and return address
- Find gadgets to effect a call to write
- Dump binary to find gadgets for shell code

If asked for example, use the image here





4. What are the challenges present during code injection and how is it feasible for an attacker to inject code?

The challenges present during code injection are:

Challenge 1: Loading code into memory

- The code should be machine code instructions - that is it should be compiled and ready to run.
- It shouldn't contain any all-zero bytes as scanf, sprintf, getf etc will stop copying. It's difficult to write assembly without containing an all-zero byte.
- It can't use the loader since we are injecting code.

Challenge 2: Getting injected code to run

- We can't insert a "jump into my code" instruction directly as we don't know precisely where our code is.
- Resolved by - hijacking the saved %eip

Challenge 3: Finding the return address

- If we don't have access to the code, we don't know how far the buffer is from the saved %ebp.
- Resolved by - address randomisation, nop sleds

Delivering a code payload that spawns a new shell to the attacker is one of the most common ways to perform a code injection attack. The first challenge is the code that launches a new shell, i.e the shellcode. The simplest way to spawn a new shell is to simply launch using a variant of the 'execve' system call.

Since we know that the injected code should be machine instructions, GDB tool can be used to convert the payload program into assembly code, and then into its corresponding machine code. Code injection is often done by entering the machine code as string input to the program, hence zeroed bytes must be avoided in the equivalent machine code as it would be taken as a string terminator, preventing the entire code from being stored in the target system's memory. Additionally, the assembly code created using the GDB tool will use hardcoded addresses which will not work on all machines. Hence, the assembly code must be modified to use relative addressing

By replacing the contents of the stack with the shellcode payload, the system executes the instructions corresponding to the execve function call, replacing the currently executing program with a standard shell, thus granting access to the system to the attacker.

5. Write a program that helps to by-pass the authentication check by performing a buffer overflow attack, explain in detail the changes that take place in the stack as each instruction gets executed

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

The Stack will look like this for func

```
Arg1 -> return address to main -> saved ebp -> authenticated(4bytes) -> buffer(4bytes) ->
```

Now if the input is larger than 4 bytes, it will overwrite the authenticated variables, memory region.
This makes it non zero, thus bypassing the authentication here

6. Discuss in detail how control flow integrity helps to overcome the ROP kind of attacks?

In behaviour-based detection, there are 3 main challenges. Control-flow Integrity has a set of techniques to overcome them

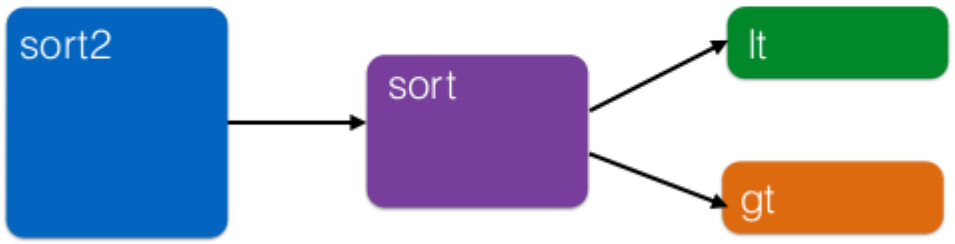
- Defining expected behaviour: Control-Flow Graphs
- Detect deviation from expectation effectively: In-line reference monitor
- Avoid compromise of detector: Randomness and immutability

For ROP, CFI works by ruling almost 95% of gadgets as non-compliant. Also rules out almost all indirect jumps.

A call graph shows which functions call which other functions.

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



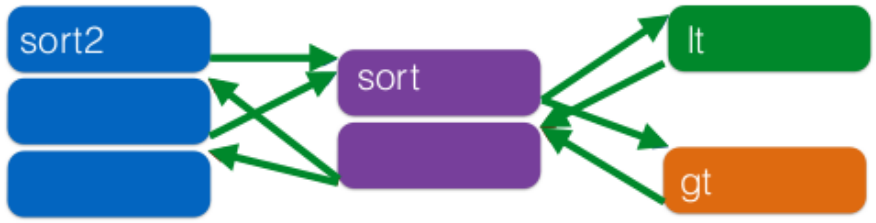
Which functions call other functions

A control flow graph distinguishes calls and returns in the programs. The program is also broken into basic blocks.

Control Flow Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



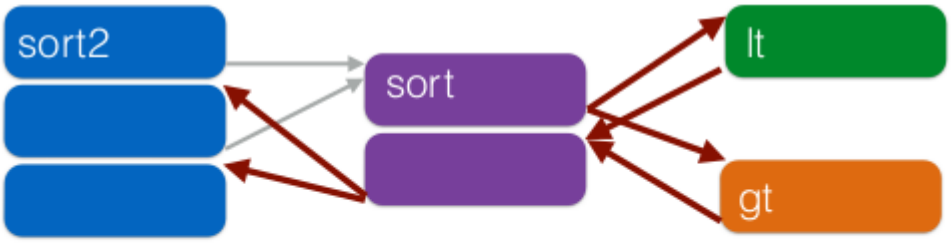
*Break into **basic blocks***
*Distinguish **calls** from **returns***

This control flow graph can be computed in advance, during compilation or from the binary. Then during runtime, the control flow of the program is monitored and we can ensure that it follows only paths allowed by the Control Flow Graph. This helps mitigate against ROP, by not allowing non-compliant paths to be taken.

