Project 1: Cryptography

This project is split into two parts, with the first checkpoint due on **Wednesday, September 9** at **6:00pm** and the second checkpoint due on **Friday, September 18** at **6:00pm**. The first checkpoint is worth 4%of your total grade, and the second checkpoint is worth 8%. We strongly recommend that you get started early. Each semester everyone will be given ONE late extension that allows you to turn in up to one assignment up to 24 hours after the due date. Late work will not be accepted after 24 hours past the due date.

This is a group project; you SHOULD work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If have trouble forming a team, post to Piazza's partner search forum.

The code and other answers your group submits must be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions MUST be submitted electronically in any of the group member's svn directory, following the submission checklist given at the end of each checkpoint. Details on the filename and submission guideline is listed at the end of the document.

"Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."

– Bruce Schneier

Introduction

In this project, you will be using cryptographic libraries to decrypt multiple type of ciphers, furthermore breaking them, and launch attacks on widely used cryptographic hash functions, including length-extension attacks and collision attacks. In 1.2, you will be decrypting ciphers with given ciphertexts and key values. Then, you will use the same technique to break a weak cipher with a limited key space. In 1.3, you will start out with a small exercise that uses a hash function to observe the avalanche effect, and then build a weak hash algorithm and find collision on a given string. In 2.1, we will guide you through attacking the authentication capability of an imaginary server API. The attack will exploit the length-extension vulnerability of hash functions in the MD5 and SHA family. In 2.2, you will be attacking RSA, a widely-use public key encryption algorithm. In 2.3, you will use a cutting-edge tool to generate different messages with the same MD5 hash value (collisions). You'll then investigate how that capability can be exploited to conceal malicious behavior in software. Lastly, in 2.4, you will use collision attack to undermined integrity of a set of documents.

Objectives:

- Become familiar with existing cryptographic libraries and how to utilize them
- Understand pitfalls in cryptography and appreciate why you should not write your own cryptographic library
- Execute classic cryptographic attack on md5 and other broken cryptographic libraries
- Appreciate why you should use HMAC-SHA256 as a substitute for common hash functions.

Guidelines

- You SHOULD work in a group of 2.
- You SHOULD use Python.
- You MAY use any API or library you like as long as you cite them in program comments. You SHOULD NOT create your own crypto library.
- Your answers may or may not be the same as your classmates'.
- All the necessary files to start the project will given under the folder called "mp1" in your SVN directory. We've also generated some empty files for you to submit your answers in. You MUST submit your answers in the provided files; we will only grade what's there!

1 Checkpoint 1 (35 points)

1.1 Python tutorial (5 points)

In this section, you will be writing several python scripts to do string encoding and manipulations needed to correctly read our input files and submit your answers.

1.1.1 Reading .hex files

In the later parts of this mp, you will be reading in .hex file, which is a plaintext files containing an ascii string representing a single hexadecimal number. This is an example content of a .hex file:

```
3dab821d92b5ca7f48beee066996b8abc82f7e5646a0561710ea5bc11c80d
```

The following python code snippet will read the content of the file as a string and store it in `file_content`:

```
# strip() remove any leading or trailing whitespace characters
with open('file_name') as f:
    file_content = f.read().strip()
```

From here, there's a number of things that you could do depending on the cryptographic library that you are using. You may need to use different type of representation of the data, but here we list the most common conversions that you may need:

```
# parse the hexstring into binary array of the hexadecimal number
binary_content = file_content.decode('hex')

# parse the hexstring into integer
integer_parsed = int(file_content,16)

# parse the integer to a hexstring and remove the leading '0x'
hex(integer_parsed)[2:]

#parse the integer to a binary string and remove the leading '0b'
bin(integer_parsed)[2:]
```

1.1.2 Exercise

Files

1. `1.1.2_value.hex`: an ascii string representing a hexadecimal value

Based on what you learn in the last section, we want to to convert the given value into different representations and submit them in the specified files.

