CSE 539 Applied Cryptography Fall 2015 Course Project

Attempts on Matasano Crypto Challenges

by Girish Raman and Shankar Manickavasagam

Introduction

The following is a documentation of what we have learnt via the course project of CSE 539 Applied Cryptography in Fall 2015. We have taken up the Crypto Challenges posted by Matasano Security in 2013. It involves practical cryptographic challenges that let us learn the nuances of how various cryptographic algorithms work, about the flaws in them, and how an attacker can utilize these flaws to gain an advantage on breaking the cryptosystem. The document first talks about the various challenges we solved, in specific details. Then it covers the several secure coding practices (in Java) that we learnt through this project and the details of their implementation.

A brief Description

We have been able to implement random challenges from the entire set, as part of this project, mostly covering the Basics and Block Cryptography. Some challenges also cover Stream Cryptography, Public Key Cryptography and Hashes. We started with the basics - simple XOR operations, Single-byte keys, Vigenere ciphers, etc and breaking them. Block Crypto challenges were about implementing AES in the ECB, CBC and CTR modes and understanding the flaws in them and learning how to take advantage of them. Public key cryptography challenges were about implementing the key exchange algorithm - Diffie Hellman algorithm and one of the first used public key cryptography methods - RSA algorithm. These challenges also helped us to know the various attacks on the key exchange and the RSA algorithm.

What the implementation involves

Our implementation of these challenges have been guided by specific goals to each challenge given at the website - cryptopals.com. The challenges typically start with implementing certain schemes, simple in the beginning, that we will later attack in the future challenges. The initial challenges set up the cryptosystem-attacker environment, establishing things that need to be set up like oracle functions, black boxes, etc. Then the actual challenge comes up after these set-up steps wherein we would have implemented the actual attack, typically making use of the way that particular scheme functions, to our advantage.

What we have learnt

Broadly speaking, we could say that we have learnt to implement cryptographic schemes in practice and understand how these various schemes work. Understanding how they work, also helped us learn the kinds of attacks possible on them. We learnt to implement these attacks in practice with actual ciphers, and succeeded in at least gaining an advantage as an adversary if not decrypt and get the complete plain text, without the secret key. We have also learnt secure coding practices for the Java language and the importance of such practices in securing our applications.

Implementation Description

We have implemented the Matasano Crypto challenges in Java. Our implementation of these crypto challenges does not involve just the attacks themselves. We first start with implementing the cryptosystem itself, setting up the attack environment - including the algorithm (usually simple in the first few challenges, like XORing, Vigenere, etc. Later we move on to the more complex crypto functions (mostly AES) in various modes of operations.), establishing the resources that are made available to the attacker, setting up oracles like encryption oracles, decryption oracles, padding oracles, etc.

After all the initial setup, comes the actual attack from the attacker/adversary. We have not set up different pieces of code separating the crypto system, the oracle functions, the code for the attackers and all other components of the system. For each attack, all these components are weaved into the same project and it needs to be understood which parts of the code are controlled by which user/misuser of the system.

Since the focus of this course project is to understand and perform the various attacks on crypto systems, we have not put in a lot of effort in implementing the complex algorithms like AES for instance. We have just used the standard Java libraries and other external libraries when needed. What we have indeed coded is the attack itself, concentrating more on what the attacker would try to do to break the system.

All parts of the code have been written on our own except the parts which have been followed from SEI / CERT in order to learn and implement Secure Coding Practices in Java. While writing the code for performing all these attacks, we have made sure to follow certain Secure Coding Practices that are suggested by CMU-SEI / CERT that lead to the production of secure code - security covered in various aspects and angles.

Cryptographic Learnings

This section describes in detail about the project, the attacks we have performed through this project and our learnings from a cryptographic point of view.

First and Foremost : Always Operate On Raw Bytes

We learnt this quite the hard way. Always operate on raw bytes. *Always.* In almost all of the challenges that involved encryption and decryption, the result of such operation produces bytes. Sometimes these bytes might represent the byte values of ASCII printable characters (the result of decryption resulting in English text, for instance), but sometimes these bytes are not ASCII printable - they either range in 0x00

to 0x19 or the higher ASCII values - 0x7F to 0cFF. In such cases, it is not advisable to operate on such bytes in the form of, say, a String object.

In either case, it is always advisable to just store the raw bytes in a byte array, than to convert it to any object notation. Doing that won't be a problem when we are dealing with ASCII bytes, but in most cases while coding for crypto, the bytes are not ASCII. When we do convert non-ASCII bytes to a String object, they sometimes get lost / mixed with successive bytes, resulting in a different set of bytes than what would be expected. That is where character encodings come in. Non-ASCII bytes can always be encoded into a common representation involving just ASCII characters, from which it is possible to extract these bytes later on.

Character Encoding, the ⊕, Vigenère and the other basic concepts

The first set of challenges give a basic introduction to Cryptographic concepts, starting with Encoding styles. We learnt about the HEX and Base64 encoding styles, and how to convert a piece of message from hex to base64 and vice-versa.

XOR – the most widely used logical operation in anything Crypto. The next few challenges made us implement XOR in different ways – XORing 2 starting byte by byte, single-byte XOR, etc. Single byte XOR detection came next, wherein we learnt about making a computer program understand what plain text 'makes sense' – by using the ETAOIN SHRDLU rule of English alphabet frequencies. We can make a program *score* English plain-text strings based on such character frequencies and find out which one string has been single-byte-XOR'd from a set of random strings.