DEP doesn't let us modify instructions. Therefore, only indirect calls need to be monitored, like jmp, call and ret.


```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

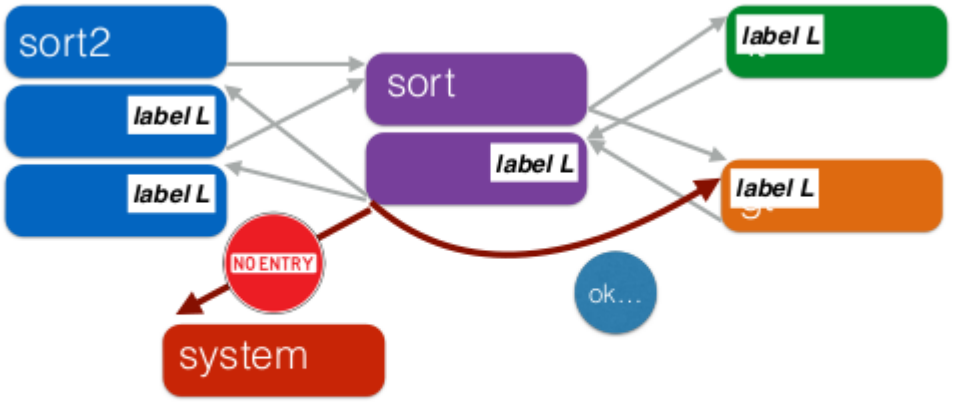
```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

Indirect transfer *(*`call` *via register, or* `ret`*)*

Calls to sort are direct. The address of the instruction is directly used to call it. But the calls to lt and gt, are indirect. They are passed as parameters on the stack. Similarly, all the returns are also indirect because they'll use the return addresses saved on the stack.

An in-line monitor is implemented as a program transformation. Insert a label determined by the CFG before the target address of every indirect transfer. Check the label of the target at each indirect transfer and abort if the label does not match. The simplest is to label every target with the same label, say L. But this won't prevent ROP, since returning to another target with the same label will be allowed.



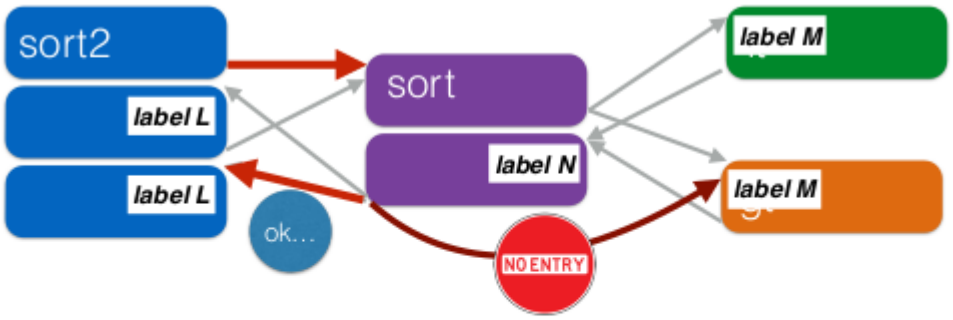
Use the same label at all targets

Blocks return to the start of direct-only call targets but not incorrect ones

To prevent this, more detailed labelling is used, say

- Returns sites from calls to sort, must have label L
- Call targets gt and lt much have label M
- N is an unconstrained label

This prevents ROP by only allowing jumps, rets from and to labelled targets, with the constraints enforced.



Constraints:

- return sites from calls to `sort` must share a label (*L*)
- call targets `gt` and `lt` must share a label (*M*)
- remaining label unconstrained (*N*)

Still permits call from site A to return to site B

7. Discuss the automatic defences against low level attacks

a. Stack Canaries
   i. Stack canaries are a secret value placed on the stack that changes every time a program is started. Before a function returns, the stack canary is checked and if it appears modified, the program exits immediately. They are a way to mitigate any stack smashing. Leaking the address of the canary and brute forcing are the only two methods that would allow attackers to get through the canary check.
   ii. When an attacker overflows a buffer, the stack canary would also get overwritten. When the function returns, this would get detected, thus preventing any injected code from being executed.

b. Stack Non-executable
   i. Buffer overflows often put some shellcode into the program's stack and then jump to it. If all writable addresses are non-executable then this is prevented. By setting the stack to be non-executable we can stop the attacker from putting shellcode there and then executing it.

c. ASLR
   i. This works by randomizing the address of the stack and the base address of memory areas. Typically, low level attacks work by overwriting a return address so that the function will return to an attacker-chosen address.
   ii. Address Space Layout Randomization works by making it difficult for the attacker to find an address to jump to.