What to submit

1. Convert the value in `1.1.2_value.hex` to decimal and submit it in `sol_1.1.2_decimal.txt`
2. Convert the value in `1.1.2_value.hex` to binary string and submit it in `sol_1.1.2_binary.txt`

1.2 Symmetric Encryption, Public Key Encryption, and Cryptographic Hashes

In this section, you will be writing your own cryptographic library to decrypt a substitution cipher, and using existing cryptographic libraries to experiment with a symmetric encryption called AES and a public key encryption called RSA.

1.2.1 Substitution Cipher (5 points)

Files

1. `1.2.1_sub_key.txt`: key
2. `1.2.1_sub_ciphertext.txt`: ciphertext

`sub_key.txt` contains a permutation of the 26 upper-case letters that represents the key for a substitution cipher. Using this key, the *i*th letter in the alphabet in the plaintext has been replaced by the *i*th letter in `1.2.1_sub_key.txt` to produce ciphertext in `1.2.1_sub_ciphertext.txt`. For example, if the first three letters in your `1.2.1_sub_key.txt` are `ZDF`..., then all `A`s in the plaintext have become `Z`s in the ciphertext, all `B`s have become `D`s, and all `C`s have become `F`s. The plaintext we encrypted is a clue from the gameshow Jeopardy and has only upper-case letters, numbers and spaces. Numbers and spaces in the plaintext were not encrypted. They appear exactly as they did in the plaintext.

What to submit Submit the plaintext, which is obtained using the key `1.2.1_sub_key.txt` to decrypt `1.2.1_sub_ciphertext.txt`, in the file `sol_1.2.1.txt`.

1.2.2 AES: Decrypting AES (5 points)

Files

1. `1.2.2_aes_key.hex`: key
2. `1.2.2_aes_iv.hex`: initialization vector
3. `1.2.2_aes_ciphertext.hex`: ciphertext

`1.2.2_aes_key.hex` contains a 256-bit AES key represented as an ascii string of hexadecimal values. `1.2.2_aes_iv.hex` contains a 128-bit Initialization Vector in a similar representation. We encrypted a Jeopardy clue using AES in CBC mode with this key and IV and wrote the resulting ciphertext (also stored in hexadecimal) in `1.2.2_aes_ciphertext.hex`.

Cryptographic Library

For this checkpoint, we recommend PyCrypto, an open-source crypto library for python. PyCrypto can be installed using pip with `sudo pip install pycrypto` or by going to their website at `https://www.dlitz.net/software/pycrypto/`.

What to submit Decrypt the ciphertext using the provided information and submit the plaintext in `sol_1.2.2.txt`.

1.2.3 AES: Breaking A Weak AES Key (5 points)

Files

1. `1.2.3_aes_weak_ciphertext.hex`: ciphertext

As with the last task, we encrypted a Jeopardy clue using 256-bit AES in CBC and stored the result in hexadecimal in the file `1.2.3_aes_weak_ciphertext.hex`. For this task, though, we haven't supplied the key. All we'll tell you about the key is that it is 256 bits long and its 251 most significant (leftmost) bits are all 0's. The initialization vector was set to all 0s. First, find all plaintexts in the given key space. Then, you will review the plaintexts to find the correct plaintext that is in Jeopardy clue and the corresponding key.

What to submit Find the key of the appropriate plaintext and submit it as a hex string in `sol_1.2.3.hex`.

1.2.4 Decrypting cipher text with RSA (5 points)

Files

1. `1.2.4_private_key.hex`: RSA private key (d) as hexdecimal string
2. `1.2.4_RSA_modulo.hex`: RSA modulo (N) as hexadecimal string
3. `1.2.4_RSA_ciphertext.hex`: an encrypted prime number that is encrypted with 1024-bit RSA as hexadecimal string

In this part we use 1024 bit RSA to encrypt a prime number with your public key and stored it in `1.2.4_RSA_ciphertext.hex` as a hex string. Using a cryptographic library of your choice, the given private key, and RSA modulo, decrypt the ciphertext and find the prime number.

What to submit Decrypt the ciphertext and submit the prime number as a hex string in `sol_1.2.4.hex`.

1.3 Hash Functions

This section will give you a chance to explore cryptographic hashing using existing cryptographic libraries and illustrate the pitfall in writing your own cryptographic functions.

