Security Programming Problem-Solving: Putting More Security into the Curriculum

Westchester CS Faculty
Putting More Security into the Curriculum

The curricula of Seidenberg programs seem to be in a period of rapid evolution, upgrading in response to the current and anticipated professional activities of information technologists and broadening to include interdisciplinary domains emerging as digital technology transforms traditional institutions. Computer security, which has been a Seidenberg strength for many years, is now moving into center stage. The reasons for its assent are that it is of concern to industrial practitioners managing networks and databases; its reach into privacy, law, and law enforcement makes it interdisciplinary; and its rooting in mathematics, algorithms, code, operating systems, and digital networks rests it on a firm foundation of computing.

The way in which security has been nestled into traditional programming and algorithms courses has been through exercises. In the first programming class, passwords composed of randomly selected letters are created. The alphabet is put into a string, and random values in the range of 0..25 are used as arguments in `charAt()`. The Vigenère cipher has made a good project in courses at different levels depending on whether it is implemented procedurally, with an object-oriented design, and/or with a graphical user interface. Mary Courtney and Allen Stix, in Technical Report #155, show how they used the Vigenère cipher to illustrate object-oriented design. Advanced courses have introduced steganography by replacing the low-order bits of RGB subpixels with the bits of the `chars` in a hidden message (Courtney and Stix, Technical Report #227). One of the authors once used steganography for the project in a software engineering class.

Exercises like these are particularly useful when supplemented with discussion. With password generation, students are interested to hear about brute force password cracking and how vulnerability is reduced, due to combinatorics, with increasing password length and the expansion of the set of characters from which passwords are formed. They are also fascinated by the realization that password files, such as for email accounts, do not actually store user passwords (even as encrypted with a secret key) but, instead, store one-way hashes. (None of us have ever had the time in a programming class to explain salt.) With the Vigenère cipher, students hear the terminology associated with crypto, including "ciphertext," "Encrypt," "plaintext," "key," and "Decrypt." They also see the associated symbology: \( c = E(p, k) \). The formally obvious truism that \( k = D(p, c) \), hence the power of a known plaintext attack, comes as startling surprise. All of this is far from state of the art crypto, but it makes a good beginning.

Employing "dual purpose" exercises that couple concepts in security to programming, moves students along farther, faster. If the faculty, collectively, could agree on a methodical way of doing this, it could bring a distinctive enrichment to the curriculum. For instance, in teaching about data types (e.g. `ints`, `chars`, and `ASCII`) and argument passing to methods, we show that `Integer.toBinaryString(ch)` returns a `String` of `0`s and `1`s representing the integer value of the argument in binary. Because leading `0`s are truncated, an easy exercise is getting the `length()` of the returned `String`, representing a character's value in the collating sequence, and prepending the number of `0`s needed to bring the `String`'s length to eight, so that it always looks like a full byte.
static void display(int ch) //ch is a character such as 'A', 'z', '3', or '}'
{
    String binaryRepresentation = Integer.toBinaryString(ch);
    int padding = 8 - binaryRepresentation.length();
    for (int i = 0; i < padding; i++)
    {
        binaryRepresentation  =  "0" + binaryRepresentation;
    }
    System.out.print(str);
}

Later, this could be used to inspect encryption by EXCULSIVE-ORing the bits in the plaintext character stream with the bits in the key to get the bits in the ciphertext. The XOR operation is important in modern, symmetric cryptography. Here, the students are surprised by the operation's self-invertibility: the plaintext is restored by XORing the ciphertext with the key in exactly the same way that the ciphertext was created. This means that exactly the same procedure that performs encryption performs decryption. In Java, the bitwise exclusive-or operator is ^ (an "upper case" 6). The left and right operands are integers. The centerpieces for encryption and for decryption are below.

\[
ciphertextChar  =  (\text{char}) (plaintextChar) \ ^ \text{keyChar});    \ //encrypt
\]
and
\[
plaintextChar   =  (\text{char}) (ciphertextChar \ ^ \text{keyChar});  \ //decrypt
\]

And, of course, for a known plaintext attack it is:

\[
\text{keyChar}         =  (\text{char}) (plaintextChar \ ^ \text{ciphertextChar});  \ //attack
\]

Getting the plaintext string and the key string, culling their corresponding characters, and building the rest of the superstructure is what makes the implementation laborious and a meaty programming exercise.

