Project 6: “Secure Cyberspace”

Area Coordinator:
Dr. John Franco
Geier Professor, Computer Science
Department of Electrical Engineering and Computing Systems (EECS)
College of Engineering and Applied Science
PO Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030
Office: 831 Rhodes Hall
E-Mail: franco@gauss.ececs.uc.edu
Phone: 513-556-1817

Sub-Area Coordinator:
Dr. Prabir Bhattacharya
Professor, Computer Science
Department of Electrical Engineering and Computing Systems (EECS)
College of Engineering and Applied Science
PO Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030
Office: 829 Rhodes Hall
E-Mail: bhattapr@ucmail.uc.edu
Phone: 513-556-1815

Graduate Research Assistant:
Mr. Minzhe Guo
Graduate Student in Computer Science and Engineering (CSE)
Office: Old Chemistry 811E
E-Mail: guome@mail.uc.edu
Phone: 678-507-9683 (C)

Project Abstract

Computer Science is the major provider of basic tools for computer and network security. Many of these tools use mathematical concepts that often are omitted in middle school and high school curricula. In this project participants will familiarize themselves with these mathematical concepts, and see how these are applied to maintaining data confidentiality and integrity, and to the authentication of remote parties that may be allowed access to sensitive data. To reinforce knowledge of these applications, participants will form teams with the aim of building software systems (in Java) to compete in a contest, controlled by a monitor provided by the instructor, with the goal of stealing sensitive data from other teams' systems while protecting their own sensitive data. The systems will be built in stages, each following some significant aspect of the course.

Early in the project participants will experiment with and discover mathematical concepts that are the foundation of many security tools. Several applets are provided for this. For example, generating prime numbers and testing whether a number is prime are critically important tasks in cyber-security. The classic Miller-Rabin algorithm for doing these involves the use of several aspects of modulo arithmetic including the fact that several numbers may have square roots, several may not have a square root, all square roots are integers, and if the modulus is prime there are only trivial square roots of 1. Figure 1 shows an applet which helps gain some understanding of these principles without mathematical proofs. The participant selects \( n \) to be 2 for square roots (3 would be for cube roots and so on) and a number that
could be prime or not prime. In Figure 1, 17 was selected. There are only 16 numbers modulo 17, not
counting 0, and they are shown in the upper row. Squares of these numbers are shown in the bottom
row. Looking upward from the bottom row, square roots modulo 17 can be seen. In this case there are
two trivial roots of 1. If an even number is chosen as the modulus, one would likely see four or more
square roots of 1. Fermat's little theorem is also used in the Miller-Rabin algorithm and there is a similar
applet for understanding this theorem without mathematical proof. These facts are crucial to the
operation of the Miller-Rabin algorithm as it provides a fast, iterative, way to determine whether a number
\( n \) is prime: choose a number \( a \) and exponentiate it (there is an applet for understanding modulo
exponentiation as well) based on \( n \). Iterations involve squaring and checking whether +1 or -1 mod \( n \) is
obtained. If the answer is no for all iterations, then \( n \) is certainly not prime, otherwise it is prime with error
probability \( \frac{1}{4} \). Repeating for a different \( a \) with the same result, changes the error probability to 1/16 and
so on. The applet for experimenting with this algorithm is shown in Figure 2. This applet can be used to
generate a prime of a specified number of bits or test whether a specified number is prime. In this case, a
32 bit prime number was sought. The yellow area indicates the current number under test and the field to
its right is the current error probability. Several numbers were tried before this and failed (with a red area
instead of a yellow one) and the lowest field shows why. Other mathematical concepts are demonstrated
via applets in similar fashion.

Group theory is important to understanding public key cryptosystems including elliptic curve
cryptography. Figure 3 shows an applet that helps a participant to understand what a generator of a
cyclic group is and what conditions under which a generator can be found. In this case, it has been found
that 8 (selected) is not a generator for the numbers mod 17.

