Himesh Buch (hnb17)
CS [419]: Computer Security
Final Exam (Spring 2020)

1.

a) malloc(int n):

```
int *p
if (n > MAX_SIZE/sizeof(*p)) {
        // overflow happens so handle it here
}

/* checking if the total size is equal to n or not */
while  total_size != n:

   /* as described in the definition, this method
        Will only allocate the sectors whose size is larger than
        sizeof(struct heap_sector) + 1
   */
   check if sizeof(*first few memory sectors) > sizeof(struct *heap_sector)+
1:

        /* if no memory left => return NULL
           Else: => move memory from heap_free to heap_used
        */
        if no memory left:
           return NULL
        else:
           return move memory from heap_free -> heap_used
```

free(char *h):

```
free heap corresponding to char *h
return move memory from heap_free -> heap_used
```

b) Basically, any value greater than maximum supported value of unsigned char leads to integer overflow. Looking at the malloc(int n) function, the integer n is being passed as an argument. If the value of this integer is greater than the bounds of Integer, then overflow will happen. Also, malloc methods have sufficient range to represent the size of the objects to be stored, if the value of the passed object is not in that range, integer overflow will happen. In my code, comparing the values of int n using the proper sizeof command will help resolving the overflow. These are the lines that does that,

```
int *p
if (n > MAX_SIZE/sizeof(*p)) {
        // overflow happens so handle it here
}
```

c) Basically, if we continuously allocate memory and we do not free that memory space after use, it may result in memory leakage – memory is still being used but not available for other processes, and thus, heap overflow occurs. In our case, we allocate the first few memory sectors whose size is larger than sizeof(struct heap sector) + 1 (until the total size equals to n). In this process of memory allocation, if the memory is not freed after, heap overflow will occur. In the pseudocode, we check if there is no memory left, we simply return NULL, which ends the program. These are the lines:

```
if no memory left:
   return NULL
else:
   return move memory from heap_free -> heap_used
```

d) Off-by-one refers to a single-byte buffer overflow. This vulnerability is often related to the lack of strict boundary verification and string operations, even if the size of the write is just one byte more. While allocating memory, this can happen. In our case, while allocating the first few memory sectors whose size is larger than sizeof(struct heap sector) + 1, it can happen. In the pseudocode, we have a constraint that checks for it and hence it can be prevented that way. It can also happen because heap free is initialized to the first available byte.

```
check if sizeof(*first few memory sectors) > sizeof(struct *heap_sector)+ 1:

        /* if no memory left => return NULL
           Else: => move memory from heap_free to heap_used
        */
        if no memory left:
           return NULL
        else:
           return move memory from heap_free -> heap_used
```

e) a TOCTOU attack happens when two or more concurrent processes are operating on a shared file system. An attacker can alter the contents of heap in our case to successfully execute this attack. It can happen where we try to free the heap. In my pseudocode, these lines prevent it

```
if no memory left:
   return NULL
else:
   return move memory from heap_free -> heap_used
```

f) in the pseudocode, if the users free the memory multiple times and it might end up with no memory at all. In the code, we check if there is no memory left, we return NULL, so even if the heap is freed multiple times, if there is no memory left it will just end the program.

2.

a)

- Library Functions is nothing but an ordinary function that is placed in the collection of functions called the library. Here are some annotations,

@indicator: Indicating the switches between task instances and defines the place (where) to instrument. Here is an example on how it is used,

```
@indicator
buf_T* curbuf;
```

@identifier: Identifies different task instances and defines the value (what) to expose

```
struct buf_T {
   @identifier
   int buf_id;
   char* name;
   …
};
```

@channel: Communication channels used between instances and helps find relationships of all instances

```
@channel
struct yankreg *y_current;
```

@delegator: It adds one field for top-level task id and inherits task identifiers from the top-level task

```
@delegator
calss nsConnEvent {};

PostEvent(...) {
    event = new nsConnEvent(...);
    rv = target->Dispatch(event);
}
```

Here is another example,

```
do_ecmd(. . .){
   ... //a new buffer
   curbuf = buf;
   curbuf->name = . . .
   if (curbuf != oldbuf) {
      oldbuf = curbuf;
      expose(curbuf->buf_id);
   }
}

expose(id){
    kill(-100, id);//for linux audit system
}
```

Looking at the above solutions, here are some designed library calls. The goal of these commands is,

- Audit file access and modification and see who changed a particular file
- Detect unauthorized changes and monitor system calls
- Detect anomalies like crashing processes
- Record commands used by individual users

1. We can basically define what process we want to track and the related system call. We can do that by creating a rule. The entire audit library in linux, works perfectly fine for this

```
auditctl -a always,exit -F arch=b64 -F pid=8000 -S open -k crypto-open-files
```

The above created rule, calls monitor on open on PID 8000, Now when this process uses the open system call, it will be logged in the audit log. We give it a key “crypto-open-files”. Once this rule is added, users can simply trace the file activity and will get better results which will help to back trace and monitor the process. Here is how it can be called,

```
ausearch -k crypto-open-files
```

As explained, we can easily run this command by above mentioned key “crypto-open-files”, and search for activity.

