5271 Exercise Set 1

Helge206 Andrew Helgeson
Wayt0012 Ethan Waytas
Maure113 Brian Maurer

1)
(Part 1)
Properties we want to insure (security goals):

Problem Assumptions:
• We assume that the database that the professor has created will be somewhat accessible to students via the internet/intranet from some front end system (e.g. moodle, GRIT). We make this assumption for the following reasons:
    • Students like to be up-to-date on their standing in a class. Having the grades available online alleviates the amount of queries sent to the professor/TAs.
    • If we assumed that the professor would be the only one accessing the database on a private personal server or laptop (see part 2), the threat model would be reduced from a large set of people to a smaller set with close proximity to the professor. This would make the question not nearly as interesting.

Internal (within the course):
• A given student can only access grades to their own assignments/exams/etc. (read only)
• A professor can access all fields in the database for this course (read/write)
• All students in the course, professors, and TAs should have read access to homework, and grades.
• TAs, depending on course policy, should have read and write access to everything in the database.
• People outside the class, but associated with the college or university, with legitimate claims to certain contents should be able to access them in a controlled fashion. For example, the registrar may need to access the grades in the database, but through the principle of least privilege should not be able to access exams or homework.

External:
• Assuming complete class confidentiality, the default for anybody who is not a professor, TA, or student (the next bullet point covers all other specific cases) cannot access the database.
• There could be reasons for other staff or faculty to be allowed access but we will assume that anyone under these circumstances can be added on a case by case basis with the most limited possible permissions granted to them.

General:
• Actions shall be traceable.
    • ex. if a grade is changed, no matter who changed it, the action should be recorded. This

should be read only and only accessible by the prof, the TAs and advisors (who would need special permission as described in the second bullet point under the “External” list above). Nobody can overwrite this.

• The database should be dependable.
    • Students might check assignments, grades, etc at any time so there should be a reliable system in place.
    • TA’s or professors may need to make emergency assignment or exam changes which require access to the database

Possible Adversaries (threats):

Internal:

• Student (or other internal persons with some special case access) trying to change modify information that they shouldn’t be able to modify (grades, exams, etc)
• Students (or other internal persons with some special case access) may try to access information that they shouldn’t have access to (ex. trying to recover exams might be an easier way to cheat than trying to modify grades.)
• Professors/TAs might try to secretly modify grades in a way that is advantageous to their “favorites”

External:

• Outsider trying to change grade's. These outsiders aren’t necessarily part of department or University.
    - could be a sophisticated attacker (more expansive resources)
        - The attacker could also be more sophisticated in the sense of technical knowledge, specifically in the system being used.
    - could be a person interested/related to a student (low resources)

General:

• Power failure – there should be a backup power source in place
• Natural disaster – data should be backed up (encrypted) off site in the rare chance of a natural disaster
• Hardware/software failure – backup off site (see above)
• We should exclude threats from situations that are completely out of budget range, such as:
    - Attacks from a supercomputer
    - Natural disasters that are unlikely in the location of interest (ex. We can put servers in the basement if the given location never floods.)
• We can also exclude threats that come from within our trust boundary
    - This includes the professor himself, although even his actions will should be traceable if anyone wants to question his actions.
    - Depending on the class policy you could likely put TA’s within the trust boundary for the system. Since you are granting them read/write access for likely the entire system it makes sense they would fall into this trust boundary.
    - Earlier we discussed access for people outside the class itself such as the registrar.

While we said the principle of least privilege should be used here you can likely assume the registrar of the school is without an ulterior motive.

Potential Attack Vectors (for the adversaries above):

• Physical access to the hardware

- Physical access to the hardware would make it easier for an attacker to steal or copy the entire database.

• Exploiting a software vulnerability

- Software vulnerabilities could compromise the system. Obviously if the system providing the service/interface to the database has an exploitable flaw then the data is insecure.

• Denial of Service attack

- Even if someone doesn't have knowledge to exploit the software or are without physical access to the database they can still create a denial of service. Say a student doesn't have a big homework assignment done on time. With not that much technical knowledge they could try and bring down the system so other student's cannot submit, thus forcing a deadline extension.

1)

(Part 2)

For both options below we want to note that our primary concern is data confidentiality. It is less damaging to the university and the students if we destroy the data than it is if we leak the data. Lets look at some options to counter those threats.

