Robust Programming for the New Programmer at Cal Poly

Tanya N. Luthin

June 25,2012

1 Introduction

Recently there has been an increase in news stories regarding failures of security, including releasing confidential customer information. This and the cost of a security breach has led companies to place more of an emphasis on computer security. Colleges must react to this by incorporating computer security into school curriculum.

A major part of computer security is robust programming, which means ensuring that programs do exactly what they are supposed to do and nothing else. Programs must not only do what is required, but they also must be well-written and hard to manipulate both erroneously and maliciously. A good example of a market where this is necessary is the mobile market, where a wealth of personal information is accessible and program hacks are expected to continue to rise. If robust programming was more of a focus in school, we would not see so many problems in the news.

1.1 Robust Programming at Cal Poly

It is unfortunate that robust programming is not a focus of the curriculum taught here at Cal Poly because of its importance in the business world. As the curriculum is now, students don't encounter anything related to computer security or robust programming until almost ready to graduate, and even then it's not a part of the required curriculum. Computer Science (CSC) students see security-related topics more than Computer Engineering (CPE) students, with a couple classes focusing on code reviews and efficiency. Only those with a specific interest in computer security take a special computer security course, which is not offered consistently each year, making it difficult for most programmers to become aware of this important issue. My colleague, Eric Gustafson, wrote in his paper that most CSC faculty believed that robust programming needs more of an emphasis[2]. It is unacceptable; all programmers should have exposure to robust programming. I'd like to examine a possible solution; incorporation of robust programming training into the introductory-level programming course at Cal Poly.

Work on this topic began with Matt Bishop's[1] experiment, discussed in the Background section of this paper, in which he asked students of the computer security course to write a simple program robustly. This experiment was repeated by my colleague at Cal Poly[2]. The results for both experiments were not what one would expect from a group of students with an interest in security. Their code contained well-known flaws that were also discussed during class lectures. Even after a clinic discussing program flaws, students still turned in code containing many of the same flaws as before the clinic, although there was a significant reduction.

If students interested enough in computer security to take a specialized class cannot demonstrate robust programming, what exactly is the underlying problem? Does the current method allow students to disregard security for too long before we attempt to teach them? I believe that if we build good habits as students learn programming skills, it will lead to higher code quality with a greater focus on computer security. This should eventually lead to a more secure digital world as students who are taught robust programming

techniques enter the job market and develop programs we use every day. That is why I wanted to run this experiment in the introductory-level class at Cal Poly. I have shown that it is worthwhile to teach students certain methods of robust programming as they are learning, and I would like to see robust programming permanently added to the curriculum of less-experienced students.

2 Experiment: The Robust Programming Clinic for CPE 101

My senior project was based on the experiments of Matt Bishop and Eric Gustafson[1][2]. Below is some information regarding those experiments and my experiment.

2.1 Background

Matt Bishop of UC Davis conducted an experiment in 2009 to test the effectiveness of a robust programming clinic in educating students about robust programming[1]. The experiment was conducted with a class of undergraduate students enrolled in the computer security class at UC Davis. The students were asked to write a program called 'mksuid', which asks the user for a password, verifies it with the system's password file, and check's permissions on a file in the user's directory called 'sniff'. Then, the program would set the SetUID bit of 'sniff'. The program is considered simple to write; however, the real challenge is to write this program securely. Some of the things students had to consider were sensitive user input, file permissions, and race conditions. After the clinic, most errors were significantly reduced.

This experiment was repeated last year at Cal Poly by Eric Gustafson[2]. Students in Cal Poly's computer security class were asked to write the same program after several lectures on robust programming. Gustafson's results closely resembled those of Bishop's experiment (see Table 1). Minor differences can be accounted for by different student backgrounds, but the general conclusion was that students were not great at avoiding program flaws, despite awareness and interest in security.

The fact that similar results were achieved from two different schools, both from security students, signifies a much deeper problem than lack of awareness. This problem likely stems from a lack of focus in computer security in every other programming course during the educational process. For instance, in my programming experience at Cal Poly, most students (myself included) turn in programs that work for only the situation given in the program specification and nothing else. In most assignments, students are told to assume that user inputs will be valid. Some assignments even come with test cases that the instructor will use to grade the assignment, so why bother writing in additional checks if you already know the situations to be prepared for? And even if ysuch assumptions weren't possible, students are already rushed to learn the material for any given class at Cal Poly and turn in incomplete solutions or make assumptions anyways.

