International Journal of Advanced Computer Science, Vol. 1, No. 1, Pp. 1-8, Jan. 2011.

Incorporating the Security Mindset into Introductory Programming Courses

Vahab Pournaghshband

Manuscript
Received: *2013*
Revised: *2013*
Accepted:

Published:

Keywords
CS 1, Computer Security, C++, Security Mindset, Security Awareness

Abstract— The inherent problems of computer security are becoming increasingly important, and it is critical that our students gain the necessary skills and knowledge, early in their academic programs, to handle these problems. Specifically, the lack of security mindset is responsible for many overlooked and exploitable security bugs in the computer programs that these students design. While learning the security concepts generally requires a more advanced knowledge of computer science, learning the security mindset can be, and should be, addressed as early as CS 1. Although the primary focus of any traditional CS 1 course is that of basic programming concepts, we believe that teaching the security mindset in this course is valuable and effective. In this paper we discuss the course that we have taught for four terms—an introductory course that teaches the security mindset to beginner programmers. We start out by using the term-long incremental development of a security-sensitive program—the login program. Students develop the security mindset by thinking as both hackers and defenders, in order to catch and fix the logical and run-time errors that may lead to security breaches in the program.

1. Introduction

Over the past several decades, software has become a crucial element in our lives, allowing us to manage and control systems that provide critical infrastructures in areas such as communications, energy, and transportation. Unfortunately, hackers are capable of interrupting these connections by exploiting software vulnerabilities and breaching software security. For instance, in 2011 programming flaws allowed hackers to steal millions of dollars through stolen credit cards in a single cyber attack incident [3]. Very recently, a flaw in Java put millions of Windows and Mac users worldwide at risk, and the damage cost is still rising [6]. New threats are further emerging as computers become more embedded and more intimately into our environment and daily lives, e.g., recent security vulnerabilities found in mobile medical devices [14]. Flaws in programs have also been exploited (e.g., by Stuxnet worm [11]) to sabotage critical infrastructures as an act of cyberwarfare.

Software security vulnerabilities are the most common cause of software security breaches. Most security problems can be traced back to underlying errors in a program's source code. For example, 64% of the nearly 2,500 vulnerabilities in the National Vulnerability Database in 2004 were caused by programming errors [7]. Fixing these bugs through constant security patches has its own problems, since security patch management and distribution are known to be fairly ineffective [2]. This suggests that catching security bugs in the pre-release version of programs is what is most effective.

These errors exist in programs mostly because programmers often fail to notice how the programs might fail and how those failures might be exploited. Among the numerous factors that explain this problem, the lack of a security mindset plays an important role in being unable to spot the exploitable bugs in programs in the early stages of development.

The following is an excerpt from Bruce Schneier's blog on the security mindset [15]:

"Security requires a particular mindset. Security professionals - at least the good ones - see the world differently...This kind of thinking is not natural for most people. It's not natural for engineers. Good engineering involves thinking about how things can be made to work; the security mindset involves thinking about how things can be made to fail. It involves thinking like an attacker, an adversary or a criminal. You don't have to exploit the vulnerabilities you find, but if you don't see the world that way, you'll never notice most security problems."

Therefore, if we teach programmers the security mindset, it will go a long way toward making a world with fewer cyber attacks and will substantially improve the security of future technological systems.

We believe that this security mindset should be taught to beginning programmers in classes as basic as CS 1, and that it should be emphasized throughout a student's

This work was partially supported by Intel.

Vahab Pournaghshband, University of California, Los Angeles, vahab@cs.ucla.edu.

This paper is an extension of the original paper [16].

Digital Object Identifier ??.????/j.issn.????-????.????.??.???

computer science undergraduate program. In this paper we illustrate how to develop and teach this mindset in a CS introductory course. Through an example, we will show how to effectively teach the basic programming concepts, which is the primary goal of any CS introductory course, while developing the necessary security mindset for the students. The rationale for teaching the security mindset in an introductory course is elaborated on in Section 2.