1.3.1 Avalanche Effect (5 points)

Files

1. `1.3.1_input_string.txt`: original string
2. `1.3.1_perturbed_string.txt`: perturbed string

`1.3.1_input_string.txt` contains another Jeopardy clue in ASCII.
`1.3.1_perturbed_string.txt` is an exact copy of this string with one bit flipped. We're going to use these two strings to demonstrate the avalanche effect by generating the SHA-256 hash of both strings and counting how many bits are different in the two results (a.k.a. the Hamming distance.)
What are their SHA-256 hashes? Verify that they're different.
(`$ openssl dgst -sha256 1.3.1_input_string.txt 1.3.1_perturbed_string.txt`)

What to submit Compute the number of bits different between the two hash outputs and submit it as a hex string in `sol_1.3.1.hex`.

1.3.2 Weak Hashing Algorithm (5 points)

Files

1. `1.3.2_input_string.txt`: input string

Below you'll find the pseudocode for a weak hashing algorithm we're calling `WHA`. It operates on bytes (block size 8-bits) and outputs a 32-bit hash.

```
WHA:
Input{inStr: a binary string of bytes}
Output{outHash: 32-bit hashcode for the inStr in a series of hex values}
Mask: 0x3FFFFFFF
outHash: 0
for byte in input
  intermediate_value = ((byte XOR 0xCC) Left Shift 24) OR
                       ((byte XOR 0x33) Left Shift 16) OR
                       ((byte XOR 0xAA) Left Shift 8) OR
                       (byte XOR 0x55)
  outHash =(outHash AND Mask) + (intermediate_value AND Mask)
return outHash
```

First, you'll need to implement `WHA` in the language of your choice. Here are some sample inputs to test your implementation: `WHA("Hello world!") = 0x50b027cf` and `WHA("I am Groot.")=0x57293cbb`

In the file `1.3.2_input_string.txt`, you'll find another Jeopardy clue (surprise!) Your goal is to find another string that produces the same `WHA` output as this Jeopardy clue. In other words, demonstrate that this hash is not second preimage resistant.

What to submit Find a string with the same `WHA` output as `1.3.2_input_string.txt` and submit it in `sol_1.3.2.txt`. Also, submit the code for WHA algorithm code with the name `wha_collision.py`. (Given WHA code format is in python but any other preferred programming language is accepted.)

Checkpoint 1: Submission Checklist

Inside your mp1 directory svn, you will have the auto-generated files named as below. Make sure that your answers for all tasks up to this point are submitted in the following files before **Wednesday, September 9** at **6:00pm**:

SVN Directory

`https://subversion.ews.illinois.edu/svn/fa15-cs461/NETID/mp1`

File Format

All submitted files should be submitted as plaintext. The autograder runs on Unix, so students are encourage to validate in EWS that the files can be read as you intended. Any lines in your submission that begins with '#' will be ignored.

example content of .txt solution file

```
SPN WMKTQIW QR SPBW HQGRSEMW HQVS QY VEKW ENAENWNRS VEVZQR
HVWSBIN QSPNE PBWSQEBH LBRZFQKW
```

example content of .hex solution file

```
3dab821d92b5ca7f48beee066996b8abc82f7e5646a0561710ea5bc11c80d
```

Team Members

`partners.txt` : a text file containing netIDs of both members, one netid per line. Place the student's netID, whose directory contain your project submission, at the top of the file.

example content of partners.txt

```
netid1
netid2
```

List of solution files that must be submitted for checkpoint 1

- `partners.txt`
- `sol_1.1.2_decimal.txt`
- `sol_1.1.2_binary.txt`
- `sol_1.2.1.txt`
- `sol_1.2.2.txt`
- `sol_1.2.3.hex`
- `sol_1.2.4.hex`
- `sol_1.3.1.hex`
- `sol_1.3.2.txt`
- `wha_collision.py (or any other preferred language format)`

2 Checkpoint 2 (70 points)

Note: There is a component of section 2.4 that MUST be submitted by **3:30PM** on Friday, September 18. Please read ahead and plan accordingly.

