Lab#1
Matthew Langlois - 7731813
Monday Sept. 25

Part 1

The following code is the encryption/decryption program written in Javascript. Some sample commands have been provided at the top of the file which will generate the keys. I have also attached the file to my submission.

```
const readline = require('readline');
const process = require('process');
const crypto = require('crypto')
const ursa = require('ursa');
const fs = require('fs');
const path = require('path');

//Commands to run...
//genkeys keys/aprv keys/apub keys/bprv keys/bpub
//encrypt test/file password keys/aprv keys/bpub
//decrypt test/file.enc test/file.enc.key test/file.sig keys/apub
↪  keys/bprv

const commands = {
    encrypt: (file, encryptionKey, senderPrivKey, recipientPubKey) => {
        //read input file
        let input = fs.readFileSync(file);

        //Sign and Hash file
        let sigKey =
        ↪  ursa.createPrivateKey(fs.readFileSync(senderPrivKey));
        let signature = sigKey.hashAndSign('sha256', input);

        //Encrypting file
        let cipher = crypto.createCipher('aes-256-cbc', encryptionKey);
        let encrypted =
        ↪  Buffer.concat([cipher.update(input),cipher.final()])

        //Encrypting symmetric key
        let recPubKey =
        ↪  ursa.createPublicKey(fs.readFileSync(recipientPubKey));
        let encryptedKey = recPubKey.encrypt(encryptionKey);

        //Write files to disk
        fs.writeFileSync(`${file}.enc`, encrypted);
        fs.writeFileSync(`${file}.enc.key`, encryptedKey);
        fs.writeFileSync(`${file}.sig`, signature);

        //Let the user know the processs id done
        console.log(`Done saving files! ${file}.enc ${file}.key.enc
         ↪  ${file}.asc`);
    },

    decrypt: (encFile, encSymKey, signature, senderPubKey,
     ↪  recipientPrivKey) => {

        //Decrypt symmetric key
        let privKey =
         ↪  ursa.createPrivateKey(fs.readFileSync(recipientPrivKey));
        let symKey = privKey.decrypt(fs.readFileSync(encSymKey));

        //Decrypt file
        let decipher = crypto.createDecipher('aes-256-cbc',
         ↪  symKey.toString());
        let decrypted =
         ↪  Buffer.concat([decipher.update(fs.readFileSync(encFile)) ,
         ↪  decipher.final()]);

        //verify signed file
        let senderSigKey =
         ↪  ursa.createPublicKey(fs.readFileSync(senderPubKey));
        let verified = senderSigKey.hashAndVerify('sha256', decrypted,
         ↪  fs.readFileSync(signature));

        console.log(`Verified: ${verified}`);

        //Write out decrypted file
        if (verified) {
           let output = `${encFile.substr(0,
            ↪  encFile.lastIndexOf('.'))}.dec`;
           fs.writeFileSync(output, decrypted);
           console.log(`File saved to: ${output}`);
        }
    },

    genkeys: (alicePriv, alicePub, bobPriv, bobPub) => {
        //Create Alice's key
        let alice = ursa.generatePrivateKey(1024);
        let alicePrivKey= alice.toPrivatePem();
        let alicePubKey = alice.toPublicPem();

        //Create Bob's keypair
        let bob = ursa.generatePrivateKey(1024);
        let bobPrivKey= bob.toPrivatePem();
        let bobPubKey = bob.toPublicPem();

        //Write keyfiles to disk
        fs.writeFileSync(alicePriv, alicePrivKey);
        fs.writeFileSync(alicePub, alicePubKey);
        fs.writeFileSync(bobPriv, bobPrivKey);
        fs.writeFileSync(bobPub, bobPubKey);

        //Let the user know that the operation is complete
        console.log(`Keys generated and saved to disk!`);
    }

};

readline.createInterface({
    input: process.stdin,
    output: process.stdout,
}).on('line', line => {
    const [command, ...args] = line.split(' ');
    const action = commands[command];
    if (!action) {
        console.error(`Invalid command: ${command}`);
        return;
    }
    action.apply(null, args);
});
```

To hash and sign I use the sender's private key along with SHA-256. I chose to use SHA-256 since other algorithms have such as SHA-1 have proof-of-concepts which break them.
To achieve the encryption task I chose to use AES-256-CBC. Advanced Encryption Standard (or AES for short) is one of the industry standard used for symmetric encryption. In this case I chose to use a 256 bit key for maximum security.
Once symmetric encryption is complete the key is then encrypted using RSA. To do so I chose to use the package "ursa" which offers some helper methods to access the standard Crypto methods in Node. Essentially I'm using alice's public key to encrypt the data and then Alice can decrypt it with her private key.

Finally I've also created a helper method to generate 2 sets of private and public keypairs. In this example I use Alice and Bob.

Part 2

To complete part 2 we started with a scan for vulnerabilities using NMAP using the vuln script like so: `nmap -script vuln 137.122.47.142`. NMap came back reporting that IIS 7.5 was running on port 80 and was vulnerable to `MS15-034`. Furthermore there are other exploits which exist for this version of IIS however most of them rely on other services such as IIS-FTP server.
There were 2 public exploits which made use of this vulnerability. One was a denial of service and the other was a memory leak. Both of which could be exploited using Metasploit (which is publicly available on GitHub):
Finally, to patch these vulnerabilities the best course of action would be applying the patches provided by the vendor. Some other preventative measures could include deploying a firewall which could block ports which arent meant to be open. For example if this ISS instance wasn't supposed to be public then port 80 could be closed off.