As participants are gaining insight into the mathematical concepts necessary for cryptosystems they
will be coding algorithms that will be used as building blocks in their contest systems. Examples include
Diffie-Hellman key exchange, RSA encryption and signing, Fiat-Feige-Shamir zero-knowledge
authentication, and Karn symmetric key cipher. Additional applets will help in understanding how these
work. The participant will write some of these applets (the code will be used in their contest system). The
level of code is fairly easy. For example, Figure 4 shows a section of code representing the RSA
algorithm that participants will implement. Figure 5 shows an applet that participants will write to
experiment with RSA. They will try some parameters and notice that in some cases encryption/decryption
does not work. This will lead to several discoveries about necessary restrictions on the parameters and
how to compute an inverse. Other applets will be used to experiment with, for example, \( \mathbb{Z}^n \) to help
understand why RSA cryptography works. Some symmetric ciphers will also be considered for coding
including AES, DES, and 3DES. Depending on the ability and desire of the participants these may be
implemented in the language Cryptol which can be used to determine the correctness of their design. For
example, from an implementation of 3DES functions \( \text{encrypt} \) and \( \text{decrypt} \) one might want to prove the
following:

\[
\text{theorem thml:}
\{k x\}. \text{DES.encrypt}(k, \text{DES.decrypt}(k, x)) == x;
\]

Running this proof in cryptol looks like this:

```plaintext
3DES> :s sbv
3DES> :prove thml
Q.E.D.
3DES>
```

Key distribution is an important aspect of cyber-security and Java programs will be developed by the
participants to provide public keys, as a certification authority, or secret session keys, as a KDC. For the
KDC this entails providing a secure protocol which can be implemented by the participants. For public
keys this entails making certificates and serving them publicly.

A protocol for establishing secure communication between a remote device and a device that is
behind a firewall protected network (e.g., VPN) will be implemented. Several protocol proposals will be discussed with a critique of each provided.

Finally, with all software components completed, participants will become contestants in a cyber-war game lasting four days with the aim of protecting sensitive data while trying to steal another team's sensitive data. A group accumulates points via secure communications between two or more group system accounts. A contest server, called the monitor, records and relays all communications, completes all requested transactions and maintains the official point tallies for all groups (it is the official scorekeeper). Each group has three accounts. The group achieving the maximum number of points over all three accounts at the end of the contest is declared the winner. All competing systems are aware of each other but lack some details such as what port another system might be transacting on. If one group has complete information about another group's system, that system would be compromised and all its points would be vulnerable to theft. In order to receive points from the monitor, an account has to "expose" itself to the other groups: that is, some information must be leaked. So, securing communications and even applying some ad-hoc tricks is absolutely necessary to prevent losing everything.

The protocol supporting communication with the monitor, including commands that may be used for authentication, gaining points, and stealing follows protocols that are actually used in practice. Each system must obey the protocol and should contain features intended to defend itself from attack by another group's system. A system's components include: secret key cryptography via Karn Symmetric Key algorithm, one-time secret key generation via Diffie-Hellman key exchange, signing via public key cryptography (RSA), certificates and a certification authority (SHA and RSA), authentication, integrity protection, zero-knowledge authentication, and login protocols.

The monitor itself has some bugs. The monitor code is distributed to all contestants. Usually, the more motivated groups will exploit the bugs to steal points. It is not uncommon to watch a group accumulate many points, become the leader, then in one minute, have no points whatsoever. The expression on the faces of people in these groups is striking and shows how much impact this contest has on them. The more motivated groups will sometimes go through several boom-bust cycles and are more likely to eventually win the competition.
Figure 1. Interactive applet demonstrating square roots modulo a number, in this case 17

Project 6
Secure Cyberspace
Summer 2015

Roots in Modulo Arithmetic
Secret Key, Public Key, Hash Algorithms, Protocols, Authorization, etc.

Exponentiate all the numbers in the top row by \( n \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod: 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>res: 1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>8</td>
<td>2</td>
<td>15</td>
<td>13</td>
<td>13</td>
<td>15</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Instructions: Select a number from the drop down menu labeled \( n \) and call it \( n \). Select a number from the drop down menu labeled \( mod \) and call it \( p \). The set of all integers from 1 to \( p-1 \) is displayed in the row labeled \( n \). For each number \( m \) in that row, \( m^n \mod p \) is shown in the same column in the row below it.