Option 1: Disk encryption

Prevents data that is stolen, by (a)hacking the database or (b)physically stealing the HD, from being readable without decryption.

Annualized attack incidence

(a) **1** per year (or less, but use 1 as worst case)

(b) **1** per year (or less, but use 1 as worst case)

Annualized cost to implement defense

$0 to encrypt.

Defense effectiveness

In very rare cases an attacker might have a supercomputer and the abilities to decrypt the information stole. We will estimate that disk encryption is **~95%** effective (conservative estimate) against attacks where the data is stolen.

ALE

(a) Here data is not lost (just stolen), but user confidence in their grade confidentially is lost. This does not cause an instant loss but it could discourage students in the future. Lets say that for every **10** students compromised, **1** future student does not attend the university where tuition is **$10,000/yr**.

(b) Here data is stolen and lost. We lose everything the same as (a) plus all value

of the time that students, professors, and TAs spent working to create and enter the data into the database.
In the worst case scenario everything has to be redone so we will value student by **(cost per credit * # of credits)** for that course and we will value the professor by **(hourly wage * hours spent to create the course material)**.

So if we assume: class size = **60**, cost per credit = **$1000**, # of credits = **3,** hourly wage = **$45/hr**, and hours spent = **200hr**

$ALE(a) = \frac{60}{10} * \$10,000$
= $60000
$ALE(b) = ALE(a) + \frac{200hr}{\$45/hr} + (60 * \$1,000 * 3)$
= $60000 + $189000
= $249000

Net Risk Reduction

Since encryption is **~95%** effective against attacks and costs **$0** to implement
NRR = .95($249000 + $60000) - $0
= $293550

Option 2: Store a computer in a safe(waterproof, fireproof) when not in use

While in the safe (d)loss of data due to unexpected floods and fires is prevented and (e)data theft is prevented.

Annualized attack incidence

(d) **1** per year (or less, but use 1 as worst case)
(e) **1** per year (or less, but use 1 as worst case)

Annualized cost to implement defense

One time purchase of ~$500 (about how much a safe like this costs.)

Defense effectiveness

This defense is not effective unless the laptop is in the safe. As a worst case scenario we will say the prof. has the laptop out during all work hours (2000hr in a normal work year). Since there are 8760hr total hours in a year, the case is used (1 - 2000/8760)*100% = **77%** of the time.
When in the safe:
(d) Assume the safe is **100%** effective against fires and floods/rain/sprinklers.
(e) Assume the safe is **95%** effective against theft. (Conservative estimate.)

ALE

(d) Here the only problem is that data is lost. The logic is the same here as in the previous example so use ALE(a) and ALE(b) from above to get:
ALE(d)= ALE(b) - ALE(a)
= $189000
(e) Here data is lost and stolen, so it is the same as ALE(b).
ALE(e)= ALE(b)

= $249000

Net Risk Reduction

NRR = (($249000*.95 + $189000*1)*.77) - $500
= $327173.5

Interestingly enough, from this analysis, even though we care more about keeping person's information private, it is more expensive to lose the data, partially because that could also result in loss of privacy, than to lose privacy.

2)

(2a) We can input a string for username such as "; <arbitrary system call(s)>". For example, assuming the CGI script's permissions are sufficient, we could input " ; rm -rf /*", which would start deleting from the file system root. The program would execute the command because of the following line:

- $result = 'last -1000 | grep $username_to_look_for';

Using backticks in a CGI scripts means that the line will be executed as a system call. The attack string begins with a semi-colon which acts as a command separator. If an adversary desired to delete all the files from a computer, they would set $username_to_look_for to the following.

- " ; rm -rf /*"
- The full string would then become:
  - `last -1000 | grep ; rm-rf /*`;

The first command executed would be "last -1000 | grep ;", which would do nothing. The second command would then be "rm -rf /*", which would start recursively deleting all files that the CGI script has permission for from the root directory. By simply inserting a semicolon and then system command(s), an attacker could either completely take control of a computer or wipe it out.