Then came the age-old Vigenère cipher – or what's called the Repeating Key XOR here. The first task was to construct a Vigenère cipher with a given key. We formed a repeating string of the key as long as the given plain text message, and then used the simple byte by byte XOR function we created in the previous tasks to form the Vigenère cipher. That was the easy part. The hard(er) part was breaking a Vigenère cipher, which was the next challenge.

Of course, using a Vigenère cipher today is not even a thought; but learning to decrypt one was fun. We don't know what the key is. Worse, we don't even know how long the key is. When brute force seems like the favorable option, turns out we can narrow down the possibilities very efficiently by guessing the key size and normalizing the hamming distance of 2 successive blocks of the cipher. Once we find the key size that has the smallest normalized hamming distance, we can divide the cipher into blocks of that size and the corresponding characters of each block forms a string that is basically a single–byte XOR cipher. Then we used the single-byte XOR function we created before to decrypt each string and rearrange the characters in the right order to get the original plaintext.

We also learnt two other methods for finding the key size. The Kasisky Test - it relies on the fact that if 2 identical portions of the plain text appear at a distance of a multiple of the key size, they will be encrypted to the same cipher text. Index of Coincidence - It represents the frequency that 2 characters chosen at random from a plaintext are the same. Using English character frequencies, we can efficiently guess the key length that makes the plaintext 'look like' English text.

java.util.Random vs java.security.SecureRandom

In Java, there are 2 classes in the JDK that let us generate randomness. One is the class java.util.Random and the other is the class java.security.SecureRandom. It turns out that the instances of the class Random are not cryptographically secure [7]. The nextInt() function in the class Random that generates the random integers, is a linear congruential pseudorandom number generator. It turns out, we can predict the output of linear congruential generators [8]. The Random class maintains a seed in an internal state and for every number generated, it will be changed using a static formula. So it has been proven that this seed can be found out by easily brute forcing it (only 16 bits, 65536 possible values) in less than a second.

On the other hand, the class SecureRandom generates cryptographically secure random numbers and hence we have used only SecureRandom instances to generate random keys and IVs wherever needed.

The Advanced Encryption Standard and its Modes of Operation

After the first few basic challenges, we landed in a set of more serious concepts, first few of which deals with the Advanced Encryption Standard (AES, the 128 bit key version). We learnt to implement the various modes of block cipher operation - the Electronic Code Book mode, Cipher Block Chaining mode and the Counter mode. We have been able to exploit the vulnerabilities of these 3 modes of operations in the following ways -

Electronic Code Book Mode Detection

In challenge #7, we implemented AES in the ECB mode of operation using Java's javax.crypto package. In challenge #8, we were given a set of cipher text strings in which one was encrypted in the ECB mode of operation and the goal was to detect which one it was. The way that ECB works – encrypting each block of the plaintext separately – it makes it very easy to detect ECB. Similar PT blocks are encrypted into similar CT blocks! So it is a matter of finding if there are any identical blocks in the ciphertext. It is fair to say that, with high probability, if one particular ciphertext string has many identical blocks than any other ciphertext string, then that particular string has been formed by an ECB mode of operation of the block cipher!

Byte by byte decryption of ECB - Simple

In challenge #12, the task was to decrypt a cipher, without knowing the key that was used, just by having a prior knowledge of what the first few bytes of the plaintext would have been. It turns out that, in many practical cases, most pieces of messages that are encrypted have something like a header as the first few bytes which are then followed by the actual message. So the attacker almost always knows what the encrypted message starts with. In this challenge, we have modelled this situation slightly different - the attacker is given the capability of modifying the first part of the to-be-encrypted string like this - `attacker-controlled-string || unknown-string`. We were able to decrypt the

'unknown-string' just by controlling the first portion of the plaintext through repeated calls to the encryption oracle. The way that ECB works, similar blocks of plaintext always encrypts to similar blocks of ciphertext. So we were able to decrypt the first byte of the unknown-string by comparing cipher texts with a compiled dictionary - ciphers obtained by repeated calls to the encryption oracle with attacker-controlled-string AAAAAAAAAAAAAAAA, AAAAAAAAAAAAAAAB, and so on. One of these would be the same as the original ciphertext block and that corresponding byte is the first byte of the unknown-string. Similarly, we can drop the first byte, then find the second one, drop it, find the third one and so on.

Byte by byte decryption of ECB - Harder

The next challenge was similar to the previous one - decrypting an unknown AES-ECB encrypted string by knowing a portion of the plaintext preceding it. But what makes this challenge harder than the one before is that the plaintext is in this format - (`random-prefix || attacker-controlled-string || unknown-string, random-key`). The attacker only knows a portion of the plaintext in the middle. We have modelled this such that a random number of random bytes are prepended to the attacker-controlled-string which is then followed by the unknown-string which is to be decrypted. The problem here is that we don't know where the attacker-controlled-string starts. Turns out we can find that easily by repeatedly changing the bytes one by one in the attacker-controlled-string. One of these changes would result in the next ciphertext block to scramble. With that, we will know the number of bytes of the random-prefix in that block. After that, we can proceed like in the previous challenge to decrypt the unknown-string.

ECB Cut and Paste Attack

This is another attack that relies on the ECB mode of operation. Again, the way that ECB operates, a block of plaintext always encrypts to the same ciphertext block. With this said, we can just copy a block of ciphertext from one oracle access, and paste it in place of another block from a different oracle access and this goes undetected. This way, in challenge #13, we have been able to forge 'admin' user accounts just by cut-pasting a cipher block into the appropriate place. Such 'active attacks' go undetected in the ECB mode of operation.