A vitally important algorithm is the Diffie-Hellman key exchange. This is the algorithm that enables two individuals (e.g. Alice and Bob) to establish, between themselves (i.e. without a key distribution center or certified public keys), a secret key for symmetric encryption. The key remains concealed from an eavesdropper who captures the entire exchange. Even if the mathematical underpinnings are left unexplained, learning about the algorithm's inputs (e.g. known to both Alice and Bob, a prime number, \( p \), \ ~and~ \ an integer \( q \), less than \( p \), which may or may not be prime), its output (an integer known to Alice and Bob but not to anyone else), and its use is a worthy educational step.
The computational constructs for a prototype implementation include only the modulus operator, \texttt{Math.pow()}, and casting (in order to change the return from \texttt{Math.pow()} from a \texttt{double} to an \texttt{int}):

\begin{verbatim}
//get prime p and q < p -- Alice and Bob must be informed on these
//because they must use these same values
//for p and q in their respective computations
int p = 11;  //a prime
int q = 3;   //may be a prime or a composite

//Alice chooses a secret exponent, an integer less than p (prime or composite)
//She does not share this with Bob or anyone else. Some authors term
//Alice's secret exponent her secret key.
int aliceSecretExponent = 4;

//Bob chooses a secret exponent, an integer less than p (prime or composite)
//He does not share this with Alice or anyone else. Some authors term
//Bob's secret exponent his secret key.
int bobSecretExponent = 7;

//Alice uses her secret exponent to compute a "public key"
int alicePublicKey = ((int)Math.pow(q, aliceSecretExponent)) % p;

//Bob uses his secret exponent to compute a "public key"
int bobPublicKey = ((int)Math.pow(q, bobSecretExponent)) % p;

//Alice sends her "public key" to Bob ~and~ Bob sends his "public key" to Alice

//Alice computes the shared secret key
int shared_key_via_Alice = ((int)Math.pow(bobPublicKey, aliceSecretExponent)) % p;

//Bob computes the shared secret key
int shared_key_via_Bob = ((int)Math.pow(alicePublicKey, bobSecretExponent)) % p;

// ***the shared_key_via_Alice and the shared_key_via_Bob are the same
System.out.println( shared_key_via_Alice ); //the same as below
System.out.println( shared_key_via_Bob );    //the same as above
\end{verbatim}
The program above is prototype chiefly because the shared keys it produces are `ints`, meaning at best 32 bits long. `longs` would be better, but real cryptographic quality would required a `BigInteger` object. The other catch is getting the value for `p`, the shared prime. Java, starting in version 5.0, made this easy with the method `nextProbablePrime()` in the `BigInteger` class (in the `java.Math` package). `nextProbablePrime()` is called on a `BigInteger` object and returns a `BigInteger` whose value is the smallest prime that is greater than this. (The probability that the returned value is not a prime never exceeds $2^{-100}$, which is 7.889 x 10^{-31}.). If the implementation uses `BigIntegers` for its values, then Bob and Alice can compute their "public keys" using the `BigInteger.modPow()` method:

```java
//p, q, and aliceSecretExponent are BigInteger objects
BigInteger alicePublicKey = q.modPow(aliceSecretExponent, p);
```

Likewise for the computation of `bobPublicKey` and the shared secret key.

The problem-solving part of a Diffie-Hellman exercise is how to convert the integer computed as the shared secret key into characters, such as for use in a Vigenère system. [Robert Edward Lewand introduces this on pages 173-174 in Cryptological Mathematics, Mathematical Association of America Textbooks (2000).]