This paper is organized as follows: Section 2 presents the advantages offered by this course. Section 3 presents related work. Methodology is presented in Section 4. Section 5 presents an informal evaluation of our work. Section 6 is future work and Section 7 concludes this paper.

2. Advantages

In this section we discuss why it is important to teach the security mindset in the first CS introductory course, while still keeping the primary focus on teaching basic programming concepts.

1) An early exposure to security issues is essential to a student's foundational appreciation and understanding of computer security. If the security mindset is developed at an early stage, students are more prone to naturally have security in mind while programming. From a practical point of view, this is a more effective approach than the one that only emphasizes program correctness and then, later in a student's programming career, attempts to change the undesirable habit of overlooking security bugs.

2) Students will also learn the significance of security bugs by understanding how these minor inaccuracies in detail, if overlooked, can lead to security breaches even though their code might be free of syntax, logical, and run-time errors. Moreover, it forces students to realize the importance of programming mistakes that can result in minor logical or run-time errors. Students will realize that these errors not only cause simple mistakes in program correctness, but they create an opportunity for hackers to attack the systems that run their programs.

3) Our proposed curriculum changes in the introductory course still emphasize learning the basic concepts of programming. By incorporating simple attack/defense scenarios to teach and verify program correctness interactively, we make learning those concepts more attractive to our students. As Fanelli et al. [5] state, "Security can make the *other stuff* more interesting. Studying security can lead students to a deeper understanding of computer science and information technology concepts. In many cases a thorough understanding of how a program works is needed to effectively attack or defend it."

4) Many students outside of the CS major are required to take only CS 1 (and/or CS 2), and quite possibly will never take any other CS course beyond the introductory courses. Therefore, it might be their last chance (within the curriculum) to learn about computer security. Even if a student never pursues a programming career, or programs only infrequently, this is still an opportunity for him to learn about the technical aspects of computer security. Students will understand the root cause of known security problems, why they behave maliciously, and how to protect themselves from them effectively. For instance, once students learn about how powerful brute force attacks are, they will understand the importance of choosing a strong password. In any and all cases, "The security mindset is a valuable skill that everyone can benefit from, regardless of career path." [15]

5) Secure programming takes extensive practice in order for it to evolve into a skill. While gaining this knowledge is valuable, a single course in computer security in the undergraduate curriculum often fails to actually build this skill for students. This suggests that exposing students to security issues, as early as an introductory course, establishes the security mindset and provides more practice in secure programming; hence, skill will be built faster and better.

3. Related Work

An extensive body of literature has been created that focuses on computer security education. Some methods use active learning and visualization tools to teach security effectively. As an example, IPsecLite [8] was developed as a tool to demonstrate the inner-networking of IP security standards. Other methods emphasize the importance of teaching the physical and social aspects of computer security, along with technical aspects [4, 9]. A different class of work in this area, such as Peterson et al. [13], has developed hands-on laboratory exercises for a term-long security course.

There have also been proposals that emphasize introducing security concepts in the CS 1 course [10]. Validating user input, array range checking, numeric overflow and underflow, operator precedence, and rounding errors are a number of concepts that are suggested for course material in CS 1. The list is, however, not very long due to the limited knowledge of students in a CS introductory class.

Finally, there have been several papers on how security education fits into the CS undergraduate curriculum. Some

introduced various ways to teach a single course [12]. Some presented effective track approaches, where a sequence of specialized courses on security is offered [1]. And some support the thread approach that is used as a unifying theme across the standard core CS curriculum [12].

4. Methodology

In this section we present the methods used to teach this introductory course at UCLA. C++ is used in the introductory CS course sequence. The material we present here, however, can be applied to other programming languages with some modifications, except for some topics such as pointers and C strings that are unique to a specific set of languages.

In our approach we used a widely known practical program as our example— the login program. Our login program asks for username and password from the user and will reveal a secret word if the credentials are provided correctly. It generates an error message if the username or the password entered is invalid. This program evolved throughout the school term, as the students learned new concepts. Note that traditional examples were still used to teach the materials, in addition to examples relevant to our login program case study.