Cipher Block Chaining mode and PKCS Padding

We have implemented AES in the CBC mode of operation in one of the challenges. We learnt about how this mode of operation could be used along with a padding scheme such as the PKCS #7, to encrypt messages that are irregularly sized and not an exact multiple of the block size. We also learnt to implement a Padding Oracle - one that tells you if a given cipher decrypts into a plaintext that is correctly padded according to the PKCS#7 scheme. This padding oracle was used in some of the challenges involving the CBC mode.

ECB or CBC? with a Black Box

One of the challenges was to detect if a black box has encrypted the given PT in ECB mode or in CBC mode. The black box flips a coin to decide if it's going with ECB or CBC every time. Because of the properties of ECB and CBC, it is very easy to detect this. At first, to make things a little bit difficult, 5-10 (chosen randomly at runtime) random bytes are added as a prefix to the PT and 5-10 (chosen randomly at runtime) random bytes are added as a suffix to the PT. Then comes the padding, if needed. The key is the fact that we have control over the PT. Since we can give out any PT of our choice, we can give something like this -

abcdefghijklmnop abcdefghijklmnop abcdefghijklmnop

The block size is 16. When we give this as the PT, random bytes are added to the beginning (and also to the end which we don't care about). After this, the second block and the third block will be identical, irrespective of the number of bytes added in the beginning.

Now the CT will have the second and the third blocks to be identical as well if it was ECB, and not equal otherwise.

CBC Bit flipping attacks

The next attack we performed was on the CBC mode of operation. In the CBC mode of operation, we learnt to implement the bit flipping exercises that we solved in class and from the textbook. The way that CBC works, if an active attack is performed on a piece of cipher block, changing,say, the third bit of that block, then when this modified ciphertext is decrypted, the third bit of the next plaintext block will be modified too, apart from scrambling the current block completely. We used this property to modify a cipher block and introduce a new message in the successive block - in a communication between two people, we changed user data from '...role=manager…' to '...role=admin...' by changing the corresponding bits in the cipher block.

CBC Padding Oracle attack

Though the CBC mode allows irregularly sized messages to be encrypted, it does pad the last few bytes to a complete block size. Often, at the decryption side, there are padding oracles that give out error messages about invalid paddings. Using such an oracle constructed before, we implemented the padding oracle attack on the CBC mode of operation that lets us decrypt the entire ciphertext. We learnt two ways to do that - the first[11], to decrypt the last block, and the second[12], to decrypt the rest of the blocks.

CBC Key recovery when used as IV

This was another particular case in the CBC mode of operation wherein the sender and the receiver decide to use the key itself as the IV / Nonce. We learnt that this is one of the worst things to agree to,

in the CBC mode of operation. We were able to very easily break the cryptosystem and find the Key itself (not just *something* about the plaintext) from a scheme that uses its secret key to be its IV. Through an active attack on the ciphertext, changing a block to all zeros, we were able to find the key via the decrypted plaintext that looked corrupted to the receiver.

CTR mode fixed nonce attack #1

Moving on to the Counter (CTR) mode of operation, we learnt about making the AES (or any other block cipher) act like a Stream Cipher. Apart from the key, that is to be kept a secret, there is also a random *nonce* or the *Initialization Vector* that is involved in the CTR mode. The rule is that a random choice of nonce should be made for every encryption. We learnt how to attack a system that doesn't follow this rule and uses a fixed nonce every time. We were able to find patterns in the ciphertext, make meaningful guesses about the plaintext and learnt many things about the plaintext just from the ciphertext.

CTR mode fixed nonce attack #2 - the repeated key method

We also learnt to approach the same system (one that uses a fixed nonce in CTR mode) in a slightly different way. We truncated the ciphertexts to a common length - this gave us a set of bytes that were the result of encryption with the same 'key stream' (since the same nonce was used). We were able to implement the 'Repeating-key XOR attack' (the attack on Vigenere cipher) on this system now, and we were able to decrypt the ciphertext completely.

CTR bit flipping attack

Another attack we did on the CTR mode of operation was the bit flipping attack that we previously performed on the CBC mode. Doing the same with the CTR mode, we were able to create 'user=admin' profiles by modifying ciphertexts of 'user=manager' profiles. The difference between the 2 modes of operations wrt this attack is that in the CTR, we needed to modify the i^{th} ciphertext block to modify the i^{th} plaintext block, while in the CBC mode, we had to modify the $(i-1)^{th}$ ciphertext block to do the same.

CTR random access read/write attack

We learnt about how when the block cipher is used in the CTR mode of operation, we can perform a 'Random Access Read / Write' to a specific block. To modify the N^{th} byte of the plaintext, all we need to do is generate the N^{th} byte of the key stream, decrypt the cipher byte, encrypt with the new plaintext byte. Though this sounds handy, we learnt that this is not very secure. Assuming that such an 'edit()' function gets exposed to an attacker through an API abuse, we were able to modify the plaintext and as a result find what the original plain text was!

Public Key Cryptography

Some set of challenges deals with encryption using different keys (Asymmetric key encryption). Basically it starts with the key exchange algorithm - Diffie Hellman key exchange algorithm and moves on to the most important public key encryption algorithm - RSA. As with all other encryption algorithms RSA also suffers from various attacks.

Diffie Hellman Key Exchange

In public key cryptography public key is sent over a public channel and private key is kept secret. The adversary must not be able to identify the key to be used. In order to obtain secure key exchange Diffie Hellman key exchange algorithm plays a major role. We have implemented the key exchange algorithm as a client-server application where server sends the public key to the client. Then the client generates the key K and client sends value of X to the server. The server calculates the same value K using the value of X. Though it seems to be secure it is also been attacked.