The ausearch command can also be used to Finding the related event or access to the file

```
ausearch -f /etc/passwd
```

Here are some more commands that serves the same purpose,

- aureport: reporting tool which reads from log file (auditd.log)
- autrace: using audit component in kernel to trace binaries

- aulast: similar to last, but instead using audit framework
- aulastlog: similar to last log, also using audit framework instead
- ausyscall: map syscall ID and name
- auvirt: displaying audit information regarding virtual machines

2. **lsof**: this call will help us see which files are open. It can really show any type of open files, to tracking open network connections. This will help in understanding what files were used/opened and help us get to the bottom of the dependence explosion attack. As explained, it will also help in identifying what is going on, on what port, which will help to identify what process is running on what port, if the port number is known. Here is pseudo code for lsof call:

```
lsof -c <daemon name>
lsof -i -n #to check open network connections
```

3. **strace**: This is used to track the right call, and we can see exactly what files are opened while it happens. I think the main advantage of this command (library call) is for tracking required file access, dependencies, and troubleshooting purposes.

```
strace ls
```

4. **ltrace**: detailed list of what process called the function, and works the best when run on individual process or group process

```
ltrace -e some_func@some_lib
```

5. **grep -r**: check whether there are just two or three doubles fit for calling the function and work your way from that point. In spite of the fact that that could work in some constrained measures of cases, this is in no way, a general solution, and wouldn't work if for example “some_func” is universal in different binaries yet is only here and there called.

```
grep -r some_func /some_dir
```

b)

All the above-mentioned calls pass these requirements,

1) applications can use to actively tell the kernel its semantics
2) kernel can capture such semantics and log them
3) such semantics can be used to improve attack forensics (e.g., solving the dependence explosion problem).

Before we mention how the above-mentioned commands help with dependence explosion problem, here are some solutions to that,

- 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 will not be aware of that.

All these calls actively either logs or notifies the kernel on what the situation is of any given process or command. This will help in knowing what command is being executed by what processes and can also identify how long has it been running for. If the process or command has been running for a long time without the user knowing about it, it should raise the red flag and the user should immediately trace and investigate it in order to prevent any kind of malfunction. As mentioned in part a, in order to be safe from the dependence explosion problem, logging and killing the unwanted processes is an important part and all the mentioned commands help in doing that.

For instance, the “indicator” call helps in identifying when the tasks have changed, and the “identifier” call helps to identify what tasks have changed. This helps a lot in understanding what processes were run, and eventually be back traced. The “lsof”, “grep” commands also help in doing the same. And hence, we can say that all the added library can achieve the three mentioned goals.

3.

a) FGSM uses gradient information to generate adversarial examples and they are maliciously perturbed inputs intended to misdirect AI (ML) models at test-time. They regularly move: the equivalent adversarial examples can fool more than one model. By examining the transferability of adversarial examples on multiple state of the classifiers we can conclude our theory of it attacking two models than one. It would still be reasonable to believe that it is only neural networks that are vulnerable to these kinds of attacks and that adversarial examples transfer between a wide variety of model types. Also, the FGSM can be formulated like this x’ = x + (sigma)*(function of Loss), where sigma controls the strength of the attack. Hence, by examining this “moving” or transferable property and the increase in sigma clearly guarantees that x’ can attack two models instead of only one

b) As referenced in the question, the adversary does not approach class probabilities or scores, and does not even know the model architecture, trained weights, training data and not even the number of output labels. Rather, the enemy just approaches a list of k inferred labels. We expect to devise an attack that works for this situation and can exploit additional data. The assault strategy is to train a substitute system on network on a small number of initial queries,

and then iteratively perturb inputs dependent on the substitute network's gradient to expand the training set. To summarize

(1) gain a little beginning dataset, labels in our case
(2) select a design for the substitute network that is reasonable for the domain, and emphasize the accompanying
(3) question the target model for labels to any unlabeled datapoints
(4) train the substitute system on current datapoints
(5) enlarge the dataset by using the information gained from above steps

c) Image preparing changes, for example, picture interpretation and brightness alteration, are normal and promptly accessible blinding functions. Let x be the image that an attacker would like to query the model for, x' be the model's output for the query, and T be a randomized picture processing algorithm (e.g., by turning it or moving it by an arbitrary amount p). Since the reason for our blinding function is to trick the query detector by changing a succession of queries which are pairwise like an arrangement of queries which are not, we would like the distortion between the original image and the transformed image to be large. Basically, after distortion, these changes despite everything hold the essential substance of the picture, and a model with high accuracy should create moderately comparable outputs for the original and changed images. By this way, we can effectively generate adversarial examples in the black box setting