Lewand's solution is to separate the computed integer into pairs of digits, mod each pair by 26, and interpret that value as a letter of the alphabet (a = 0, z = 25):

```
+--------------------------------+--------------------------------+
<table>
<thead>
<tr>
<th>computed shared secret key</th>
<th>0 added on to complete the last pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>18458675002</td>
<td>18 45 86 75 00 20</td>
</tr>
<tr>
<td>18 19 8 23 0 20</td>
<td>s t i x a u</td>
</tr>
<tr>
<td>mod 26</td>
<td></td>
</tr>
</tbody>
</table>
```

In the course where stacks are introduced and program designs are involved enough to benefit from the UML, a wonderful exercise is the implementation of Leslie Lamport's one-time use password system. In an ordinary password system (e.g. Banner/email/Blackboard at Pace) an individual uses the same password again and again, and, as a result, the password file is pretty unchanging. This introduces the vulnerability to its theft and attack. Lamport's system makes a stolen password file all but useless because by the time a password is cracked (which would be time-intensive computational task in itself), it is likely to have been replaced. This possibility, itself, may be enough to dissuade would-be attackers. [Communications of the ACM, November 1981, Volume 24, Number 11, pages 770-772 (online at http://research.microsoft.com/en-us/um/people/lamport/pubs/password.pdf) or in Cryptography and Network Security by Behrouz A. Forouzan (McGraw-Hill, 2008) on pages 419-421].
Lamport's system is built on a one-way hash function. In the parlance of cryptographic hashing, the message to be hashed is called the "preimage" and the result of applying the hash function to the preimage, $h(\text{preimage})$, is called the message digest (or digest, for short):

$$h(\text{preimage}) \xrightarrow{} \text{digest} \quad \text{//hashing}$$

Hashing is one-way because it is computationally infeasible to compute the preimage from the digest. The MD5 (Message Digest 5) is a hash function that produces 128 bit hashes -- a 128 bit digest (32 digits in hex) is produced for all preimages, regardless of their length. To see what MD5 digests look like, see the working hash generators at [http://www.adamek.biz/md5-generator.php](http://www.adamek.biz/md5-generator.php) or at [http://www.miraclesalad.com/webtools/md5.php](http://www.miraclesalad.com/webtools/md5.php).

The claimant (i.e. the user whose identity is to be authenticated with a password) sets up by computing a series of passwords, usually a thousand of them. This is done with a recursive succession of hashes, kicked-off with a random seed:

$$\text{password}_1 = h(\text{seed})$$

$$\text{password}_2 = h(\text{password}_1)$$

$$\text{password}_3 = h(\text{password}_2)$$

$$\text{password}_{i+1} = h(\text{password}_i)$$

$$\text{password}_{1000} = h(\text{password}_{999})$$

Then, the claimant sends the verifier (e.g. the email system) $\text{password}_{1000}$ (i.e. the last one computed), keeping the others on a stack:

<table>
<thead>
<tr>
<th>Verifier's Password File</th>
<th>Claimant's Stack of One-Time Passwords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claimant's password: $\text{password}_{1000}$</td>
<td>$\text{password}_{999}$</td>
</tr>
<tr>
<td></td>
<td>$\text{password}_{998}$</td>
</tr>
<tr>
<td></td>
<td>$\text{password}_{997}$</td>
</tr>
<tr>
<td></td>
<td>$\cdot$</td>
</tr>
<tr>
<td></td>
<td>$\cdot$</td>
</tr>
<tr>
<td></td>
<td>$\text{password}_3$</td>
</tr>
<tr>
<td></td>
<td>$\text{password}_2$</td>
</tr>
<tr>
<td></td>
<td>$\text{password}_1$</td>
</tr>
</tbody>
</table>
Now, when it's time to log-on:

The claimant pops the stack and sends the retrieved password to the verifier, which (as per the illustration above) is password_999.