We used the login program for teaching purposes for various reasons: (1) the basic concept of the program is familiar to students since it is a program that students use multiple times in an average day, ranging from logging into the department laboratory systems, to Facebook, and e-mail accounts. This would also make interacting with the program interesting, whether one was hacking into it or preventing hackers from breaching it. (2) This program has the unique feature of being able to vary from being a very simple program to a complex one. (3) As more programming concepts are introduced in the course, the program can be incrementally built up from a simple program to a coherent complex program. (4) The program is security-sensitive by its nature, making defensive programming even more essential.

In many cases, a thorough understanding of how a program works is needed to effectively attack or defend it. Moreover, to be a good defender you must be a good attacker; hence, we constantly guide the students to wear a hacker's hat to spot the vulnerability, and then switch to a defender's hat to fix the problem. Note that we use the term “hacking” in the most non-pejorative sense possible. Hacking, for our purposes, is a process for pointing out programming bugs rather than any negative use of such skills. For instance, we present a faulty program containing a particular logical error. We then ask the students to spot the error and exploit that vulnerability as if they were hacking into the system. We then further ask them to fix the error in the program to protect it against that particular attack.

Below, we outline, in order, some of the concepts and approaches that we use to simultaneously introduce core programming ideas and to develop the security mindset.

A. Introduction and Variables

Early in the course, variables are introduced. For the login program, we discuss how to define and use variables to hold the username and the password. Integer overflow is also introduced, and is illustrated through an example involving simple arithmetic operations. However, an example of how integer overflow is exactly exploited will be presented later in the course when introducing dynamic memory allocation.

B. Conditional (if-else) Statements

A very simple login program is implemented. This program will only support a single user and only accepts an all-numeric (`int`) password. If the password matches a hardcoded password, the secret word will be revealed (Fig. 1).

```
if ( password==12345 )
   cout << "The secret word is Kosar.";
else
   cout << "Invalid password!";
```

Fig. 1 Our first login program.

Even in such a simple implementation, common beginner mistakes, like using the assignment operator “=” in the condition instead of the equality operator “==,” can be illustrated to students as an error that can lead to revealing the secret, even in the case of an incorrect entered password.

C. String Class

A major problem that variables of type `int` used to store passwords have is that they are vulnerable to brute force attacks due to relatively short range of values they can hold. Most platforms hold 4 Bytes for integers which leaves approximately 4 billion different possibilities. In modern typical machines it takes only a few seconds to crack the password when trying every single possible value. To alleviate this security problem, we change our program to use the C++ string class to store the password instead. Even so, our program is still vulnerable to dictionary attacks. This is because most users, if not all, do not choose a random set of characters as their password, and rather something they can remember. Thus, this enables the hackers to use dictionary attacks, which is to systematically try entering every word in a dictionary (or a sequence of them) as a password to guess the correct password.

D. Nested if-else and switch Statements

The program evolves into a two-user login program. We start our presentation with a faulty version of a two-user program (Fig. 2). We then lead the students into a

discussion of how this code is problematic, and examine the set of information that hackers would need in order to get into the system without entering the correct credentials. Finally, through a class discussion, we explain how to fix this problem by replacing the inner `||`'s by `&&`'s.

```
if ((username=="vahab" || password=="Ta#3rEh") ||
    (username=="peter" || password=="h0o$H@Ng"))
   cout << "The secret word is Kosar.";
else
   cout << "Invalid username and/or password!"
```

Fig. 2 An erroneous two-user login program.

Afterwards, this program is converted to a nested `if-else` statement, where outer `if` statements now check the usernames and inner `if` statements check the validity of the corresponding password. In this case, a more specific error message will be produced that indicates whether the username or the password entered is invalid. As this is being implemented, the emphasis will be on not only the possibility of errors in the code, but also the magnitude of problems caused by those errors. Ultimately, this program will be converted to use a `switch` statement in the same way.