Table 1: Summary of results from Gustafson's experiment

Error	*Before Clinic*	*After Clinic*
`Stat/chown` race condition	17	2
Unsafe function call(`strcpy`,`strcat`,etc)	9	6
Format string vulnerability	3	0
Unnecessary code	10	9
Failure to zero out password	11	0
Failure to sanity-check file modification time	14	6
(Total class size: 17 students; defects reported once per student)		

In real life, it is never the case that you can assume valid user input and ignore computer security. Most widely-distributed programs must account for not only user error but also malicious users. If students were always required to validate user inputs for all assignments, it would become second-nature and students could then concern themselves with more complex and/or subtle program flaws. While this does add to requirements, if the educational system had focused on robust programming sooner, maybe there would be less in the news about accidental security breaches. It will be a challenge to incorporate additional material into the lectures, but it will be worthwhile in the long run and will make Cal Poly students more valuable in the job market.

2.2 Our Experiment

The growing importance of computer security is why this experiment is designed to introduce robust programming to beginning programmers. It is slightly different than the experiments ran by Bishop and Gustafson, but follows their overall structure. Because the results of these experiments indicated a deeper problem, I wanted to run an experiment on robust programming in the first programming class at Cal Poly, CPE 101. Many students in this class are learning programming for the first time, making this the ideal opportunity to teach them security as they are learning. While this is ideal for the experiment, there are a few considerations. The first is that students need an understanding of basic programming concepts before introducing security. For instance, you need to know what a function is and how to call it before you can consider detecting error cases after calling it. The second consideration is that the first programming class is already full with material to prepare students for the next class in the sequence, and since students are learning most things for the first time, it is important not to overwhelm them with new material in such a short period of time. This is a challenge that can be resolved by teaching topics already in the curriculum with a security-related twist, and it is so essential to emphasize computer security that it is worth the challenge.

The nature of an introductory class required a few modifications to the original exper-

iment. Because the 101 class already has so much material to cover, I was only given a week of lab periods in which to run the experiment instead of as an actual class assignment. This, in addition to the fact that students still had assignments from the lecture portion of the class at the same time, may have negatively impacted the results. Furthermore, the assignments that were part of the experiment were only about 2% of the overall class grade, meaning that most students could choose not to participate at all and it would have little academic impact. In addition to all of the above obstacles, the only practical way to grade an experimental assignment is to grade by participation, so students could pretend to participate while not actually having put forth any effort. See the Results section for a more complete analysis.

The second major change was the actual assignment. The original assignment, 'mksuid', would be much too difficult for beginning programmers, requiring advanced functions to complete the assignment. For this experiment, I assigned a program simple enough for beginners to write and understand, but would still give an opportunity to demonstrate potential security flaws in the solutions. The details of the assignment will be discussed in later sections. The final major change was that, since the class has not covered security at all, I needed to host a special security lecture during the lab period to cover the basics of security and why it is important since most students did not know much on the topic.

2.2.1 Lecture

The lecture on security was fairly successful. Although attendance was low, this convinced the few students that did show up to pay attention since it would be more obvious if they were not making eye contact. I attempted to create an interactive lecture by asking questions about current events and allowing them to help point out potential security violations in a given situation. Class participation remained low despite my efforts to get them involved. This could be due to a lack of experience teaching a class on my part, but it could also be related to student background. Students taking this course during spring quarter tend to be non-majors (meaning not CSC or CPE) and thus the topics are less familiar and less interesting to them. The majors of the students are shown in Table 2. 32 out of the 36 students in the class would be considered non-majors, approximately 89% of the class. In my experience, many students treat non-major courses as a task that needs to be completed to fulfill a requirement, and not something they are willing to put extra effort into. This means that students came in with even less background knowledge in related topics, explaining why students were not familiar with the security-related current events discussed in class.

At the end of the lecture, I allowed the students to ask questions about both the lecture and the assignment. Not many questions came up and I'm not sure if they were overwhelmed, didn't know what to ask, or weren't interested. For the next two meetings, I attended their lab sessions where students were allowed to work in class and ask me questions about the assignment. I tried not to give away answers, but I did not want to mislead the students or refuse to help those who were genuinely lost.