8. How to create an executable code and place it over the stack?

If the attacker uses the execve system call to execute shellcode, the system will replace the currently running program with a newly initialized stack, heap and data sections. If a bash shell is launched, complete access to the system can be obtained.

To carry out an attack the code must be injected into the target system as machine code instructions. There will be no loader to help. GDB can be used to convert the payload into assembly code and then into its corresponding machine code. Code injection is done by entering the machine code (hex values) as string input to the program. Because of this, zeroed bytes cannot be present in the machine code string as functions like printf, scanf will consider it as the null character and terminate reading the input.
The assembly code will have hardcoded addresses. It must be modified to use relative addressing.

To place the shellcode over the stack, it must be given as an input string of hex values. The vulnerable program will place it on the stack in a local variable.

9. Global Offset Table

The Global Offset Table is a section inside programs that holds the addresses of functions that are dynamically linked. Common functions are “linked” into the program so that they can be saved once on the disk and reused by every program.

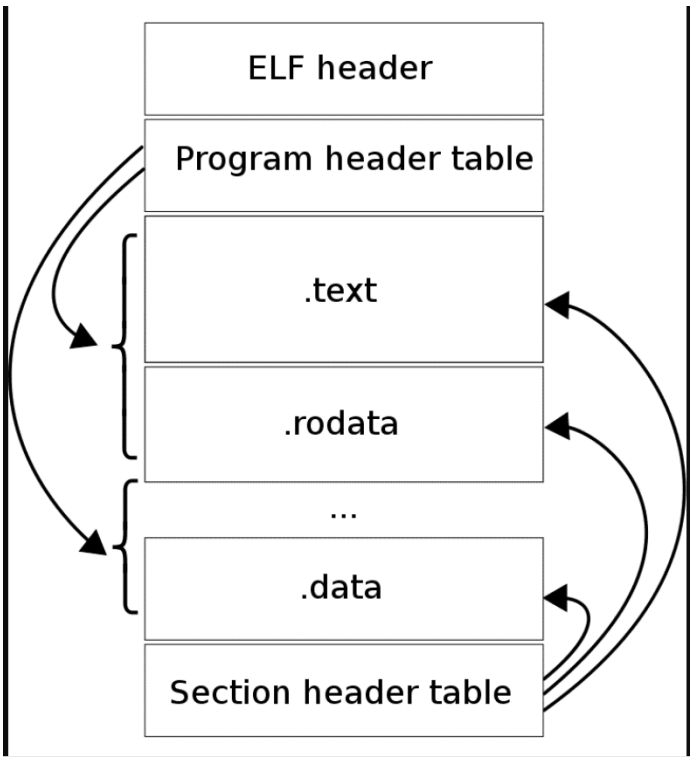
All dynamic libraries are loaded into memory along with the main program. Functions are not mapped to their actual code until they’re first called. To avoid searching through libraries each time a function is called, the result of the lookup is saved into the GOT so future calls to the function can just use the entry in the GOT to know the address of the function.

The GOT contains pointers to libraries whose addresses move around due to ASLR.
The GOT is also writable making it the target of many attacks.

Before a function’s address has been resolved, the GOT points to an entry in the Procedure Linkage Table. This is a small stub function which calls the dynamic linker with the name of the function that should be resolved. Once resolved, the entry in the GOT for the function is populated with the address of the function.

10. ELF executable format

ELF is the abbreviation for **Executable and Linkable Format** and defines the structure for binaries, libraries, and core files. The formal specification allows the operating system to interpret its underlying machine instructions correctly. ELF files are typically the output of a compiler or linker and are a binary format.



Each ELF file is made up of one ELF header, followed by file data. The file data can include:

- Program header table, describing zero or more segments
- Section header table, describing zero or more sections
- Data referred to by entries in the program header table, or the section header table

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation.

The ELF header is 32 bytes long, and identifies the format of the file.Among other values, the header also indicates whether it is an ELF file for 32 or 64-bit format

The program header shows the segments used at run-time, and tells the system how to create a process image. The primary data structure, a program header table, locates segment images within the file and contains other information necessary to create the memory image for the program.

.rodata - These sections hold read-only data that typically contribute to a non-writable segment in the process image.

.text - This section holds the ‘‘text,’’ or executable instructions, of a program.

.data - These sections hold initialized data that contribute to the program’s memory image

The third part of the ELF structure is the **section header.** It is meant to list the single sections of the binary.