There are two ways to defend against this type of attack. The first would require removing all undesired characters from the input string (blacklist). For example, all occurrences of semicolons could be replaced with the empty string. Depending on the language, there is a possibility that a function exists to check an input string for unallowed characters. For perl, it is conceivable to use the "is_tainted()" function to see if input is safe (See: Laundering and Detecting Tainted Data). Additionally, unallowed characters could be parsed out of the data as shown below (source: Laundering and Detecting Tainted Data):

```
if ($data =~ /^([-\@\w.]+)$/) {
        $data = $1;                     # $data now untainted
  } else {
        die "Bad data in '$data'";      # log this somewhere
  }
```

The second option would be to have a whitelist of accepted characters. The problem with the first option of a blacklist is that new attack vectors are always found (e.g. using “|” instead of “;”). All vulnerability fixes become reactive, instead of proactive. To head off attacks, we could restrict a username, for example, to only have letters. In this case, acceptable characters would be a-z and A-Z. It would be up to the program creator (or project requirements) to decide what domain of characters are allowable. By creating a whitelist, we only allow characters that we know will not cause a problem.

- For example, some pseudocode would be:
  if ($data !~ /^[a-zA-Z]+$/)
  //data is invalid, perhaps report an error

In summary, any user input should be parsed to check for malicious intent and be potentially restricted to a smaller set of characters.

(2b) In order to successfully delete the last byte from file /what/ever, we must exploit a Time of Check / Time of Use (TOCTOU) vulnerability. Specifically, we are going to be creating a file and then a link to another file to manipulate the program to remove the last byte from file /what/ever (which we are not supposed to be able to do).

A couple of quick assumptions:

1. We assume that we can get the size of /what/ever. This may just be as simple as going to directory /what and doing and “ls -l” and seeing what the size is (of course if we are unable to do this, we could spam silly_function with different size files, but what fun is that?). The size of of /random/file must be the same as /what/ever for the following reason:
   a. A specific amount of memory is malloc’ed. If the size of /what/ever is greater than /random/file, we will not be able to write all of /what/ever into the buffer. Conversely, if the size of /what/ever is less than /random/file, we will not meet the requirements of the assignment, namely to delete the last byte from /what/ever. Therefore, we must have the exact size of /what/ever.
2. For this attack we assume that the processor nicely puts silly_function to sleep between certain parts of code (maybe to run another process, etc.). A quick check of wikipedia about TOCTOU attacks (source: TOCTOU) reveals that it is best to create a maze of files that all have the same hash, so that when a system call is made, the OS has to search through a long list after hashing and then go to disk. This gives the attacker time to create a symbolic link between the innocent file and the desired file. For simplicity we just assume that everything nicely works out (which, probabilistically, it would, if you tried enough times). Multiple tries might be in order to actually have the correct order of context switches line up.
   a. If local access to a machine is readily available, it could be possible to run the executable in gdb and break in the appropriate places, ensuring that linking and unlinking is done correctly.

The first step of the attack would be to create a file, say “/random/file” with the same size as “/what/ever” (this ensures that we delete exactly the last byte of “/what/ever”). We then call silly_function with “/random/file”. Remember, /random/file only needs to be the same size as /what/ever, the contents do not matter.

```
void silly_function(char * pathname) {
        struct stat f, we;
        int rfd, wfd;
        char *buf;
        stat(pathname, &f);  <- contains file information about “/random/file”
                At this point, pathname is still set to “/random/file”
        stat(“/what/ever”, &we);
        if (f.st_dev == we.st_dev && f.st_ino == we.st_ino) {  <- a context switch here does
        not change what is stored in the struct f
                return;
        }
```

At any point between stat(“/what/ever”, &f) and immediately before “open(pathname, O_RDONLY)” if the process is context switched, an attacker would do the following (or more likely a script):

- **“ln -s /what/ever /random/file”**

Now pathname still points to /random/file, but /random/file is a soft link to /what/ever, meaning that opening /random/file will open /what/ever

```
        rfd = open(pathname, O_RDONLY);  <- we have actually opened /what/ever
        buf = malloc(f.st_size - 1);
        read(rfd, buf, f.st_size - 1);
        close(rfd);
```

Once again, we need a context switch between rfd = open(pathname, O_RDONLY) and before stat(pathname, &f). During this context switch the attacker would delete the soft link (not what it points to) /random/file. Then, /random/file would need to be created again with the same size as /what/ever.

```
        stat(pathname, &f);  <- pathname now points to the actual file /random/file

        if (f.st_dev == we.st_dev && f.st_ino == we.st_ino) {  <- a context switch here does
        not change what is stored in the struct f
                return;
        }
```