The verifier hashes the sent password, and if the digest is the same as the claimant's password on file, the claimant is authenticated.
In that case, the verifier replaces the password on file with the one just received and allows the claimant to proceed.

And so it goes. The verifier holds password_{i+1}. At log-on the claimant transmits password_i. The verifier computes h(password_i), getting password_{i+1}, which has to agree with the password on file. The verifier and the claimant remain synchronized because at the same log-in where the claimant pops the stack, the verifier updates the held password on file.

This exercise is a good candidate for replacing the time-honored use of a stack to evaluate a postfix expression. However, it involves more scaffolding, the extensiveness depending upon the features that the program will include. All implementations will need something that stands-in for the cryptographic hash. A very simple substitute could be something like the following, based on the power-residue method for generating pseudo-random values in the range of 0..1 non-inclusive (e.g. 0.25984678211). The crux of the power-residue computation is:

\[
\text{nextRandom} = \text{fractionalPart}(\text{previousRandom} \times \text{CONSTANT MULTIPLIER})
\]
\[
\text{previousRandom} = \text{nextRandom}
\]

where CONSTANT MULTIPLIER is a positive integer such as 789456123.

class Hash
{
    static int hash(int previous)  
        \text{not a cryptographic-quality hash function}
    {
        if (previous == 0)  //get a seed
            {
                double seed = Math.random();
                previous = (int) (seed * 1000000);  //0..999,999
            }
        double previousAsFraction = (double) previous / 1000000;
        double product = previousAsFraction * 789456123;
        double fractionalPart = product - (int) product;
        int next = (int) (fractionalPart * 1000000);

        previous = next;
        return next;
    }
}

When used, Hash.hash() gives a succession of values such as: 254428 462643 113089 493946 131357 948910 675930 219390 824970 791309 235007 97860 196779 ….
One discretionary feature of a Lamport simulation concerns the program's performance when the claimant's supply of passwords has run out. The possibilities range from refreshing the supply transparently to branding the program a pre-release version and letting this go. An intermediate possibility is outputting a message or throwing an exception. Another feature relates to versatility. Should the program merely go through the motions with one user logging-on to one system ~or~ should it support multiple users on one system ~or~ how about multiple users on multiple systems?

In its most extensive form, the program could have six or so different classes. Visualizing their relationships and services is a problem that begs for the Unified Modeling Language as a design and development tool. Every system (e.g. Pace's email, Google's email, etc.) needs a verifier. A **Verifier** object has a **LoginTable** that maps claimants' IDs to passwords. A **User** could have accounts on any number of systems, and each **Account** needs its own **PasswordManager**. Each **Account** is associated with a different **Verifier**.

Relative to the implementation, the students could put-up the stack for themselves from an array, or they could use a structure from the Java Collections Framework. Likewise for the verifier's table from which passwords are looked-up by userID, and the user's table from which account objects are looked-up by their names.

Computer security is rich with topics that yield exercises and projects suitable for classes in programming and software design at all levels. The purpose of this written matter is to proffer the suggestion that the Seidenberg School build upon its security strengths by incorporating security into its programming, algorithms, and software classes. If this idea were agreeable to faculty; if the security concepts and exercises in each course could be installed as "official" course objectives; and if the program's curriculum was planned to be cohesive; we'd have something of distinctive value to offer and advertise. The trick to effective education in these times of curriculum bloat is to be creative in killing two birds with one stone -- that's the overarching principle.
While the curriculum is the focus of this document, it originated from thinking about the problems for use in the programming contest to be hosted by the Westchester Department of Computer Science this spring. We are seeking problems that relate to security. Much discussion lies ahead, but here are some starter ideas.