Attacks on Diffie Hellman - MITM Attack

As already mentioned Diffie Hellman key exchange algorithm is designed to obtain secure key exchange. But it has also been subjected to numerous attacks. One such attack is MITM attack where the adversary injects new values for X and Y sent over the common channel. The parameter sent over the common channel was changed to p by the adversary. So now we have

$$K2 = p^y \bmod p = 0$$

$$K1 = p^x \bmod p = 0$$

So we have a common key to be 0. Thus the adversary knows the key to be 0 and performs decryption to obtain the desired plaintext.

Attacks on Diffie Hellman - Playing with 'g'

Similar to the previous attack we attacked the Diffie Hellman algorithm by injecting various values of g. In this attack we played with different values of g and obtained different values of keys.

for g=1 we obtained the value of key to be 1.

for g=p we obtained the value of key to be 0.

Thus the adversary gets to know the value of the key which is to be used for encryption and decryption purposes. Hence from the above two attacks, the adversary can modify the values of g and p sent over a common channel and implement the man in the middle attack.

Plain RSA

In challenge #39, we implemented the plain RSA just using simple integers.This algorithm consists of two parts - key generation part and an encryption part. We used two simple prime numbers and used the product of two primes as a part of a key. The public key is known to all. The plain RSA uses simple math for its encryption and decryption. RSA without padding can suffer due to many attacks which are discussed below. Hence RSA is generally combined with some padding scheme so as to avoid such kind of attacks.

Attacks on RSA - Broadcast Attack

In challenge #40, we implemented broadcast attack. This attack is based on certain assumptions:

1. The public key e should be very small (This is to calculate 1/e th power of the ciphertext captured)
2. The sender wants to send the same message to many receivers. So he uses different keys for encryption.
3. The modulus Ni used for different encryption should not have a common divisor.

This attack is based on Chinese Remainder Theorem(CRT) wherein the attacker captures three cipher-texts along with their public keys (same message encrypted with the same key) and uses CRT to solve for the number represented by the three cipher-texts. Finally taking the cube root (since e=3) gives the original plain text. This attack is based on certain assumptions and it is basically due to RSA unpadding.

Attacks on RSA - Unpadded message recovery attack

In challenge #41 we implemented the unpadded message recovery attack on RSA. This is similar to man in the middle attack(MITM). Since the ciphertext and the public key component e is known to the attacker the attacker chooses a random number and multiplies with the cipher text. The attacker also has access to the decryption oracle and hence obtains the new plain text. Then the attackers uses the new plain text to obtain the original plain text sent by the sender. This attack is also due to unpadding in RSA. Hence RSA suffers mostly from unpadding attacks. As a result of which we learnt that RSA padding is a must to prevent such attacks.

Secure Coding

Java apparently has a bunch of important security features embedded into the language itself like – automatic bounds checking (avoiding Buffer Overflow vulnerabilities), referencing null pointers are automatically taken care of, there are no explicit pointer manipulations like in C/C++ - and a couple more which make Java a relatively secure platform. The following is a documentation of the secure coding practices we have implemented in our project.

Do not operate on files in Shared directories [FIO00-J]

Most challenges involved reading text from external files. This calls for secure usage of input output methods with files, as external files cannot be trusted.

- It's not secure to operate on files that don't belong to what are known as secure directories – a secure directory is one in which only the current user and the system administrator can create/modify/delete files in it – any shared directory is not secure.
- Device files – We need to be aware of device files like the STDIN, STDOUT, mouse, etc. A secure program must ensure that it doesn't operate on any of the device files if it's not intended to.
- File links in a shared directory cannot be trusted. It is possible for someone else to change the target of the file link.
- Even if a file is checked to be a regular file and not a device file or a link before operating on it, it is possible for a Time of Check, Time of Use (TOCTOU) race condition.
- Check-Use-Check doesn't work either. It is possible to identify the fileKey of a file (if one exists) to check if the file opened is the file that's going to be read from. But some operating systems do not support fileKeys. Also, it is still possible for an attacker to do the TOCTOU attack.

File Related Errors [FIO02-J]

We always need to make sure we handle file related errors. For some reason, an operation on files, say, file.delete(); might not succeed - may be because the file is in use, or may be because the program does not have the permissions to do that. The right practice is to capture the return value of the delete() function and make sure that the file was actually deleted. In some cases, it is possible for file related errors to throw exceptions. In such cases, we need to make sure to catch the exceptions and react appropriately.

Create files with appropriate access permissions [FIO01-J]

Most systems are multi-user nowadays. So when we are creating files, we need to create them with appropriate access permissions for each kind of user that system might have. The constructors for file creation in Java usually do not have attributes to set the file access permissions.

So according to this rule, we need to use the java.nio package that has classes and functions that let us create files with appropriate access permissions. We can use this package to set permission in a POSIX supported platform like Linux. For Windows, we have used the functions in class File - setReadable(), setWritable() to set the permissions. We can also get the runtime context of the program and use the windows command line command 'attrib' to set the file permissions.

Validate Method Arguments [MET00-J]

We need to be careful with methods that take in arguments. Through these crypto challenges, we have learnt that though it might look like a function could only be called from within the program, that's not actually the case - there could be tens of other ways a function can be called from an outsider, say, because of an API abuse. So we always need to make sure to validate the arguments received by a function before operating on them. *Caller* validation of arguments might work, but not in situations like explained above. *Callee* validation should always be done and it is enough to do just the callee validation as it also reduces the number of places validation needs to be done.