E. Loops

The program shall be enhanced to allow multiple login attempts in such a way that it will quit after a predefined number of successive failed attempts. This illustrates a simple way to prevent brute force attacks, introduced earlier in the course. We also emphasize the magnitude of logical errors in the loop condition (Fig. 3). For instance, an infinite loop will give the hacker virtually an unbounded number of attempts.

```
int login_attempt = 0;

do{
    cin  >> username;
    cin  >> password;
    if (username == "vahab" && password == "cs31")
        cout << "The secret word is Kosar."
} while(++login_attempts < 3);
```

Fig. 3 Limiting login attempts to prevent brute force attacks.

F. Nested Loops

When choosing a new password, the program should determine if it is a legitimately strong password. In our example, with the help of nested loops, we will write code to validate the password strength. The restrictions that define the level of strength of passwords are arbitrary. In this example, we make a strength requirement that the password must be at least eight characters long, and must include at least one uppercase and one lowercase letter, and one digit. In addition, it must contain at least one string of at least four letters long. Again, failing to write this piece of code in the program correctly may result in accepting weak and vulnerable passwords. One variation of such implementation is presented in Fig. 4.

```
bool isStrongPassword(const char* password)
{
    if (strlen(password) < 8)
        return false;

    bool one upper = false;
    bool one lower = false;
    bool one digit = false;
    bool four_letters = false;
    int letter_count = 0;

    for(int i=0; password[i]!='\0'; i++)
    {
        if (letter_count > 3)
            four_letters = true;

        if (isdigit(password[i]))
            one digit = true;
        else if (isupper(password[i]))
        {
            one_upper = true;
            letter_count++;
            continue;
        }
        else if (islower(password[i]))
        {
            one_lower = true;
            letter_count++;
            continue;
        }

        letter_count = 0;
    }

    if (one upper && one lower && one digit &&
            four letters)
        return true;

    return false;
}
```

Fig. 4 Implementation of `isStrongPassword` to prevent users to choose weak passwords.

G. Functions

After familiarizing students with the concept of functions, we implement the following two functions as examples of how to modularize our login program using functions:

```
bool isStrongPassword(std::string);
int authenticate();
```

The first function reuses the code implemented in the nested loops section to validate the strength of passwords. The latter function abstracts away the entire authentication process by moving all the details of how it is done into the function `authenticate()`. This way, only a simple `if` statement is needed to decide on revealing the secret word:

```
if ( authenticate()==0 )
  cout << "The secret word is Kosar.";
```

The function has no argument. It returns a non-zero value only when authentication fails (1=excessive failed

attempts, and 2=user gave up trying prematurely).

And last, as we introduce some popular functions from the standard library, we mention the importance of knowing the difference between safe and unsafe functions.

H. Arrays and C Strings

In addition to learning and interacting with arrays, the students learn about C strings and some popular library functions associated to it. C strings can now replace the string class that was used as the type for the username and the password earlier in the course (`char username[SIZE];` `char password[SIZE];`). At this point we present the students with the code presented in Fig. 5 and ask them to examine it for potential problems.

```
char password[SIZE];
bool logged_in = false;

cin >> password;

if ( strcmp(password,correct password)==0 )
  logged in = true;
if ( logged_in==true )
  cout << "The secret word is Kosar.";
else
  cout << "Invalid password!";
```

Fig. 5 A login program that is prone to buffer overflow.

There is no logical error here but at run-time, entering a password larger than the length of the array is problematic. Technically, among numerous undesirable consequences that could happen in this case, crashing at run-time is most desirable, from the security point of view. We explain why by showing how this could be an opportunity for the attacker to successfully overwrite `logged_in` to `true`, bypassing the entire authentication process. Even worse, the attacker can exploit the buffer overflow vulnerability in this piece of code and gain full control of the system. The latter, however, would be only explained in a very high level discussion since the students are not expected to have any knowledge of the operating systems concepts required to fully understand the details of this effect.