Table 2: Number of students in each major for this CPE 101 Course.

Major	Number of Students
Electrical Engineering	19
Biomedical Engineering	4
General Engineering	2
Math	2
Business	2
Computer Engineering	2
Computer Science	2
Statistics	2
Graphic Communication	1

2.2.2 Lab Assignment

A simpler program was needed to accurately test the intro-level students and allow them to focus on learning security rather than learning new programming techniques in addition. The program was to write a simple caesar cipher, a type of substitution cipher that simply shifts the alphabet a given number of characters and does a direct replacement. The program summary is shown in Appendix B. I tried to keep the program functionality very simple, and I even provided a partial solution to the shifting calculation after the clinic because the goal was not to make a computationally complex assignment, but instead to allow students to practice robust programming.

3 Results and Analysis

Below is a summary of the program flaws I was looking for in the assignment and a brief explanation:

Buffer overflow risk: Most students used the function `scanf` to read in from files and from the user. When using `%s` in the string format, this does not limit the number of characters read in to the size of the buffer that they will be stored into. This can cause a buffer overflow, either malicious or accidental. This is bad because it can allow a malicious programmer to manipulate the program and inject arbitrary code. Ways to avoid this include using `fgetc` (in a loop), `fread` (which limits the number of characters read), `fgets` (which also limits characters read), and `scanf` with `%99s` (which limits characters read to the number before the 's'). This can also occur when the string functions like `strcpy` are used, which have no size limits on the buffer used for the new string. These can be replaced by functions like `strncpy`, which does limit the buffer size, or `memcpy`, which works for data types other than strings.

Null-termination on input or output: I noticed this issue when students read the input and/or stored the output without ensuring that it was null-terminated. Most functions do not guarantee a null at the end of reading characters from a string, especially if a null character was not read and/or if the maximum number of characters was read. This causes an issue when the input/output is tested as if it were, such as `while(input != '\0')`, which may or may not occur. This means that under some circumstances, the program may read past the end of the input, likely causing a segmentation fault at best. Ways to avoid this include using a function that guarantees null termination (i.e. `fgets`) or manually null-terminating the 'string'.

Input sanitation observed in a few different cases: Input should be checked to ensure that it is valid. This is in accordance with Principle of Complete Mediation and Principle of Fail-safe Defaults, two of the famous security principles written by Saltzer and Schroeder[3] (see Appendix A)

Reading Integers: I saw this issue when students read in the shift size. Most students used the function `atoi`, which performs the desired operation, but the error condition is not always detectable. Students could avoid this issue either by disallowing zero as a shift size, or by using the function `strtol`, which has more robust error detection. A few students used `scanf` with `%d` in the string format, which should return an error value if a proper integer is not read; however, not one student checked the return value.

Input Data: This issue was observed when dealing with the input to be encrypted. Students were required to encrypt alphabetic characters only and all other characters needed to pass straight through. Once encrypted, alphabetic characters were required to be in uppercase. If students did not convert to uppercase or check for alphabetic characters, it could cause unpredictable run-time bugs.

Validating Shift Size: Large shift sizes were sometimes not disallowed or accounted for. For instance, if the shift size was larger than the size of the alphabet and if it was not disallowed when read, it must be divided using something like the modulus operator to ensure that the output remains within valid alphabetic characters.

Opening Files with Wrong Mode: This was an extremely uncommon error, which is a good thing! Students should open the input file for reading and the output file for writing. Any additional permissions are unacceptable in robust programming, which is consistent with the Principle of Least Privilege, one of the famous security principles written by Saltzer and Schroeder[3] (see Appendix A). My speculation is that the one student demonstrating this issue had a simple typo.

Input and/or Output Streams Not Closed: This issue is demonstrated in two ways. The first way was forgetting to close the stream before the program closed with a function similar to `fclose`. The second way this issue appeared was when students

used `return` when an error was encountered instead of calling `exit`, which closes streams for you before closing the program, whereas return may not necessarily close streams. It is a bad idea to leave streams open for potential misuse.

Function not error checked: This happens when a return value from a function call is not used to see if an error was encountered. I tried to take note of when students did this at all, even if inconsistently. This showed awareness of the issue even if the students were unsuccessful with the implementation.