Do not use deprecated / obsolete classes or methods [MET02-J]

Often, new features, bug fixes and stuff like that are added to any language. These are given out as updates to the users, like how Java gets updates quite often. Every time a new version is released, some bugs are fixed and better versions of certain functions are released that are compatible with the latest developments. So when a function has been declared as deprecated, we need to practice to not use that function anymore in our code. We need to keep abreast with the Java SE / EE Documentation for the latest version of Java and its updates.

Do not allow exceptions to expose sensitive information [ERR01-J]

In languages that have Exception handling procedures, like in Java, programmers always make good use of them to catch possible exceptions that could be thrown in a piece of code. While this is a good thing to do - to handle exceptions manually and take care of things if at all an exception occurs, it is also equally important, if not more, to make sure that we do not leak any information about the exceptions that occur during the program's runtime. There have been many attacks just because a program / application / system gave out specifics about the type of exception that occurred. While such information might be useful while the system is still in development, we should always make sure we don't give out any specific information about the runtime exceptions that could potentially lead to a greater advantage for the attacker.

Security check methods need to be private or final [MET03-J]

*W*e need to declare those functions that perform security checks need to be declared final, or they need to be private functions. But making these functions final, it will not be possible for subclasses to override

them, thereby assuring the correct functionality of the security checks. Or, making them private functions will render them inaccessible from outside that class, assuring the same.

Do not use finalizers [MET12-J]

In Java, the finalizer is called once an object is found to go out of scope and will be unreachable after that. It is called by the Garbage Collector to release any resource that might be in use by that object before its space is reclaimed. But calling the finalizer method explicitly will lead to a lot of problems that are unintentional. So it is advisable not to use finalizers to avoid any such unintentional issues.

Detect or Prevent Integer Overflow [NUM00J]

In all programming languages, there is a limit or range to every data type. For example, int varies from -2^31 to 2^31-1. When a piece of code performs arithmetic operations, it is possible for an intermediary or the final result to go beyond this range. This results in an Integer Overflow. We learnt how to prevent such overflows before the arithmetic operation is performed - nstead of using the normal arithmetic expressions like x + y, we have used the functions specified in this rule - safeAdd(), safeSubtract(), etc. that cautiously look for overflows.

Look for Divide by Zero errors [NUM02-J]

Division and remainder (modulus) operations have the possibility of resulting in a divide-by zero error. It is not in our usual coding practice to always check for such conditions - we usually just perform the division / remainder operation. But it is important to first check if an arithmetic operation, not just a division/remainder operation would possibly result in a divide by zero error and then decide whether or not to perform that operation.

Floating point variables as loop counters [NUM09-J]

Floating point numbers in Java are not always exact to the value they are supposed to represent. The value 0.1f would actually be slightly larger than 0.1. So using floating point numbers in loops like for and while as counters would result in an unexpected number of iterations rather than the desired output.

Buffer overflow with negative loop indices

In all of the challenges, we never had to get a user input at any point. But in trying to follow secure coding practices, we came across how a user input to a number that will later be used as a loop counter or an index could potentially lead to a buffer overflow. If a loop is desired to run 10 times, reading values into a buffer, and the initial value of the loop counter / index is a negative number instead of zero, then the loop condition i < 10 will stay true for more than 10 iterations, leading to a buffer overflow.

String Concatenation

In Java, concatenating String objects using the + operator will do the job. But the memory consumption in concatenating Strings using + is very high. s3 = s1 + s2 - this statement makes s3 occupy a new space in memory apart from the space occupied by s1 and s2. After this operation, we might not even use s1 and s2 ever again. In such cases, 2 times the required memory is consumed. We need to make use of the append() function in the class StringBuilder. This appends the new strings to the same memory location, thereby occupying only the required space. If in case, string concatenations happens inter-threads, then we need to use the class StringBuffer.

Try with Resources and The finally{} block

From Java 7, the usual try-catch for exception handling has been augmented with a feature called try-with-resources. In this kind of try block, we can give an object (that implements the Closeable interface) as an argument to the try block, and Java will itself call close() on the object when it exits the try block. Most times, there is memory leak or resource leaks because an opened connection or a stream is not closed properly. This is a secure way of avoiding such problems. The finally{} block after the catch block in a try-catch area, will *always* get executed irrespective of whether an exception occurred in the try block or not. In fact, it even gets executed when the control goes out of the function because of a return statement, or the loop breaks or continues. So we need try and make good use of the finally{} block to deallocate resources and other clean up code.

Do not catch NullPointerException or any of its ancestors [ERR08-J]

In places where a NullPointerException is suspected, we should not ever catch it with a try-catch block. If we suspect a particular object to be null at the point of running, we should rather check if it is indeed null, before operating on it, and return from the function if it is.

Do not suppress or ignore checked exceptions [ERR00-J]

If we are using try-catch blocks, and if an exception is caught, not acting upon it is not the best idea. We need to make the best use of such caught exceptions during runtime. Each catch block must be used to make sure that further execution of the program will not result in the same problem again. There are a couple of ways to make use of the catch blocks - Make the catch block interactive, Report the Exceptions, etc.

Do not encode non-character data as a string [STR03-J]

Non characters (higher ASCII) data should not be represented in a String object, simply because they are not meant to be. Not all bytes are characters and if we still use Strings to represent them, it will result in erroneous behavior and sometimes even data loss. So we should just operate on raw bytes.

Do not attempt comparisons with NaN [NUM07-J]

Performing complex arithmetic operations sometimes could result in NaN - Not a Number - this usually happens when we have a divide by zero error somewhere in the calculation or when mathematical functions like trigonometric functions result in infinity. In such cases, we should not perform direct comparisons with NaN. Instead, we need to be using the inbuilt functions for comparisons with NaN like Double.isNaN(), Integer.isNaN() and the rest.