I. Multidimensional Arrays

The program should now support multiple users. One way to store the required information could be as follows:

```
char credentials[2][MAX_NUM_USERS][SIZE];
```

In this case, the corresponding password for username `credentials[0][i]` is `credentials[1][i]` (As illustrated in Fig. 6).

J. Parallel Arrays

Our multiple-users login program will contain meta information about users in addition to their corresponding passwords: first name, last name, age, date of birth ("MMDDYY"), the security question and its corresponding answer, and whether the user has administrative privileges or not (Fig. 7). This is also a good way to teach array traversal and basic sorting to the students.

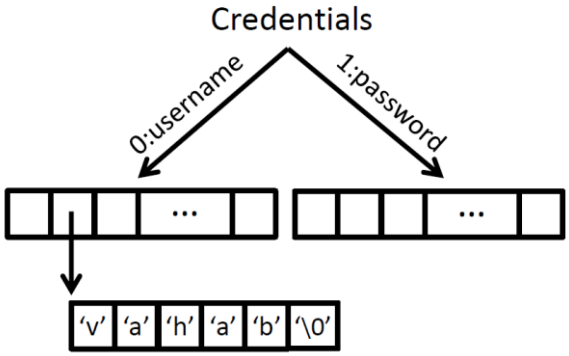

Fig. 6 Credentials 2D array structure.

```
char username[MAX_NUM_USERS][SIZE];
char password[MAX_NUM_USERS][SIZE];
char first_name[MAX_NUM_USERS][SIZE];
char last_name[MAX_NUM_USERS][SIZE];
int age[MAX_NUM_USERS];
bool admin[MAX_NUM_USERS];
long dob[MAX_NUM_USERS];
char security_question[MAX_NUM_USERS][SIZE];
char answer_security_question[MAX_NUM_USERS][SIZE];
```

Fig. 7 List of parallel arrays used in the login program.

K. File I/O

Hardcoding passwords into an executable file is considered a bad security practice for numerous reasons. Reverse-code engineering the executable file may leak the passwords. Also, if the passwords are exposed by other means, changing them would require changing the source code—making it a virtually impractical task.

Instead, the credentials should be stored in a separate file. At this point, students learn how to read the already stored usernames and passwords from a file designated for this purpose—the *credentials file*. Furthermore, they learn how to modify this file to add new users, remove users, and update users' passwords. We also use this opportunity to touch upon using cryptographically secure one-way hash functions (e.g., `crypt`) for storing credentials in files, and to point out the disadvantages of storing passwords in plaintext.

L. Pointers and Dynamic Memory Allocation

Previously, we have used the constant `MAX_NUM_USERS` as the size of the parallel arrays. By taking this approach, we are bound to either wasting memory resources by allocating more than necessary, or that our array runs out of space if there are more users than the predefined size of the arrays. To alleviate this problem we need to allocate arrays dynamically.

We first need to know how many users are there before being able to allocate enough memory to hold that

information. Hence, we first call a function that is assumed to read a file and returns the number of rows in it, representing the total number of users.

```
int n = getNumberOfRows("credentials_file");
```

We then dynamically allocate the parallel arrays. Here we create a two dimensional array to hold the usernames:

```
char** username = new char*[n];

for(int i=0; i<n; i++)
  username[i] = new char[SIZE];
```

Finally, we call a function that would iterate through the file and copy the information for each user in the corresponding arrays including `username`, one user at a time.

The students are then asked to examine the code for problems. Basically, the problem with this code is that if the number of lines are too large (beyond the range of `int`), it would result in integer overflow. This means that `username` is not large enough to hold the information about all users stored in the file. Technically, this would lead to accessing and writing into elements out of range, and hence, leading this cascading effects to heap overflow. A subtle attacker could corrupt the credentials file maliciously to cause integer overflows.