Overflow the Alphabet: This was observed when students did not realize that the shift size would place their output past the letter 'Z' when calculating the encrypted message. This could even result in overflowing a character variable. Something like the modulus operator would help avoid this issue. This is different than input sanitation because this can happen with a perfectly valid shift size, although both errors can be resolved in the same line of code.

Magic Numbers: Any numbers that are not plainly obvious at first glance should be defined at the top using the `#define` functionality of C programming. This makes the reasoning behind code more obvious, and also makes changes easy since you can change it in one place for the whole program.

Extra code: In general, it is a bad idea to have extra code because it makes programs harder to debug. Extra code is not by itself a security flaw, but it definitely plays a role in creating them. This error appeared in a couple different ways.

Defined an Alphabet: This was observed in students' programs when they defined their own alphabet instead of using the ASCII values.

Repeated Code Segments: This was observed when students wrote the program over again for each case of program command-line arguments. This could be avoided by writing and calling a function, or writing the code only once after dealing with command-line arguments. This is dangerous because if a bug appears in one code segment, it could cause hard-to-detect bugs if the programmer doesn't remember to fix it in both locations.

Plaintext Remains in Memory: This was a problem because the plaintext is supposedly a secret, which is why you would want to encrypt it. Leaving it in memory when the program closes is a bad idea because a malicious person could obtain this before the operating system gets around to clearing out memory. Students could deal with this by not buffering the input or clearing the memory by overwriting the plaintext buffer with all zeros at the end of the program. I also recommended to students to clear the shift size since this could give a malicious person a good idea of what the program was doing.

Printing the plaintext: This was a bad idea, and I believe it was mostly a misunderstanding of the purpose of the program. Again, the plaintext is supposed to be a

secret and should not be printed by the program. I was surprised how often this occurred. This was either because the students misunderstood the sensitivity of the plaintext, or print statements were accidentally left in from prior debugging.

3.1 Pre-Clinic Results

The turnout for the assignment before the clinic was higher than expected despite the relatively low impact of the assignment on the grade. 28 out of 36 students turned in the assignment. Code demonstrated poor quality and little consideration for security. It was obvious that students either did not think much about robust programming, ran out of time, or just did not know how to go about programming robustly. On the last lab period before the assignment was due, I observed a significant increase in questions regarding the assignment. Most of the questions indicated that students, in general, had not worked on the assignment much in advance. A few reasons for this may include busy schedules outside the class, assignments from the lecture portion of the class, and the fact that previous lab assignments were relatively simple and required less time to write. Questions also indicated that I was asking students to learn to use a significant number of functions they have never seen before. It is more difficult to learn a new programming technique at the same time as learning new functions. Although it was possible to write a robust solution using just a few new functions, the best solutions used the functions I suggested.

Most submissions contained every flaw on the above list of program flaws, and almost all of them were missing functionality required by the program specification. Few students demonstrated awareness of program flaws and all programs demonstrated at least one flaw, despite my lecture and resources given. Even if students attempted robust programming techniques, the implementation was poor, mostly incomplete and many times broken. Some students may have gotten ideas from the answers given to questions in class. Many of the submissions indicated a severe lack of understanding in computer programming. This is probably a result of student backgrounds. For instance, a biomedical student I helped during class, despite her wealth of knowledge, was not aware of the ASCII table or how computers treat characters internally. Many students did not understand binary and were only aware of its existence. Had the class contained more CSC/CPE majors, code quality may have been higher because those students tend to have more experience outside of the college in the subject matter, and would value the topics more because they expect to learn more on the subject later in their college careers.

3.2 Post-Clinic Results

After the first submission, I held a robust programming clinic to discuss with the students which flaws were demonstrated in their programs, why it is a problem, and how to fix it. The clinic was intended to be in person, but due to lack of time in the 101 class, I ended up holding it via email. The class indicated that this was an acceptable way to communicate. In each email, I was sure to say that I was available to meet with them in person if they did not understand or needed additional help. I felt that this was necessary given the varying

Table 3: Errors present in student submissions before the robust programming clinic. Those listed under "Awareness Shown" are those who had the error present in one place, but may have fixed it in another, showing awareness of the issue.