Do not perform bitwise and arithmetic operations on the same data [NUM01-J]

According to this rule, integers should either be seen as a numeric value or as a collection of bits. So, we should not be performing bitwise operations and arithmetic operations on the

same data. Such operations would result in the correct, intended result, but it isn't a good practice as it reduces code readability.

Do not ignore values returned by methods [EXP00-J]

Often times, predefined functions in the packages that we import in a Java program, return values to the caller. Information such as if the function executed properly, if there were any exceptions thrown during its functioning and so on are returned back to the caller. For secure functioning of the program, it is important to make sure to look at the result returned by a function before proceeding.

Do not use Inner Classes

Java lets us create Inner Classes - classes inside other classes. Though at the high level, such inner classes would only be accessible from the class surrounding them, all code is finally converted to ByteCode. While exporting as bytecode, the JVM treats inner classes just like it would treat a regular class. So it becomes accessible within the entire package. It is a potential target vector for attackers. So we should restrict ourselves from using Inner Classes.

Make classes non-clonable and non-serializable

If cloning is enabled in a class, then an attacker could make a new instance of the class bypassing the constructors, just by making copies of the memory spaces, and could even redefine the clone method. To prevent this, we need to make classes non-clonable. Similarly, object serialization allows an object's state to be converted into a byte stream that contains information about its variables and functions, including private ones. So, object serialization should also be restricted.

Summary

Solving these Crypto challenges has helped us gain a greater level of practical knowledge about cryptography and has given us quite a deep insight into coding for crypto and the real world scenarios. Through this project, we got the opportunity to try and practically implement the cryptographic concepts, algorithms, schemes, and attacks that we learnt in class, giving us a deeper understanding of how things work. In trying to solve these challenges, we needed to learn the intricacies of the various crypto attacks we have documented above. We have learnt the do's and dont's when it comes to coding for crypto and this project has helped us better equip ourselves for coding related to cryptography. We also had the chance to learn some secure coding practices for Java - things that we wouldn't usually practice for general programming. We have learnt about various secure practices that we initially thought were trivial and about how they help us secure our system on the whole.

References

[1] The ASCII Table for reference: ascii.cl
[2] Hex-ASCII conversion for cross verification: http://www.rapidtables.com/convert/number/
[3] Base64 encoding & decoding for cross verification: https://www.base64decode.org/
[4] How AES works? : https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[5] How AES works? : https://www.youtube.com/watch?v=ayiOhApI6SM
[6] http://stackoverflow.com/questions/11051205/difference-between-java-util-random-and-java-security-securerandom
[7] http://docs.oracle.com/javase/7/docs/api/java/util/Random.html
[8] Hugo Krawczyk – How to Predict Congruential Generators
[9] https://jazzy.id.au/2010/09/20/cracking_random_number_generators_part_1.html
[10] cs.purdue.edu/homes/ninghui/courses/Fall05/lectures/355_ Fall05_lect04.pdf
[11] Padding Oracle Attack - https://www.youtube.com/watch?v=D-4mmc7frR4
[12] Padding Oracle Attack - https://www.youtube.com/watch?v=evrgQkULQ5U
[13] Random questions from security.stackexchange.com and stackoverflow.com
[14] http://www.javaworld.com/article/2076837/mobile-java/twelve-rules-for-developing-more-secure-java-code.html
[15] www.sei.cmu.edu/news/article.cfm?assetid=77817&article=268&year=2013
[16] securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java
[17] javacodegeeks.com/2015/06/java-programming-tips-best-practices-beginners.html

Appendix A - Parts of the Source Code (with secure coding documentation)

GitHub - https://github.com/RGirish/Security-Projects-Java

The following is a collection of snapshots of our documented code -

in AES-ECB

```
/*
 * FIO00-J. Do not operate on files in shared directories
 *
 * @reference https://www.securecoding.cert.org/confluence/display/java/
 *            FIO00-J.+Do+not+operate+on+files+in+shared+directories
 */


if (!isInSecureDir(inPath)) {
    System.out.println("File not in secure directory");
    return;
}

BasicFileAttributes inAttr = Files.readAttributes(inPath, BasicFileAttributes.class,
        LinkOption.NOFOLLOW_LINKS);
BasicFileAttributes outAttr = Files.readAttributes(outPath, BasicFileAttributes.class,
        LinkOption.NOFOLLOW_LINKS);

// Check if the file is a regular file and not a FIFO file or a
// device file.
if (!inAttr.isRegularFile() || !outAttr.isRegularFile()) {
    System.out.println("Not a regular file");
    return;
}
```

in Break Rep

```
/*****************************************************/

/*
 * FIO01-J. Create files with appropriate access permissions
 *
 * @reference
 * https://www.securecoding.cert.org/confluence/display/java/FIO01-J
 * .+Create+files+with+appropriate+access+permissions
 */

// Throw a warning message if file already exists and try to delete
// it, rather than overwrite it.
// No need to check if it is a directory, as we're going to delete
// and recreate it anyway!
if (outFile.exists()) {
    System.out.println("File already exists. Trying to delete it now...");
    boolean result = outFile.delete();
    if (!result) {
        System.out.println("Already existing file/directory with name '" + outFilename
                + "' not deleted successfully!");
        return;
    } else {
        System.out.println("Deleted successfully!");
    }
}
@SuppressWarnings("unused")
boolean result = outFile.createNewFile();
// no need to check result which will be false only if the file
// already exists which is already checked before!
outFile.setReadable(true, true);
outFile.setWritable(true, true);
// For java versions <1.7, use this -
// Runtime.getRuntime().exec("attrib -r " + outFilename);

/*****************************************************/
```