11. Data Execution Prevention

Data Execution Prevention or DEP , flags certain areas of memory as non-executable or executable, which prevents an attack from running code in a non-executable region.

For example, in a buffer overflow + code injection attack. The shellcode is placed in a buffer on the stack and then the return address is overwritten to the address of the buffer on the stack. Thus, when the function returns it jumps to the location on the stack, with shellcode and executes the malicious code placed by the attacker.

DEP flags the stack as non-executable. Thus, when the program tries to return to the address of the shell code on the stack, the shell code will not be executed.

Code injection depends on being able to inject code into a memory region and then transfer control to that memory region. If it is flagged as non-executable by DEP, then malicious code will not be executed.

2 - 4 Marks

12. By exploiting buffer overflow, what all can be modified?

1. Return address can be overwritten (to make a malicious code to run)
2. Shell code execution (take control over entire machine)

13. Why is format string vulnerability considered as buffer overflow?

1. The stack itself can be viewed as a kind of buffer.
2. The size of the buffer is determined by the number and size of arguments passed to a function.
3. Providing an input format string can be used to read from or write into the stack.

14. What values should canary have to overcome buffer overflow attack?

1. Terminator canaries (CR, LF, NULL (i.e., 0), -1)
   • Leverages the fact that scanf etc. don't allow these
2. Random canaries
   • Write a new random value @ each process start
   • Save the real value somewhere in memory

- Must write-protect the stored value

3. Random XOR canaries
   - Same as random canaries
   - But store, canary XOR some control info, instead

15. How to make %eip to point to (and run) attacker code, more difficult?

a. Make stack (and heap) non-executable
b. Use Address Space Layout Randomization

16. What do you mean by smashing the stack? With an example, discuss how stack smashing takes places in memory?

Stack smashing means you've written outside of ("smashed" past/through) the function's storage space for local variables (this area is called the "stack", in most systems and programming languages). You may also find this type of error called "stack overflow" and/or "stack underflow".

Whenever you write outside an area of memory that is already properly "reserved" in C, that's "undefined behavior" (which just means that the C language/standard doesn't say what happens): usually, you end up overwriting something else in your program's memory (programs typically put other information right next to your variables on the stack, like return addresses and other internal details), or your program tries writing outside of the memory the operating system has "allowed" it to use. Either way, the program typically breaks, sometimes immediately and obviously (for example, with a "segmentation fault" error), sometimes in very hidden ways that don't become obvious until way later.
In this case, your compiler is building your program with special protections to detect this problem and so your programs exits with an error message. If the compiler didn't do that, your program would try to continue to run, except it might end up doing the wrong thing and/or crashing.
The solution comes down to needing to explicitly tell your code to have enough memory for your combined string. You can either do this by explicitly specifying the length of the array to be long enough for the strings, but that's usually only sufficient for simple uses where you know in advance how much space you need. For a general-purpose solution, you'd use a function like malloc to get a pointer to a new chunk of memory from the operating system that has the size you need/want once you've calculated what the full size is going to be (just remember to call free on pointers that you get from malloc and similar functions once you're done with them).

/*
Use buffer overflow and overwrite the Return address of a function to execute a malicious function that is never called.
(Refer Security tube, 1st video, CanNeverExecute() function)
*/

Stack smashing is a form of vulnerability where the stack of a computer application or OS is forced to overflow. This may lead to subverting the program/system and crashing it. Stack smashing occurs when a buffer overflow overwrites data in the memory allocated to the execution stack. This can have serious consequences for the reliability and security of a program. Buffer overflows in the stack segment may allow an attacker to modify the values of automatic variables or execute arbitrary code. Ex: Overwriting the return address to execute an arbitrary function.

17.How does a read buffer overflow attack take place?

Read buffer overflow occurs when we are trying to read the data present in the memory which is out of our access. This can be done using format string attacks where format specifiers can be used to print address/values present in the stack.

While reading from a buffer, the program goes over the buffer boundary and reads adjacent memory.

18.How an attacker does control the format string?

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Attacker controls the format string

Format specifiers can be passed as as input string. This can be used to read or write stack elements. In the above case if %s isnt used then the user entered format specifiers can access stack.

Example:

- `printf("100% dave");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers
- `printf("%08x %08x %08x %08x …");`
  - Same, but nicely formatted hex
- `printf("100% no way!")`
  - ***WRITES*** the number 3 to address pointed to by stack entry

19.When does a dangling pointer bug occur in a C program?

1. A dangling pointer bug occurs when a pointer is freed, but the program continues to use it.
2. An attacker can arrange for the freed memory to be reallocated and under his control.
3. When the dangling pointer is dereferenced, it will access attacker-controlled data. Example:

```
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(…);
free(p);
...
q = malloc(…) //reuses memory
*q = 0xdeadbeef; //attacker control
...
p->cmp("hello","hello"); //dangling ptr
```