Error	*Error Present*	*Awareness Shown*
Buffer overflow risk	25	4
Null termination issue	19	2
Open streams with wrong mode	1	1
Input sanitation issue	28	17
Function error checking issue	28	7
Open file stream on close	19	7
Magic numbers	24	20
Overflow the alphabet	20	10
Extra code	14	n/a
Incomplete solution	27	n/a
Plaintext in memory on close	28	n/a
Printed plaintext	8	n/a
(Total class size: 28 students. Defects reported once per student.)		

backgrounds and experience levels of the introductory class. I did meet with a few students in person to help them implement my recommendations during the clinic. I observed a general lack of understanding of programming, but they did seem to care. Once I explained the issues to them, they seemed to understand and were able to continue on the assignment.

The turnout for the post-clinic assignment was significantly lower. All 28 students, plus one additional student, turned in a file. The problem was that only 8 students actually made changes to the file (as determined with diff between the old file and the new file, since only one student actually commented the 'last-modified' date), despite a full lab of students who appeared to be working on the assignment before the due date. It seemed that the remaining students turned in the same file that was submitted before the clinic, indicating a low priority for this assignment, likely due to the fact that students had other classes to worry about. It was also the last week of the quarter and students were very busy in the lecture portion of the class. The instructor indicated that his final assignment was also due later in the same day that this assignment was due. He believed that most students would strategize their time, placing the other assignment at a higher priority due to the impact on their grade. I will analyze the results of those students who submitted both before and after the clinic separate from the rest of the class in order to control for some of the external circumstances that may have impacted the results despite the fact that this creates a relatively low sample size. Results are summarized in Tables 4 and 5.

Most of the students who participated in both parts of the assignment showed awareness of each flaw discussed during the clinic. I saw a significant reduction in these flaws; however, many solutions did not completely eliminate them. For instance, one program showed an elimination of a buffer overflow risk when reading input from the file, but the risk remained in a different location in the program, such as when prompting the user for the filename. I called this showing awareness of the issue, because it was obvious that students attempted to fix the problem, but did not realize that the problem existed elsewhere in their program. Not one student had a fully robust solution, but the expectations of robust programming might be too high for a busy beginning programmer. One student's submission was extremely close, with only one function return value ignored. This is insignificant and would likely be caught by a co-worker if this were a real work environment. Another student turned in a program with significant flaw reduction, although imperfect still. It is likely that, given some additional time, most students could have produced a complete solution.

It is worth noting that my expectations may be high, but there are significant learning opportunities for the new programmer when it comes to computer security. It is also noteworthy that all of the students I met with in person after the clinic were included in the few students who submitted an improved post-clinic assignment. This shows that those who cared enough or had enough time to work on the assignment were able to significantly improve their program quality and robustness. This means that it is indeed possible to start teaching robust programming to students as they are still learning to program. The significant reduction in errors of the students that participated fully shows that robust programming training is important and can be effective. For every flaw discussed during the clinic, awareness improved even though the implementation was imperfect. Complete

reduction in errors was not seen for any given flaw, but good attempts were made by those who tried.

Table 4: Errors present in student submissions before the robust programming clinic controlling for those who participated in the second submission. Those listed under "Awareness Shown" are those who had the error present in one place, but may have fixed it in another, showing awareness of the issue.

Error	*Error Present*	*Awareness Shown*
Buffer overflow risk	8	2
Null termination issue	8	1
Open streams with wrong mode	0	0
Input sanitation issue	8	6
Function error checking issue	8	3
Open file stream on close	5	1
Magic numbers	6	4
Overflow the alphabet	6	3
Extra code	2	n/a
Incomplete solution	7	n/a
Plaintext in memory on close	8	n/a
Printed plaintext	4	n/a
(Total sample size: 8 students. Defects reported once per student.)		

4 Conclusions

This experiment has shown that it is valuable to begin teaching robust programming and computer security to new programmers as they are still learning new techniques. While some program flaws may be too complex for them to understand or too difficult for them to realize, many program flaws can easily be corrected by students with this education level. Even though the environment for this experiment was not ideal, it demonstrate the capabilities of those students who cared enough and/or had enough time to make a good attempt. I believe that this shows that if this assignment were a larger part of their grade and if computer security was a larger part of the curriculum, students would successfully produce robust programs.