eating Key XOR

```java
/*
 * ERR08-J. Do not catch NullPointerException or any of its ancestors
 *
 * @reference
 * https://www.securecoding.cert.org/confluence/display/java/ERR08-J.+Do
 * +not+catch+NullPointerException+or+any+of+its+ancestors
 *
 * Do not catch Exception. Catch the individual exception types.
 */
catch (NoSuchAlgorithmException e) {
    System.out.println("NoSuchAlgorithmException " + e.getMessage());
    return null;
} catch (NoSuchPaddingException e) {
    System.out.println("NoSuchPaddingException " + e.getMessage());
    return null;
} catch (InvalidKeyException e) {
    System.out.println("InvalidKeyException " + e.getMessage());
    return null;
} catch (IllegalBlockSizeException e) {
    System.out.println("IllegalBlockSizeException " + e.getMessage());
    return null;
} catch (BadPaddingException e) {
    System.out.println("BadPaddingException " + e.getMessage());
    return null;
}
```

in ECB - CBC Detection

```java
/*
 * FIO02-J. Detect and handle file-related errors
 *
 * @reference
 * https://www.securecoding.cert.org/confluence/display/java/FIO02-J.+
 * Detect+and+handle+file-related+errors
 */
if (file.exists()) {
    if (!file.isDirectory()) {
        // File exists and it is not a directory. Now we can start
        // working with it.

        if (file.canRead()) {
            try {
                FileReader fileReader = new FileReader(file);
```

in Detect AES ECB

```java
/* @reference https://www.securecoding.cert.org/confluence/display/java/
 * NUM51-J.+Do+not+assume+that+the+remainder+operator+always+returns+a+
 * nonnegative+result+for+integral+operands*/
public static int mod(int a,int b){
    int c=a%b;
    return (c<0)? -c : c;
}
```

```java
/* @refernce https://www.securecoding.cert.org/confluence/display/java/
 * EXP51-J.+Do+not+perform+assignments+in+conditional+expressions
 * K1 and K2 are not assigned at conditional expressions */
if(K1==K2)
    System.out.println("\nKeys are same");
```

in Diffie Hellman

```java
/*
 * EXP52-J. Use braces for the body of an if, for, or while statement
 *
 * @reference https://www.securecoding.cert.org/confluence/display/java/
 * EXP52-J.+Use+braces+for+the+body+of+an+if%2C+for%2C+or+while+
 * statement A for loop with one statement inside is enclosed within
 * braces
 */
if (s.length() % 32 == 0) {
    // No Padding
} else {
```

in ECB - CBC Detection

```java
public static String encryptionOracle(String plainText, String key) {
    /*
     * MET00-J. Validate method arguments
     *
     * @reference
     * https://securecoding.cert.org/confluence/display/java/MET00-J.+
     * Validate+method+arguments
     */
    if (plainText == null) {
        System.out.println("Plain Text cannot be NULL.");
        return null;
    }
    if (key == null) {
        System.out.println("Key cannot be NULL.");
        return null;
    }
    if (key.length() != 16) {
        System.out.println("Key should be 16 Bytes (hex characters) long.");
        return null;
    }
    plainText = appendRandomPrefixAndSuffixToPlainText(asciiToHex(plainText));
```

in ECB - CBC Detection

```java
/*
 * @reference
 * https://securecoding.cert.org/confluence/display/java/NUM00-J.+Detect+or+
 * prevent+integer+overflow
 *
 * The following are functions that perform the basic arithmetic operations
 * of subtraction, multiplication and division according to the secure
 * coding standards - by avoiding the Integer Overflow Exception.
 */

static final int safeSubtract(int left, int right) throws ArithmeticException {
    if (right > 0 ? left < Integer.MIN_VALUE + right : left > Integer.MAX_VALUE + right) {
        throw new ArithmeticException("Integer overflow");
    }
```

in ECB - CBC Detection

```java
/*
 * NUM02-J. Ensure that division and remainder operations do not result
 * in divide-by-zero errors
 *
 * @reference
 * https://securecoding.cert.org/confluence/display/java/NUM02-J.+Ensure
 * +that+division+and+remainder+operations+do+not+result+in+divide-by-
 * zero+errors
 */
if (right == 0) {
    throw new ArithmeticException("Divide By Zero");
```

in ECB - CBC Detection

```java
/*
 * ERR01-J. Do not allow exceptions to expose sensitive information
 *
 * @reference
 * https://www.securecoding.cert.org/confluence/display/java/ERR01-J.+Do
 * +not+allow+exceptions+to+expose+sensitive+information
 *
 * Instead of giving out specifics about the exact Exception that has
 * occurred, we're giving out a general error message so that this does
 * not help the adversary gain any advantage.
 */
catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyExcept
        | BadPaddingException e) {
    System.out.println("An error occured! Please try again.");
}
```

in ECB Cut and Paste Attack

```java
/* @reference https://www.securecoding.cert.org/confluence/display/java/
 * EXP53-J.+Use+parentheses+for+precedence+of+operation
 * Parentheses have been employed for the expression e*i % et == 1*/
    if((e*i)%et == 1){
        break;
```