Here is the last context switch required for the exploit. This must occur after stat(pathname, &f) and before open(pathname, OWRONLY | O_TRUNC). Once again during the context switch a soft link is created from /random/file to /what/ever.

```
        wfd = open(pathname, O_WRONLY | O_TRUNC);  <- pathname is /random/file which points to /what/ever. File /what/ever has now been opened to be written to.
        write(wfd, buf, f.st_size - 1); <- writing out everything but the last byte to /what/ever
        close(wfd);
        free(buf);
}
```

We have now deleted the last byte from /what/ever. While this is rather silly, a more sinister approach would be to write the contents out from /what/ever and view them (perhaps /what/ever contains passwords). To do this we would not change pathname after the second “if (f.st_dev...)” check. Now the contents of buf (currently holding what was in /what/ever) would be written out to the file pointed to by pathname, which could be read by the user. Also note that we could make the size of /random/file be huge, thereby ensuring that /what/ever is fully written into the buffer (in the case we do not have the permissions to see the size of /what/ever).

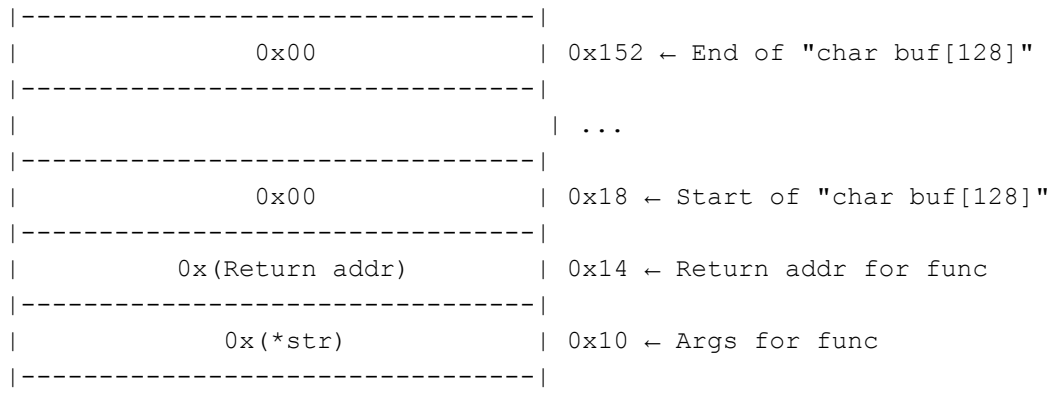
3)
(3a) Reversing the direction of stack growth or changing the direction of buffer growth does not eliminate the potential for buffer overflow control flow hijackings. This architectural modification just changes *which* return pointer you will compromise.

With a regular stack the buffer overflow is used to overwrite the return address of the frame you are currently in (the one with the exploitable buffer). If the stack moves in the opposite direction you can still overwrite the return address of called functions. In order for this to happen the function being called should be modifying the buffer.

In the example given you have a 128 byte char buffer that can potentially overflow when strcpy is called. In a normal buffer overflow you would overwrite func’s return pointer but with a reversed stack you will overwrite the return address of strcpy’s frame while it is operating on the buffer.

Here is a diagram displaying what happens with the reversed stack:

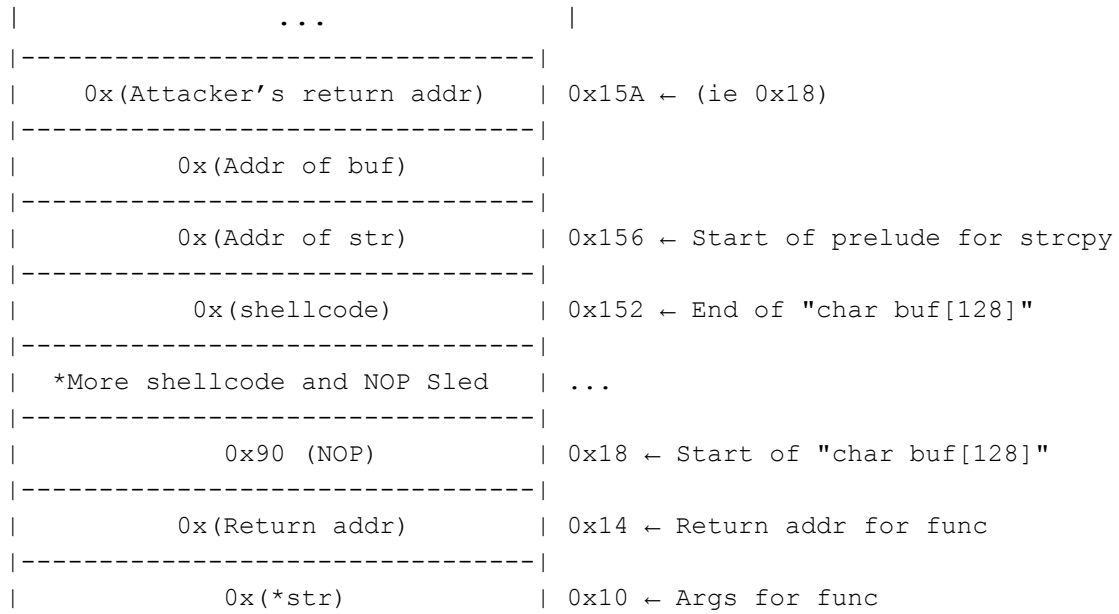
```
|               ...                |
|----------------------------------|
|    0x(Return addr for strcpy)    |
|----------------------------------|
|          0x(Addr of buf)         |
|----------------------------------|
|          0x(Addr of str)         | 0x156 ← Start of prelude for strcpy
```



This is a cutout of the frame for *func* after it gets called and right before strcpy is called (arguments and return pointer already pushed). I assume here that *buf* is initialized to null throughout for the purposes of the diagram. The first thing *func* does is call strcpy on the user input. If the input is longer than *buf* and crafted appropriately, strcpy will actually overwrite its own stack's return address in the process of the copy.

In my version of the reversed stack there is a slight problem because the arguments to strcpy come before strcpy's return address. This poses a problem if strcpy uses the arguments directly and doesn't make local copies since if you overwrite the arguments with junk or some other address the copy may fail or write data somewhere you don't want to. To avoid this the attacker would have to craft the input such that it overwrites the arguments to strcpy with their original value, essentially not modifying them. Finding these values may take some work (or guessing if your system has ASLR) but for the rest of the question I assume the attacker can find these.

The next diagram shows what the stack will look like after the attacker's string has been copied into buf and caused an overflow.



|------------------------------|

As strcpy copies *str* it unknowingly smashes its own stack frame. This allows the attacker to overwrite the return address for strcpy while it is running. In this case I gave a return address of 0x18 which would cause strcpy to hit the nopsled in *buf* and slide down to the injected shellcode.

So as we have shown, the reversing of the stack does not help protect return pointers, it just changes which one gets overwritten.

(3b)

To fool the check for 100 connections we will need to modify the *tmp* integer variable during the overflow. We also need to take into consideration the extra bytes added in the later strcat in order to not kill the canary. We also need to take the byte ordering into consideration.

Little Endian System:
On a little endian system the least significant bytes will be stored closest to the end of buf (lowest address). If total_connections is equal to 100, the only set bits for the integer will be in the 1 byte following buf so this will be our target. The following is some pseudocode for a buffer that you could pass to accept_connection to exploit the overflow and gain a connection when you shouldn't.

```
char attack_buf[] =
        "blahblah...."   // Write out 128 bytes of whatever (except a null byte)
        "\x1"            // The value we want to overwrite in the least significant byte of tmp
```

This *attack_buf* will actually overwrite 2 bytes in *tmp* because of the null terminator. The *attack_buf* would work if you eliminated the second line ("\x1") but then the exploit will only work for values of *tmp* (ie total_connections) that fit on the lowest byte. We can still write 2 bytes ("\x1\0") without later overwriting the canary with the strcat and this allows the exploit to work even when *tmp* is big enough to fill the lowest 2 bytes.

In memory from highest address to lowest consider *tmp* = 100, here is its byte layout with *buf* following *tmp*. (tmp = blue, buf = green)
I separated individual bytes with a | for clarity
00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 01 10 01 00 || 00 00 00 00 | ...