1. Password Generator (Discussed back on page 1)

A computer center needs a program that generates passwords. Each password is a random sequence of lower-case letters, eight letters long. Here are some samples:

- rtqznpons
- zvminfmv
- qfkomlwv
- ivomjitr
- sxjhfqg
- arriemve

Create the program that will do this. It must have the following features:

i) The set of characters from which the passwords are formed must be extensible.

ii) The software must allow for an upgrade enabling the operator to enter password length

---

Extensibility: It should be possible to modify a system so that it can meet new requirements including changes on existing functionality, additional features, or the ability to perform in a new environment.

```java
class Passwords {
    public static void main(String[] args) {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";

        for (int numOfPasswords = 0; numOfPasswords < 6; numOfPasswords++) {
            String password = "";
            for (int numOfLetters = 0; numOfLetters < 8; numOfLetters++) {
                int randomChar = (int)(Math.random() * 26);
                password = password + alphabet.charAt(randomChar);
            }
            System.out.println(password);
        }
    }
}
```
2. Building a P-box (an ideal application of indirect reference)

A transposition cipher changes the order of the plaintext characters, but not the identity of the characters themselves.

For instance if this were the plaintext:  

```
  a b c d e f g h i j
```

This could be the ciphertext:  

```
  d f i c g e b j a h
```

And this particular encryption may be represented as:

```
  | encrypt |
  | 0 1 2 3 4 5 6 7 8 9 |
  | 8 6 3 0 5 1 4 9 2 7 |
```

Decrypting the ciphertext is a matter of applying the inverse re-ordering.

```
  | decrypt |
  | 0 1 2 3 4 5 6 7 8 9 |
  | 3 5 8 2 6 4 1 9 0 7 |
```

If encryption is shown as a down arrow on the first array, decryption is shown by an up arrow. Similarly, when the operations are depicted by a P-box (P for permutation, inasmuch as encryption and decryption with a transposition cipher consist of re-permuting the characters). The lines inside the P-box represent wires connecting an input pin to an output pin.

Part a) Write a module that implements a 10 character P-box in software. It should:

** accept an encryption permutation action via assignment such as

```
String permutationAction = "8630514927";
```

** offer a method that accepts a ten-character `String` as an argument and returns its encryption re-permutation

** offer a method that accepts an encrypted permutation created by the other method and returns its decryption
Part b) When block ciphers are pressed into service, plaintexts of arbitrary length are processed in chunks. Here, the block-length is ten characters. When the final block of plaintext is less than ten characters long, it is padded with additional characters to bring its length up to ten. We'll use trailing asterisks.

Write a program that uses the module from Part a to encrypt and decrypt a string of plaintext entered as a String literal.

class PBox  //the logic for Part a
{
    public static void main(String[] args)
    {
        String permutationAction = "8630514927";
        //set-up the array of indirect reference
        int[] at = new int[permutationAction.length()];
        for (int i = 0; i< permutationAction.length(); i++)
            {at[i] = Character.getNumericValue(permutationAction.charAt(i));}

        String plaintext = "abcdefghij";
        char[] ciphertextArray = new char[permutationAction.length()];
        for (int i = 0; i< permutationAction.length(); i++)
            {ciphertextArray[ at[i] ] = plaintext.charAt(i);  //indirect reference}

        String ciphertext = "";
        for (int i = 0; i< permutationAction.length(); i++)
            {ciphertext = ciphertext + ciphertextArray[i];}

        System.out.println( plaintext );
        System.out.println( ciphertext );
    }
}

************ Output ************
C:\ProgrammingContest>java PBox
abcdefgij
dficgebjiah
C:\ProgrammingContest>
******************************************************************************/
Ideas for additional programming contest problems can be culled from the text. The difficult part is supplying the needed background while, at the same time, keeping instructions concise. The P-box problem is already bordering upon being too verbose. The following, seems concise but is unsatisfactory because not everyone would know what is meant by a Vigenère cipher.

3. Write a procedural program that implements a Vigenère cipher. Input is read from the command line. The first command line argument to java is the plaintext. The second is the key. Arguments that contain a space, such as a multiple-word plaintext, should be delimited by quotation marks.