Of course it would be preferred to run the experiment with a class of CSC and CPE majors; however, that was not possible in the quarter for this project. It would also increase effort put forth by the students if this were a class assignment instead of a lab assignment, but considering these circumstances, the turnout was significant. The fact that even one

Table 5: Errors present in student submissions after the robust programming clinic controlling for those who participated in the second submission. Those listed under "Awareness Shown" are those who had the error present in one place, but may have fixed it in another, showing awareness of the issue.

Error	*Error Present*	*Awareness Shown*
Buffer overflow risk	6	5
Null termination issue	6	1
Open streams with wrong mode	0	0
Input sanitation issue	7	7
Function error checking issue	7	4
Open file stream on close	4	1
Magic numbers	5	5
Overflow the alphabet	6	6
Extra code	2	n/a
Incomplete solution	5	n/a
Plaintext in memory on close	3	n/a
Printed plaintext	1	n/a
(Total sample size: 8 students. Defects reported once per student.)		

student, and in this case eight, cared enough to make a second attempt despite its relatively low input to the grade means that this is important to students. One solution had even better design than my own solution that I produced. These students are capable of learning the subject, but a little more motivation is needed.

5 Future Work

The next steps for this project would be to run this experiment again, but with more of an impact on the students' grades in the class. With the thread of course grades, effort levels would likely increase. I would love to see the results of such an experiment, especially in a class with CPE and CSC majors, the people expected to do best at and who have the most to gain from such an experiment. This could lead to full incorporation of robust programming training into the curriculum at Cal Poly, and eventually a decrease in program flaws once students move out into the business world. My hopes are that another Cal Poly student will take on this work for a senior project or a Masters thesis and that Cal Poly does improve the programming curriculum and includes robust programming.

References

[1] D.A. Bishop M.; Frincke. Teaching secure programming. *Security & Privacy, IEEE*, 3 and 5:5–56, Sept.-Oct. 2005.

[2] Eric. D. Gustafson and Phillip L. Nico. Toward robust programming for the masses. *Proceedings of the 2012 ASEE PSW Section Conference*, 2012.

[3] Jerry H. Saltzer and Mike D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63:1278–1308, September 1975.

A Saltzer's and Schroeder's Design Principles

Saltzer's and Schroeder's Design Principles

Principle of Economy of Mechanism

The protection mechanism should have a simple and small design.

- Reduce the potential for error in the implementation.
- Make it as simple as possible, but no simpler.

Principle of Fail-safe Defaults

The protection mechanism should deny access by default, and grant access only when explicit permission exists.

- Default state should be no access
- Allow only the few things we want to happen rather than trying to disallow other things. (“Don’t do anything unusual”)

Principle of Complete Mediation

The protection mechanism should check every access to every object.

- The primary underpinning of the protection system.
- A foolproof method for identifying the source of every requests
- Caching authority must be suspect

Principle of Open Design

The protection mechanism should not depend on attackers being ignorant of its design to succeed. It may however be based on the attacker's ignorance of specific information such as passwords or cipher keys.

- Put all your eggs in one basket, and watch that basket.
- More to the point, it restricts the size of the secret and makes it easier to keep.

Principle of Separation of Privilege

The protection mechanism should grant access based on more than one piece of information.

- Two-factor authentication.
- Loss of a single item will not compromise the entire system.

Principle of Least Privilege

The protection mechanism should force every process to operate with the minimum privileges needed to perform its task.

- Designed to protect against accident. (e.g. “Need to know”)

Principle of Least Common Mechanism

The protection mechanism should be shared as little as possible among users.

- Every shared mechanism represents a potential information path. (physical or logical separation reduces the risk of sharing.)
- Any mechanism serving all users must be certified to the satisfaction of all.

Principle of Psychological Acceptability

The protection mechanism should be easy to use (at least as easy as not using it).

- Users are lazy
- If the user’s mental image of what is to be done matches the actions fewer mistakes will be made.

B Assignment

CSC 101 Lab Weeks 8 and 9
Security Lab

ISSUED: Friday, 18 May 2012
DUE: Lab 8: Wednesday 23 May 2012, by the end of lab
Feedback: Friday 25 May, during lab
POINTS POSSIBLE: 1
WEIGHT: 1% of total class grade

Specification of Program Caesar

This assignment requires you to write a program that will encipher text using the basic Caesar cipher. This cipher is fairly simple to implement, but the point of the assignment is to write code that is as robust as possible.