After the buffer overflow here is what the byte layout should look like. "x" being whatever junk is put in to fill the first 128 bytes of buf.
00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 01 || xx xx xx xx | ...

Now the value of *tmp* is 1 and will pass the check. The following strcat will append the :Y or :N bytes on the 2nd and 3rd bytes of *tmp* and the null terminator will go on the 4th (most significant) byte. Nothing will be written past *tmp* so the canary and any other stack data will be left

untouched.

Big Endian System:
On a big endian architecture the attack will work almost the same except for the byte value you place at the end of *attack_buf*. In the previous attack on a little endian system we overwrote the lowest order bits that were already set in order to get the value below 100. In the big endian scenario we would have to write over all bytes in *tmp* to get the same result as before but this would cause the subsequent strcat would smash the canary.

In this case the bytes of *tmp* that are closest to *buf* are the most significant in the integer. Assuming this system uses two's complement for representing signed integers we can overwrite this most significant byte to something that has a leading 1 in binary so the integer representation will become negative. Here is our modified *attack_buf*.

```
char attack_buf[] =
        “blahblah....”   // Write out 128 bytes of whatever (except a null byte)
        “\xFF”           // The value we want to overwrite in the most significant byte of tmp
```

Here are the before and after byte patterns when *tmp* is 100.

01 10 01 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 || 00 00 00 00 | …

01 10 01 00 | 00 00 00 00 | 00 00 00 00 | 11 11 11 11 || xx xx xx xx | …

As before, you can't see any bits changing but the null terminator was also written in the 2nd most significant byte. Now if I rearrange the bit pattern to how you usually see it you can see this integer will now *always* be negative. This is even more useful than the attack on little endian which would only work up to a value that fits in 2 bytes.

11 11 11 11 | 00 00 00 00 | 00 00 00 00 | 01 10 01 00 -> Definitely less than 100

When the strcat appends its string it will work just as before and nothing will be modified outside the local variables.

4)
(a)
Bad Programming Practices:

1. a or b can be null
    a. If a is null there will be a segfault in strlen
    b. If b is null there will be a segfault when b[i] is accessed
2. Malloc could fail and return null.
    a. Malloc could fail because of low system memory or an overflow on 2*len causing

an attempted negative space allocation. There are probably other reasons but these are the most obvious.
    b. With no check on *result* it could be null, causing a segfault when accessed in the for loop
3. The for loop has an off-by-one error
    a. Actually it is an off-by-two error as well. When i == len in the for loop, *result[2*i]* will be one past result's end and r*esult[2*i + 1]* will be two past.
4. No check if *a* and *b* are the same length.
    a. The allocation of *result* is only made based on the length of *a* and even then doesn't account for the null terminating byte which causes the off-by-one/two error above.
    b. If *b* is smaller than *a* then as the for loop continues, values from past *b*'s memory will be read into result (that's if the program doesn't segfault).
    c. If *b* is larger than *a* then not all of *b* will make it into result.
    d. If *b* is not equal to *a* (regardless of direction), some portion of a or b will be missing from the **string**, not necessarily *result*. This is because when the strings are not the same length one will terminate its copying first and lay down a null byte which defines the end of the string so even if the other string continues to be written it won't show up if the string is printed later. The data will still be in result though. If one or both of the inputs is influenced by a user they could manipulate it so that only part of one of the strings makes it into the zipped string if this served them some purpose.
5. 2*i could overflow in the for loop
    a. This is a problem but it will have caused other problems earlier such as in the malloc. It would still need to be fixed.

Fixes for Bad Practices

1. Simply add a check at the top of the function for null values. Then you need to decide on an appropriate return value.
    a. You could return null and trust the caller to check the return value. This may not be what you want since one of the strings could be null.
    b. You could return a 0 length string.
    c. You could check *a* and *b* for null's separately and even if one is null still create the zip of the non-null one. If they are both null then you are still left with choices a and b. This strategy seems to make the most sense.
2. The main problem here could be fixed by adding a check on *result*. This still doesn't avoid the integer overflow of 2*len. You could assume that malloc would return null in this case but if it is undefined behavior you would be better off explicitly checking if the integer is going to overflow and then making a return decision similar to #1 if so.
3. To fix this you need to allocate 2 extra bytes, one for the null terminator in each of *a* and *b*. This does not in any way fix the problems of #4.
4. Instead of taking the length of just a and doubling you should have something like
    .. (checks on a&b)

```
int len_a = strlen(a)
int len_b = strlen(b)
… (add any other checks)
result = malloc(len_a + len_b + 1)
…
```

This doesn't fix the for loop accesses, it just gives us enough memory. To fix the for loop it should be broken up into two loops, one for each input string. You also have to consider what happens when one string finishes before the other. If you notice we only allocated 1 extra byte above since we believe the correct semantics, as opposed to the current operation, is to only have a null terminating byte at the very end, not somewhere in the middle.