Note that in teaching the basics of how to dynamically allocate memory, we ensure that the students understand both the security and performance implications of not checking for `NULL` immediately after the `new` statement as well as deallocating that chunk of memory improperly. We further mention the importance of clearing (by setting the value to zero) the allocated memory right before deallocating, for security purposes. We explain that if the allocated memory holds sensitive information, as it does in our case, deallocation without clearing could potentially expose the sensitive information.

M. Classes and Structs

By now we are ready to wrap our implementation of the login program into an API, for other programs to use.

We have defined two classes, as illustrated in Fig. 8: `Credentials` and `UserInfo`. `Credentials` is designed to perform user management tasks such as adding and removing users, and `UserInfo` is designed to hold and update the information about a particular user.

Here, we omit the details of implementation of the functions and limit ourselves to the discussion of the constructor for `Credentials` that takes the credentials filename as an argument. The constructor parses through the file, sets the total number of users, dynamically allocates memory for `users_list`, and at last, copies all the information from the file into the array.

While teaching the basics of C++ classes, it is important to address clearly that the access specifiers (private and public keywords) are not designed to be used for security, and rather they are designed for abstraction in object oriented programming. In fact, declaring methods or data as private, enables the compiler to find programming mistakes before they become bugs. In other words, this C++ access control mechanism provides protection against accidents and not against fraud. Therefore, an effective design would protect careless programmers, who will be using our API, from unwillingly committing to such bugs.

For instance, in `UserInfo` class, by design, you can never change the `username`, after it is set by the constructor. This design would protect against accidental attempts to change the username after its creation which should never happen. Another design choice we are making here is forcing indirect access to `password`. One advantage of this approach is to ensure calling `isStrongPassword()` to check for its validity before changing it (as it is called inside `resetPassword()`). Finally, by not allowing direct access to `users_list`, we leave all memory management tasks associated to it to the `Credential` class and out of the burden of the API user. This is desirable to avoid the possibility of potential programming mistakes in doing so that could lead to memory leaks by the API users.

```
class Credentials {
public:
  Credentials(char* filename);
  bool addNewUser(char*,char*,char*,int,
                  long,char*,char*,char);
  int deleteUser(UserInfo*);
  UserInfo* getUserInfo(char* username);
private:
  int num_of_users;
  UserInfo* users_list;
};

class UserInfo {
public:
  UserInfo(char*,char*,char*,char*,int,
           long,char*,char*,char);
  char* getUsername();
  char* getPassword();
  bool resetPassword(char* password);
  bool isStrongPassword(char* password);
  char first_name[SIZE];
  char last_name[SIZE];
  int age;
  long dob;
  char security_question[SIZE];
  char answer_security_question[SIZE];
  char privileges;
private:
  char username[SIZE];
  char password[SIZE];
};
```

Fig. 8 The login program API.

5. An Informal Evaluation

While we have not conducted a formal evaluation of the course materials, we do have several indicators that suggest that our approach was successful. Remember that we used two sets of examples to teach each concept: a set of traditional examples and the examples related to our login program case study.

At the end of the Spring 2010 term, we asked the CS 1 students to rank these two sets of examples that we used to

cover the materials in various categories. These categories were (1) helpfulness in learning the programming concepts, (2) level of difficulty understanding the materials, and (3) was it or was it not a "fun" experience? We used a scale of 1 (low) to 5 (high) in this survey, and provided 32 students with questionnaires to be completed. The results from this survey are shown in Table 1.

Interestingly, the ratings for "fun" averaged around 4 for the login program example through all the topics, which was higher than ratings for traditional examples. Average ratings on whether the examples were helpful for learning programming concepts were almost the same for both sets of examples. We also observed that as the concepts got harder, the average rating for difficulty rose almost equally.

In the questionnaire that we gave to students, we also asked if they thought that the login program example we used in the course helped them to be more aware of security bugs, and in general, if it had any educational value. 93% of respondents answered "Yes" to this question.