The Caesar cipher is an alphabetic shift cipher, which is a type of substitution cipher. This is the simplest and most widely known cipher. It works simply by shifting the alphabet by a specific shift size so that with a shift of 4, an 'a' in the plaintext would be replaced by a 'd' in the ciphertext.

Example:

```
Shift size: 6
Plaintext (P):    TOBEORNOTTOBE
Ciphertext (C):   ZUHKUXTUZZUHK
```

Details of the Program

- Caesar takes a number representing the shift size to encipher with, and input and output filenames. If a dash is present for either filename, stdin or stdout is used (as appropriate). If there are no command-line arguments, the program will read them from standard in.
  `Usage: caesar shift infile outfile`
- You must store these three command-line arguments in a struct. The definition details of the struct are up to you.
- Caesar will encipher its entire input until it reaches the maximum number of characters, which should be defined as 100.
- Caesar must check its command-line options for validity. If they are invalid e.g. a shift that is not a number it should print a usage message and terminate gracefully with nonzero status.
- If caesar succeeds, its exit status should be zero. If unsuccessful, it should be nonzero.
- The cipher only operates on normal alphabetic characters ("A-Za-z"), and all lower-case letters are mapped to upper-case.
- Anything that is not "A-Za-z" in the input is passed through unchanged. (Were this a real enciphering tool, these characters would either be enciphered, too, or dropped.)

Sample Runs

If there are no command-line arguments, then the program will prompt for and read three values from stdin, like you did in lab 7. For example, here's what it looks like with three command-line args:

```
caesar 6 intext ciphertext
```

where "`intext`" and "`ciphertext`" are the names of files. If you run the program without command-line arguments, then it will look like this:

```
caesar
Shift Size: 6
Plain Text File: intext
Cipher Text File: ciphertext
```

Again, this is just like what you did in lab 7, where the program either reads command-line args, or inputs from stdin if there are no command-line args.

Resources

1. Useful library functions, constants, and variables:
   - `int isalpha(int c)` -- defined in ctype.h
   - `int toupper(int c)` -- defined in ctype.h
   - `exit` -- defined in stdlib.h
   - `FILE* fopen(char* filename, char* mode)` -- defined in stdio.h; used in lab 5
   - `FILE* fdopen(int filedes, char* mode)` -- defined in stdio.h; like fopen but uses file descriptor instead of string filename
   - `STDIN_FILENO`, `STDOUT_FILENO` - file descriptor constants defined in unistd.h; use as first argument to `fdopen`
   - `void perror(char* s)` -- defined in stdio.h
   - `errno` -- library variable defined in stdio.h, and used in conjunction with the `perror` function

2. You can read the UNIX man pages for any of the preceding functions. For example, to read the man page for `perror`, type the following on a terminal:

   ```
   man perror
   ```

   You can read man pages in emacs by typing `<escape>x man`, and then entering the name of the man page after the "`Manual entry:`" prompt.

3. Some resources on robust programming will be discussed during the Friday lab presentation, and you can do your own internet search on the subject. Here are some specific suggestions from Tanya:
   - Robust programming: http://nob.cs.ucdavis.edu/clinic/
   - Bruce Schneier Blog: http://www.schneier.com
   - Read the news!

Collaboration

Unlike previous labs, you will work individually on labs 8 and 9.

Submitting and Receiving Feedback on Your Work

On or before the end of lab on Wednesday May 23, submit your work as follows:

```
handin gfisher 101_lab8 caesar.c
```

During the lab on Friday May 25, you will meet with security clinician Tanya to examine potential flaws in robustness and what to do to fix the flaws. Remember, you are trying to catch these things before she does, but the important lesson to learn is how to fix them. You will be graded on participation in the clinic and completeness of the second version of the program, which will be the deliverable for Lab 9.

Lab Schedule for Weeks 8 through 10

- ***Friday 18 May:*** lecture on security, from Tanya Luthin
- ***Monday and Wednesday, 21 and 23 May:*** work on caesar, and ask Tanya questions
- ***Friday 25 May:*** Feedback from Tanya, and assignment of work for Lab 9
- ***Monday 28 May:*** holiday
- ***Wednesday 30 May:*** lab quiz
- ***Friday 1 June:*** finish lab 9