2.1 Length Extension (15 points)

In most applications, you should use MACs such as HMAC-SHA256 instead of plain cryptographic hash functions (e.g. MD5, SHA-1, or SHA-256), because hashes, also known as digests, fail to match our intuitive security expectations. What we really want is something that behaves like a pseudorandom function, which HMACs seem to approximate and hash functions do not.

One difference between hash functions and pseudorandom functions is that many hashes are subject to *length extension*. All the hash functions we've discussed use a design called the Merkle-Damgård construction. Each is built around a *compression function* f and maintains an internal state s, which is initialized to a fixed constant. Messages are processed in fixed-sized blocks by applying the compression function to the current state and current block to compute an updated internal state, i.e. $s_{i+1} = f(s_i, b_i)$. The result of the final application of the compression function becomes the output of the hash function.

A consequence of this design is that if we know the hash of an n-block message, we can find the hash of longer messages by applying the compression function for each block $b_{n+1}, b_{n+2}, \ldots$ that we want to add. This process is called length extension, and it can be used to attack many applications of hash functions.

2.1.1 Experiment with Length Extension in Python

To experiment with this idea, we'll use a Python implementation of the MD5 hash function, though SHA-1 and SHA-256 are vulnerable to length extension in the same way. You can download the `pymd5` module at `https://subversion.ews.illinois.edu/svn/fa15-cs461/_shared/mp1/pymd5.py` and learn how to use it by running `$ pydoc pymd5`. To follow along with these examples, run Python in interactive mode (`$ python -i`) and run the command `from pymd5 import md5, padding`.

Consider the string "Use HMAC, not hashes". We can compute its MD5 hash by running:

```
m = "Use HMAC, not hashes"
h = md5()
h.update(m)
print h.hexdigest()
```

or, more compactly, `print md5(m).hexdigest()`. The output should be:

```
3ecc68efa1871751ea9b0b1a5b25004d
```

MD5 processes messages in 512-bit blocks, so, internally, the hash function pads *m* to a multiple of that length. The padding consists of the bit 1, followed by as many 0 bits as necessary, followed by a 64-bit count of the number of bits in the unpadded message. (If the 1 and count won't fit in the current block, an additional block is added.) You can use the function `padding(`*`count`*`)` in the `pymd5` module to compute the padding that will be added to a *`count`*-bit message.

Even if we didn't know `m`, we could compute the hash of longer messages of the general form `m + padding(len(m)*8) +` *`suffix`* by setting the initial internal state of our MD5 function to `MD5(m)`, instead of the default initialization value, and setting the function's message length counter to the size of *m* plus the padding (a multiple of the block size). To find the padded message length, guess the length of *m* and run `bits = (`*`length_of_m`*` + len(padding(`*`length_of_m`*`*8)))*8`.

The `pymd5` module lets you specify these parameters as additional arguments to the `md5` object:

```
h = md5(state="3ecc68efa1871751ea9b0b1a5b25004d".decode("hex"), count=512)
```

Now you can use length extension to find the hash of a longer string that appends the suffix "Good advice". Simply run:

```
x = "Good advice"
h.update(x)
print h.hexdigest()
```

to execute the compression function over `x` and output the resulting hash. Verify that it equals the MD5 hash of `m + padding(len(m)*8) + x`. Notice that, due to the length-extension property of MD5, we didn't need to know the value of `m` to compute the hash of the longer string—all we needed to know was `m`'s length and its MD5 hash.

This component is intended to introduce length extension and familiarize you with the Python MD5 module we will be using; you will not need to submit anything for it.

2.1.2 Conduct a Length Extension Attack

Files

1. `2.1.2_query.txt`: query
2. `2.1.2_command3.txt`: command3

One example of when length extension causes a serious vulnerability is when people mistakenly try to construct something like an HMAC by using $hash(secret \parallel message)$, where $\parallel$ indicates concatenation. For example, Professor Vuln E. Rabble has created a web application with an API that allows client-side programs to perform an action on behalf of a user by loading URLs of the form:

`http://cs461ece422.org/project1/api?token=b301afea7dd96db3066e631741446ca1`
`&user=admin&command1=ListFiles&command2=NoOp`

where `token` is MD5(*user's 8-character password* || `user=....` [*the rest of the URL starting from* `user=` *and ending with the last command*]).