Observe:
1. If \( n \) is 2 and \( p \) is a prime number, then only half the numbers from 1 to \( p-1 \) appear in the bottom row and each number that does appear, appears twice. Hence, half the numbers have square roots and half have no square roots.
2. If \( n \) is 2 and \( p \) is a prime number, the numbers in the bottom row form a palindrome (hence square roots are \(+/-\) some number).
3. If \( n \) is 2 and \( p \) is a prime number then 1 shows up only at the extreme left and extreme right of the bottom row. This means the square root of 1 is only +1 mod \( p \) and -1 mod \( p \). If \( p \) is not a prime, then 1 may show up elsewhere in the bottom row. But this is not always the case. For example, if \( p = 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21 \) then 1 doesn't, does, doesn't, does, doesn't, does, does, doesn't, does, does show up elsewhere in the bottom row, respectively. Hence, 1 mod \( p \) has only trivial roots of unity if \( p \) is a prime number but about half the time has non trivial roots if \( p \) is not prime.

Figure 2. Interactive Applet Demonstrating the Miller-Rabin Primality Test/Generate Algorithm

Project 6
Secure Cyberspace
Summer 2015

Miller-Rabin Prime Number Generator
Secret Key, Public Key, Hash Algorithms, Protocols, Authorization, etc.

# bits prime should have: 0

Gen Prime → Start → Next → Try Again

Gen Prime

n:

error prob:

n-1:

# n's tried: 0 # tries left for this n: 0

a:

b: the number of least significant 0 bits

m: 0-1 stripped of the least significant 0 bits

Instructions: Choose either Gen Prime to generate prime number or Test Prime to test one. In the case of Gen Prime enter a number of bits between 4 and 64 in the text field at the upper left If this cannot be done, press the Try Again button first. Press Start to begin. Press Next to advance to the next step of the Miller-Rabin algorithm. Numbers to be tested as prime are displayed in the text fields labeled n. When the Miller-Rabin test fails for a non-prime, that number will become red. Then, advancing to the next step results in a new number to be tried. If the number to be tried is prime, it will show yellow when the Miller-Rabin test has been applied, indicating it has passed the test. Then, advancing to the next step results in a new number and Miller-Rabin test. If the number is prime, it will never go red. The only way to generate a new number is that case is to press the Try Again button.

Examples: 1217 (a prime), 705 (a non-prime), 14

If Test Prime is selected the operation is the same except that the user must specify a number to try in the text field labeled n.
Figure 3. Interactive Applet Demonstrating Generators for a Cyclic Group

Figure 4. Sample of Code that Implements RSA Cryptography

```java
import java.math.BigInteger;
import java.io.Serializable;

public class PubRSA implements Serializable {
    /** The components of the public key */
    BigInteger e, n;

    /** Takes the exponent (e) and the modulus (nVal) as the two parameters of the RSA secret key. nVal was constructed from two primes (p,q) previously, and nVal = pq */
    public PubRSA(BigInteger eVal, BigInteger nVal) {
        e = eVal;
        n = nVal;
    }

    /** Encrypt the message m with this key, m < n, or you suffer truncation */
    public BigInteger encrypt(BigInteger m) {
        return m.modPow(e, n);
    }

    /** Verify that this public key's private key signed the message. "reverse cipher" */
    public boolean verifySig(BigInteger m, BigInteger s) {
        return m.equals(s.modPow(e, n));
    }

    /** Return the public exponent. */
    public BigInteger getExponent() { return e; }

    /** Return Modulus. */
    public BigInteger getModulus() { return n; }
}
```
Figure 5. Interactive Participant-Written Applet for Experimenting with RSA Cryptography

<table>
<thead>
<tr>
<th>Project 6</th>
<th>Secure Cyberspace</th>
<th>Summer 2015</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RSA: Public Key Encryption</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Public Key:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p: 89</td>
<td>n: 8033</td>
<td>d: 3793</td>
</tr>
<tr>
<td>q: 97</td>
<td>n: 8033</td>
<td>e: 49</td>
</tr>
</tbody>
</table>

**Message:**
Now is the time for all good men and women to come to the aid of their party

**Encrypted:**
\[ \text{message encrypted text} \]

**Decrypted:**
Now is the time for all good men and women to come to the aid of their party

**Background:** see this description of RSA.

**Instructions:** Enter a p and q in the appropriate textfields. Enter the encrypt key e. Hit return. The product appears as n and the decrypt key is shown as d. Enter a message in the textfield called "Message." If the inverse of e exists, the message will be encrypted, as shown, and then d will be applied to decrypt it.