in RSA

```java
private final static boolean isInSecureDir(Path file, UserPrincipal user, int s
    /*
     * MET03-J. Methods that perform a security check must be declared
     * private or final
     *
     * @reference
     * https://securecoding.cert.org/confluence/display/java/MET03-J.+
     * Methods+that+perform+a+security+check+must+be+declared+private+or+
     * final
     */
    if (!file.isAbsolute()) {
```

in Break Repeating Key XOR

```java
} catch (ArithmeticException e) {
    /*
     * ERR00-J. Do not suppress or ignore checked exceptions
     *
     * @reference
     * https://www.securecoding.cert.org/confluence/display/java/
     * ERR00-J.+Do+not+suppress+or+ignore+checked+exceptions
     */
    System.out.println("An Exception occurred.");
    return null;
}
```

in ECB Detection

```java
/*
 * STR03-J. Do not encode noncharacter data as a string
 *
 * @reference
 * https://www.securecoding.cert.org/confluence/display/java/STR03-J.+Do
 * +not+encode+noncharacter+data+as+a+string
 *
 * The ciphertext formed after encryption will not contain ASCII
 * printable characters. So we need to operate on raw bytes rather than
 * convert into a String object.
 */
byte[] cipherByes = padAndEncryptTheString(processedString, asciiToHex(asciiKey), iv);
```

in CBC Bit Flipping

```java
/*
 * Cloning is disabled for security reasons.
 * (non-Javadoc)
 * @see java.lang.Object#clone()
 */
public final Object clone() throws java.lang.CloneNotSupportedException {
    throw new java.lang.CloneNotSupportedException();
}

/*
 * Object Serialization is disabled for security reasons.
 */
private final void writeObject(ObjectOutputStream out) throws java.io.IOException {
    throw new java.io.IOException("Object cannot be serialized");
}
```

in CBC Bit Flipping

```
/*
 * Always use StringBuilder or StringBuffer for String concatenation.
 *
 * Concatenating String objects using the + operator will occupy twice
 * the required memory. Instead, we need to make use of the append()
 * function in the class StringBuilder or StringBuffer.
 */
StringBuilder s = new StringBuilder(hexString);
if (s.length() % 32 != 0) {
    int diff = (((blockSize * 2) - (s.length() % 32)) / 2);
    for (int i = 0; i < diff; i++) {
        s.append("0");
        s.append(Integer.toHexString((diff)));
    }
}
```

in CBC Bit Flipping

Appendix B - Attacks and crypto learnings

- Always Operate On Raw Bytes while coding for crypto.
- Always use Character Encoding styles to 'pretty-print' and store bytes that are not ASCII.
- In Java, always use java.security.SecureRandom rather than java.util.Random for generating Random numbers as the latter is quite easily predictable and is not cryptographically secure.
- The ECB mode of operation of block ciphers is almost as good as not encrypting your data.
- A ciphertext-only passive attack could lead to detecting whether the ECB mode has been used in a ciphertext or not.
- Knowing just the first few bytes, or a few bytes in the middle of the plaintext (as is usually the practical scenario), it is very easily possible to decrypt an entire ECB encrypted cipher text.
- ECB is susceptible to a Cut and Paste attack that is exactly how easy as it sounds like - just replacing a block of ciphertext with some other piece of ciphertext will lead to a different plain text message.
- It is very easy to detect if a piece of plaintext has been encrypted in the ECB mode or the CBC mode with just an access to an Encryption Black Box.
- The CBC mode of operation is susceptible to bit flipping attacks that could potentially lead to the decryption of an attacker-desired plaintext message. But it is far better to detect such active attacks with CBC than with ECB.
- The CTR mode of operation also is susceptible to the same kind of bit flipping attacks.
- While the CBC mode of operation makes it possible to encrypt irregularly sized messages, since that requires a padding, it leads to a padding oracle attack in the presence of a padding oracle. This leads to complete decryption of the ciphertext.
- When the CBC mode is used, and we decide to use the key itself as the IV, it is very easy to steal the the key with a very simple active attack.
- When the CTR mode is used, we should never ever *ever* use the same nonce twice. Fixed nonces makes things very easy for the attacker to decrypt the entire ciphertext and get hold of almost all of the plaintext.
- The CTR mode, if implemented, should not come with a Random Access Read/Write function. If it does, then we need to make sure it never leaks out by any means as it leads to complete decryption of the ciphertext.
- The Diffie Hellman key exchange algorithm must use parametric signatures in order to avoid MITM attacks.
- The RSA algorithm used for asymmetric key encryption must use some kind of padding schemes in order to avoid broadcast attacks and unpadded message recovery attack.
- On the whole every encryption algorithm employed in practice doesn't provide perfect security. Hence it is the matter of how long the security exists.

Appendix C - List of Secure Coding Practices

- Do not operate on files in Shared directories.
- Always handle File related errors.
- Create files with appropriate access permissions.
- Validate method arguments.
- Do not use deprecated / obsolete classes or methods.
- Do not allow exceptions to expose sensitive information.
- Methods performing security checks need to be private or final.
- Do not use finalizers.
- Detect or prevent Integer Overflow.
- Look for Divide by Zero errors.
- Do not use floating point variables as loop counters.
- Be aware of Buffer Overflow possibilities with negative loop indices.
- For String concatenation in Java, always use StringBuilders or StringBuffers, not + operator.
- Try with Resources and the finally{} block are great means for clean up code.
- Do not catch NullPointerException or any of its ancestors.
- Do not suppress or ignore checked exceptions.
- Do not encode non-character data as a string.
- Do not attempt comparisons with NaN.
- Do not perform bitwise and arithmetic operations on the same data.
- Do not ignore values returned by methods.
- Do not use Inner Classes.
- Make classes non-clonable and non-serializable.