TABLE 1
STUDENT EVALUATION OF MATERIALS WE USED IN CS 1 (H: HELPFULLNESS, D: DIFFICULTY, F: FUN)

	Login Program			Other Examples		
Topics	H	D	F	H	D	F
Intro/Variables	3.2	1.4	3.1	3.0	1.4	2.1
if-else Statements	3.8	2.0	3.9	4.0	2.1	3.1
Nested if-else	4.3	2.4	4.6	3.3	2.6	3.0
String Class	3.3	2.1	3.8	3.5	1.7	3.3
Loops	4.1	2.7	4.2	3.8	2.7	3.4
Nested Loops	3.4	3.5	3.7	3.9	3.2	3.8
Functions	3.2	3.1	3.9	3.9	3.4	3.7
Arrays/C strings	3.7	3.2	4.0	3.7	2.9	3.8
Multi-dim Arrays	3.2	3.6	4.1	3.5	3.4	2.9
Parallel Arrays	3.8	3.0	4.4	3.7	3.1	3.2
File I/O	3.6	4.0	4.5	3.4	3.4	3.1
Pointers/DMA	3.6	4.0	4.5	3.5	3.9	2.8
Structs/Classes	4.2	4.3	4.2	3.7	4.2	3.5

6. Future Work

A. Methodology Effectiveness Assessment

In our informal evaluation presented in Section 5, we did not test the students' security mindset and awareness. Hence, in our future work we plan to develop an effectiveness assessment approach to quantitatively evaluate the success of our pedagogy.

While measuring student learning is useful for evaluating the success of a particular educational technique, developing such measurements, in practice, can be challenging and can raise even more questions than answers, such as: What characteristics demonstrate student learning? Do the assessment questions really target the concept intended? Would these assessment techniques yield similar results in a different set of learner or educational contexts?

In addition, we also must clearly define what a measurement means as well as the limitation of each interpretation.

We plan to select two introductory programming classes (CS 1) within the same school term, and run them using two different teaching styles. One class (the comparison group) will be taught in a traditional style, and the other (the control group) taught in the same teaching style presented in this paper. We will then perform the following experiments and compare the data collected from the two groups for further analysis.

1) Measuring Attitude Shifts:

We will measure students' attitudes toward computer security by developing a set of questionnaires that deals with identifying perceptions and attitudes toward computer security and examine how those might change over the course of a student's progression through the introductory programming course.

We will make our collected data from the questionnaires publicly available to other researchers and educators. As with other conceptual instruments in other disciplines, maintaining the privacy and integrity of the students' responses and identities are important and will be achieved through existing anonymization techniques. We are particularly interested in making our data available for secondary analysis.

2) Code Review Exam Questions:

We will carefully design exam questions for both groups with one or more inherent security bugs, while keeping in mind that the nature of the exam questions should not be answerable by context clues or random guessing. We will then compare the percentage of students who caught the security bugs in the control group to the comparison group. Identifying or failing to identify the security bugs, however, will have no effect on their final grades in the course.

3) External Measures:

I) Final Exam Score: We will also compare students' final exam scores (or their average final grades) in the two classes to see if our proposed method had any effect (positive or negative) on learning the basic concepts of programming. These scores, while not a perfect measure for student learning, should indicate whether students were able to learn the basic concepts, along with the security concepts, without experiencing problems.

II) Student Retention: Furthermore, we plan to determine whether there is a significant ratio gap in student retention between the two groups. We achieve this by asking the students, at both the first and last lectures, if they are planning to take CS 2.

B. Teaching the Security Mindset in Other CS Courses

As part of our future work, we will explore the possibility of extending the idea of teaching the security mindset, using inherently different approaches, in other non-security lower- and upper-division computer science courses. We also aim to focus on institutions that lack a

security faculty to teach the security mindset/concepts or the instructor of the course is not a security faculty. As part of moving this to a broader scope, we will apply the multi-national multi-institutional approach to our evaluation methodologies to observe the effectiveness of our approach in different universities within different countries.