Here is some pseudocode for these for loops:

```
…
result_len = len_a + len_b + 1;
for (i = 0; i < len_a && i < len_b; ++i) {
        result[2*i] = a[i];
}
index = 2*i;
while (index < result_len&& i < len_a) {
        result[index] = a[i];
        ++i;
        ++index;
}

for (i = 0; i < len_a && i < len_b; ++i) {
        result[2*i+1] = b[i];
}
index = 2*i+1;
while (index < result_len && i < len_b) {
        result[index] = a[i];
++i;
++index;
}
result[result_len - 1] = '\0';
...
```

- If *a* is longer than *b* it will copy into even bytes only until it reaches *len_b* bytes have been copied and then it fills in the remainder of *a* into result in sequential bytes, terminating with a null.
- If *b* is longer than *a* then *a* will finish the first for loop with all of its bytes written and the while loop will be skipped. The for loop for *b* will copy until it has reached *len_a* bytes and then start writing its own bytes sequentially where *a* would have

its null terminator.
    - If *a* and *b* are equal length then *a* will copy all of its bytes to the even positions and skip its while loop. *b* will then copy all of its bytes to the odd positions and skip its while loop. Then the string will be null terminated at the end.

During the copying the null byte for each string is skipped so we won't end up will floating terminators in the *result* string.

(b)

Bad Programming Practices

1. Input1 and input2 are not checked for null
    a. This will cause a segfault in the strcmp right away.
2. Input lists are not guaranteed to end with a null terminating list element. If this is the case the merge could fail in a segfault or it could complete with an improperly terminated list.
3. Linked list nodes may have improperly terminated strings for their data field. This could cause a segfault strcmp.

Fixes for Bad Practices

1. To fix this we would add a check at the top of merge. From here, there are two choices.
    a. If both input1 and input2 are null, we could return null
    b. If either input1 or input2 is null, one of two things could be done.
        i. Return the non-null list
        ii. Return null
        iii. Both i. and ii. are valid options and would be up to the requirements of the greater project (a null return might imply a failure to solely indicate that the two lists could not be merged).
2. There is no solution if the input lists are not properly null terminated. If a pointer is not null we cannot know if it is valid until dereferencing at which point it may be too late. The responsibility of a correct list, in this case, is up to the programmer passing in the input to merge.
    a. Behavior in this case would be undefined, a small percentage of the time, a result could be returned (the remaining time would be seg faults).
    b. If you are constructing a threat model it would be important to consider who the callers to this function are. If this is a public api method for a larger application then more stringent input requirements may be necessary. On the other side, if this is a function internal to your application you can likely trust the input you receive, assuming there is not user input in the list creation or previous code properly sanitizes the data.
3. Similar to #2, a solution to an improperly terminated string (no null terminator) requires careful consideration.
    a. The safest implementation would be to restrict the size of data in the struct. Having a fixed size allows functions to assume upper limits to bounds in any loops that are manipulating data (potentially reducing overflow errors). It would

also allow the use of strncmp which prevents “walking” off the end of a string array.

   i. This is most likely the best and safest solution, however there is a drawback in terms of size. A large number of list structs could potentially have the actual data size be relatively small compared to the allocated space. Here, we increase robustness (and security) by taking a hit to total space allocated.

b. In the case where having a fixed size would not work (perhaps as required by a project), we would need to assume the data in the struct was properly terminated at some time before being sent to merge. We cannot know if a string is properly terminated.

   i. A strlen on a non-terminated string would have undefined results, most likely a seg fault would occur, but a null value might be found and a widely incorrect length returned. This incorrect length could then make other parts of a program behave strangely (loops might run past the end of arrays, etc...).