The program should open as shown below and be able to work with any alphabet supplied. A plaintext character that is not in the alphabet is placed into the ciphertext without a "roll ahead." A key character that is not in the alphabet gives no "roll ahead."

```java
class VigenereCipher
{
    public static void main(String[] args)
    {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        String plaintext = args[0];
        String key = args[1];
    }
```

The output should consist of an echo of the entered plaintext, the ciphertext, and the decrypted ciphertext (which should look the same as the plaintext).

C:\ProgrammingContest>java VigenereCipher "Encrypt this sentence." secret
Encrypt this sentence.
Ereicil vyml wgexxffg.
Encrypt this sentence.
class VigenereCipher
{
    public static void main(String[] args)
    {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        String plaintext = args[0];
        String key = args[1];

        String ciphertext = "";  //to be computed

        //encrypt
        for (int i = 0; i < plaintext.length(); i++)
        {
            char plaintextChar = plaintext.charAt(i);
            char keyChar = key.charAt(i % key.length());

            if (alphabet.indexOf(plaintextChar) != -1 &&
                alphabet.indexOf(keyChar) != -1)
            {
                int encryptionInt = alphabet.indexOf(plaintextChar) +
                                    alphabet.indexOf(keyChar);

                encryptionInt = encryptionInt % alphabet.length();
                ciphertext = ciphertext + alphabet.charAt(encryptionInt);
            }
            else
                ciphertext = ciphertext + plaintextChar;
        }

        //decrypt
        String decipheredtext = "";  //to be computed

        for (int i = 0; i < ciphertext.length(); i++)
        {
            char ciphertextChar = ciphertext.charAt(i);
            char keyChar = key.charAt(i % key.length());

            if (alphabet.indexOf(ciphertextChar) != -1 &&
                alphabet.indexOf(keyChar) != -1)
            {
                int decryptionInt = alphabet.indexOf(ciphertextChar) -
                                    alphabet.indexOf(keyChar);

                if (decryptionInt < 0) decryptionInt += alphabet.length();

                decipheredtext = decipheredtext + alphabet.charAt(decryptionInt);
            }
            else
                decipheredtext = decipheredtext + ciphertextChar;
        }

        System.out.println(plaintext);
        System.out.println(ciphertext);
        System.out.println(decipheredtext);
    }
}
The Seidenberg School of Computer Science and Information Systems
Pace University

Technical Report Series

EDITORIAL BOARD

Editor:
Allen Stix, Computer Science, Pace--Westchester

Associate Editors:
Constance A. Knapp, Interim Dean, The Seidenberg School of Computer Science and Information Systems -- Pace University
Susan M. Merritt, Computer Science, Pace--Westchester

Members:
Howard S. Blum, Computer Science, Pace--New York
Mary F. Courtney, Computer Science, Pace--Westchester
Nicholas J. De Lillo, Mathematics and Computer Science, Manhattan College
Daniel Farkas, Information Systems, Pace--Westchester
Fred Grossman, Information Systems; Doctor of Professional Studies, Pace--New York and White Plains
Fran Goertzel Gustavson, Information Systems, Pace--Westchester
Joseph F. Malerba, Computer Science, Pace--Westchester
John S. Mallozzi, Computer Information Sciences, Iona College
John C. Molluzzo, Information Systems, Pace--New York
Pauline Mosley, Technology Systems, Pace--New York
Narayan S. Murthy, Computer Science, Pace--New York
Catherine Ricardo, Computer Information Sciences, Iona College
Judith E. Sullivan, CSIS Alumna, MS in CS from Pace--Westchester
Sylvester Tuohy, Computer Science, Pace--Westchester

The Seidenberg School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. Technical Reports is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
The Seidenberg School of Computer Science and Information Systems
Goldstein Academic Center
Pace University
861 Bedford Road
Pleasantville, NY 10570-2799