Jung Yoon
CS 361S
O. Jenson

HW #2

Slip Days: 0

PROBLEM 1

a.

char *crypted = malloc(len + 4);

We need to check the return value of malloc and ensure that it isn't NULL. It might be the case that there isn't enough memory to dynamically allocate it.

Actually, even if the return value isn't NULL, there's another problem in that len+4 itself can overflow if len is large enough. For example, if we set len = 0xFFFFFFFF, then len + 4 is 3. This would mean that we would allocate a 3-byte buffer but then we could read a lot more than 3 bytes into it. That is, we would allocate 3-bytes but then we would read a lot more than 3 bytes into it within the for loop and within the line:

**(size_t *)(crypted) = len;*

This leads to saying that we need to stop the possible integer overflow where it begins by checking the value of len (the size of str) where it is first initiated. So:

size_t len = strlen(str);

The value for len is assigned without checking if it fits in the bounds of size_t variable. We need to ensure that str is an acceptable length, otherwise it is vulnerable to an integer overflow. It's important to note that size_t is an unsigned integer type AND so is *i* itself in the for loop so in a case where integer overflow happens when len is too large, len would overflow to a positive value rather than a negative value. This means that the for loop would still be a finite one, meaning that the program would run without detecting the flaw.

Like stated previously, this would be a classic integer overflow problem.

****(size_t *)(crypted) = len;***

This has already been explained but to explicitly mention it again, if len is really big then there are cases where we could extend the allocated space.

****(crypted + i + 4) = *(str + i) ^ (key[i%8]);***

Same story here. This has already been explained but if len is really big then i is also really big at times and we will write far more than was allocated because of the overflow.

> "There's another problem in that len+4 itself can overflow if len is large enough. For example, if we set len = 0xFFFFFFFF, then len + 4 is 3. This would mean that we would allocate a 3-byte buffer but then we could read a lot more than 3 bytes into it."

b. It's all up there. Just gotta rewrite it down here.

```
char *molvCrypt(char *str, char *key) {
    // MAKE SURE THAT KEY IS 8 BYTES
    // OTHERWISE RETURN NULL

    size_t len = strlen(str);
    char *crypted = malloc(len + 4);
    unsigned int i;

    // RETURN VALUE
    // if (crypted == NULL) { return NULL;}

    // INTEGER & BUFFER OVERFLOW
    // ensure that len+4 is not over FFFFFFFF
    // if (len+4 > 2147483,647) { return NULL; }

    // note that len will always be positive since it's size_t type

    for (i = len; i != 0; i--) {

        *(crypted + i + 4) = *(str + i) ^ (key[i%8]); // XORing with the key
    }

    // INTEGER OVERFLOW
    // ensure that len does not exceed 4 bytes, otherwise return NULL
    // if (len > 2147483,647) { return NULL; }

    *(size_t *)(crypted) = len;
    return crypted;
}
```

PROBLEM 2

a. First, let's note that a buffer overflow is a vulnerability in which a program overflows the buffer as a result of not properly checking the bounds. Canaries are used to verify stack frames' integrity prior to returning. That is, they make sure that the stack frames haven't changed. Most buffer overflows overwrite memory from lower to higher memory addresses so the canary value must be overwritten in order to overwrite the return pointer. This is why the canary value(s) are checked; it's in order to make sure that it has not changed before return pointer on the stack is used.

   What's important to note is that canaries, as great as they are, do not provide full proof protection. For example, some smashing the stack attacks leave canaries completely unchanged. That's where the libsafe defenses come into play.

   What libsafe does is it halts the process when overflow exploit is detected. That is, it's runtime code that detects overflow exploits. Libsafe specifically is a library that intercepts and halts calls to vulnerable buffer overflows. For example, the *gets* function is replaced with *fgets*, they replace strcpy(), scanf(), etc.

   What's nice is that Libsafe can guard against buffer overflow attacks that haven't been discovered yet (errors/weaknesses that you don't even know are there), whereas canaries are intended to catch specific ones - the ones that make changes to the stack frames. Canaries only check if stack frames haven't been compromised; they don't check if certain calls to unsafe functions have been called to bypass the canaries like Libsafe does. It's an extra measure of protection that stack-canary-equipped executables doesn't give.

b. Neither of the defenses given are perfect, even in combination. Libsafe's limitation is that it offers limited protection. There's a list of calls that it guards against. It's nice and all but it's not very much if you think about it, though it does help to detect vulnerabilities in your code that you're not even aware of.

   In particular, no, it is not protected against stack smashing. The length of uid and pwd is checked against the stack frame according to Libsafe but note that if it's too long, the attacker can still overwrite parts of the stackframe. It will just overwrite the destination in one of the strcpy calls to point to the return address, making it so that the return address is overwritten by the source of strcpy. This won't affect the canaries.

PROBLEM 3

a. First, let's note that IP spoofing is the creation of IP packets using somebody else's source IP addresses. Spoofing is just when the attacker pretends to be something that they're not; it's the process of providing false identity in order to gain unauthorized access to secure resources and/or networks.

   Now, Blind IP Spoofing is called "Blind" because the attacker attacker does not have access to the reply. That is, the attacker just sends IP packets to the victim's computer and does not wait for a response, hence the "blind" part. That is, it's "blind" because the attacker just has to blindly guess/trust that they will get a response from the target/victim computer at some point. Only after the attacker gets a successful response may he cause further damage / steal information from the victim that's communicating with the attacker. We can almost say that the attacker doesn't care about the response until he gets it.

   Just as one would expect, this is different from Non-blind IP Spoofing where the attacker is certain that he will get a response from the victim's computer to start communicating.

b. Is it perfect? No, nothing is ever perfect. Does this help against Blind IP spoofing, as asked in the question? Certainly. It is hard to do IP spoofing (both non-blind and blind) on TCP because successful spoofing requires the attacker to guess the correct initial sequence number. This is really difficult to do since it is generated by the server in a non-guessable way. This means that the attacker won't be able to guess the correct sequence. In Blind IP spoofing, the attacker would need to send all possible initial sequence numbers by brute force until they got the correct one.

   Again, we're not saying that it's impossible to use Blind IP Spoofing against this defense but it will certainly help to prevent it.

PROBLEM 4

First, DNS cache poisoning is a type of attack that exploits vulnerabilities in the domain name system (DNS) to divert Internet traffic away from legitimate servers and towards fake ones. This attack can be used to direct users from a website (http://msa.mi/topsecret) to another site of the attacker's choosing. So, without explaining how a DNS cache poisoning is performed, the attacker could use it to have the employee sent to malicious files containing either 1) spyware to get the information or 2) it could have a cross-site scripting attack to overcome the origin policy.

The good part is that once the fake address is used by the employee, it could/would continue to infect other employees who tried to access the site since DNS cache poisoning is contagious.