C. More Psychometric Assessments

The education and psychology fields have a rich history of developing and validating a variety of measurement instruments. Some tests have psychometric goals and do not specifically focus on learning assessment. We are particularly interested in measuring students' self-efficacy (confidence in developing secure codes) and anxiety to enable a comparison between the control and comparison groups. Such measurements should not be misinterpreted as measures of content knowledge.

7. Conclusions

In this paper we described the process of teaching the security mindset for beginning programming students in the CS 1 course that this author taught for four terms. Students, by and large, reacted very positively to this course. They enjoyed the material and found it educationally valuable and helpful for understanding the basic programming concepts. What we have described in this paper is only the first step toward achieving our goals, but we believe it to be an important step that can be built on later in more advanced undergraduate courses.

Acknowledgment

The author is very much obliged to Peter Reiher for his valuable input on technical aspects of the paper.

References

[1] S. Azadegan, M. Lavine, M. O'Leary, A. Wijesinha, and M. Zimand. "An undergraduate track in computer security." In Proc of the 8th ITiCSE, 2003.

[2] H. Cavusoglu, H. Cavusoglu, and J. Zhang. "Security patch management: Share the burden or share the damage?" Management Science, 54(4):657–670, 2008.

[3] CNNMoney. "Citi: millions stolen in May hack attack." http://money.cnn.com/2011/06/27/technology/citi_credit_card/index.htm, 2011.

[4] T. Dimkov, W. Pieters, and P. Hartel. "Training students to steal: a practical assignment in computer security education." In Proc. of the 42nd SIGCSE, '11.

[5] R. Fanelli and T. O'connor. "Experiences with practice-focused undergraduate security education." In Proc. of the 3rd USENEX CSET, 2010.

[6] Forbes.com. "Java flaw puts millions of Windows and Mac users at risk", 2011. http://www.forbes.com/sites /adriankingsleyhughes/2012/08/29/java-flaw-puts-millions-of-windows-and-mac-users-at-risk/.

[7] J. Heffley and P. Meunier. "Can source code auditing software identify common vulnerabilities and be used to evaluate software security?" In Proc. of the 37th HICSS, 2004.

[8] N. Kazemi and S. Azadegan. "IPsecLite: a tool for teaching security concepts." In Proc. of the 41st ACM SIGCSE, pages 138–142. ACM, 2010.

[9] T. Kohno and B. D. Johnson. "Science fiction prototyping and security education: cultivating contextual and societal thinking in computer security education and beyond." In Proc. of SIGCSE, 2011.

[10] K. Nance. "Teach them when they aren't looking: Introducing security in CS1." IEEE Security and Privacy, 7(5):53–55, Sept. 2009.

[11] NYtimes.com. Cyberattacks on Iran - Stuxnet and Flame, 2012. http://topics.nytimes.com/top/reference/timestopics/subjects/c/computer_malware/stuxnet/index.html.

[12] L. Perrone, M. Aburdene, and X. Meng. Approaches to undergraduate instruction in computer security. In Proc. of ASEE, 2005.

[13] P. Peterson and P. Reiher. "Security exercises for the online classroom with DETER." In Proc. of the 3rd USENIX CSET, 2010.

[14] V. Pournaghshband, M. Sarrafzadeh, and P. Reiher. "Securing legacy mobile medical devices." In Proc. of the 3rd International Conference on Wireless Mobile Communication and Healthcare (MobiHealth), 2012.

[15] B. Schneier. "Schneier on Security: The Security Mindset," 2008. http://www.schneier.com/blog/archives/2008/03/the_security_mi_1.html.

[16] Pournaghshband, V., "Teaching the Security Mindset to CS 1 Students," In Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE), March 2013

Vahab Pournaghshband, is a PhD candidate in the computer science department at UCLA. He received his masters in computer science from UC Berkeley in 2008. Also from UC Berkeley, he received his bachelors in Electrical Engineering and Computer science. His research interests are computer networks, computer security, and computing education.