Text files with the query of the URL `2.1.2_query.txt` and the command line to append `2.1.2_command3.txt` will be provided. Using the techniques that you learned in the previous section and without guessing the password, apply length extension to create a new query in the URL ending with command specified in the file, `&command3=DeleteAllFiles`, that is treated as valid by the server API. You have permission to use our server to check whether your command is accepted.

Hint: You might want to use the `quote()` function from Python's `urllib` module to encode non-ASCII characters in the URL.

Historical fact: In 2009, security researchers found that the API used by the photo-sharing site Flickr suffered from a length-extension vulnerability almost exactly like the one in this exercise.

What to submit A Python 2.x script named `len_ext_attack.py` and text file `sol_2.1.2.txt`. Python script should perform such that:

1. Accepts a valid query in the URL as a file input and parse the file.
2. Modifies the query in the URL so that it will execute the `DeleteAllFiles` command as the user.

The text file should contained the updated query

2.2 Breaking the RSA (20 points)

Files

1. `2.2_ciphertext.hex`: ciphertext in hex string
2. `2.2_public_key.hex`: public key in hex string
3. `2.2_modulo.hex`: RSA modulo in hex string

Let's see if you can break the RSA. You are given a ciphertext in `2.2_ciphertext.hex` that was encrypted with 1024-bit RSA using the key in 2.2_public_key.hex and the modulo in 2.2_modulo.hex. You do not know the necessary key to decrypt this ciphertext, but you do know that there is a weakness in this RSA key pair that makes it vulnerable to one of the attacks discussed in class (Please do not ask for the computation time and power consumption of our machine). The plaintext that was encrypted is a decimal integer number < the RSA modulo with a certain pattern that should be obvious when you decrypt it (The pattern is visible when the number is in its decimal representation, not hex). Your task is to find the private key and the decrypt the ciphertext to get the plaintext.

What to submit Submit the decrypted number as a hex string in `sol_2.2.hex`

2.3 MD5 Collisions (15 points)

MD5 was once the most widely used cryptographic hash function, but today it is considered dangerously insecure. This is because cryptanalysts have discovered efficient algorithms for finding *collisions*—pairs of messages with the same MD5 hash value.

The first known collisions were announced on August 17, 2004 by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Here's one pair of colliding messages they published:

Message 1:

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70
```

Message 2:

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70
```

Convert each group of hex strings into a binary file.
(On Linux, run `$ xxd -r -p file.hex > file`.)

1. What are the MD5 hashes of the two binary files? Verify that they're the same.
   (`$ openssl dgst -md5 file1 file2`)
2. What are their SHA-256 hashes? Verify that they're different.
   (`$ openssl dgst -sha256 file1 file2`)

This component is intended to introduce you to MD5 collisions; you will not submit anything for it.

2.3.1 Generating Collisions Yourself

In 2004, Wang's method took more than 5 hours to find a collision on a desktop PC. Since then, researchers have introduced vastly more efficient collision finding algorithms. You can compute your own MD5 collisions using a tool written by Marc Stevens that uses a more advanced technique. You can download the `fastcoll` tool here:

`http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5.exe.zip` (Windows executable) or
`http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5-1_source.zip(sourcecode)`

If you are building `fastcoll` from source, you can compile using this makefile: `https://subversion.ews.illinois.edu/svn/fa15-cs461/_shared/mp1/Makefile`. You will also need the Boost libraries. On Ubuntu, you can install these using `apt-get install libboost-all-dev`. On OS X, you can install Boost via the Homebrew package manager using `brew install boost`.

1. Generate your own collision with this tool. How long did it take?
   (`$ time ./fastcoll -o file1 file2`)
2. What are your files? To get a hex dump, run `$ xxd -p file`.
3. What are their MD5 hashes? Verify that they're the same.
4. What are their SHA-256 hashes? Verify that they're different.

This component is intended to introduce you to MD5 collisions; you will not submit anything for it.

2.3.2 A Hash Collision Attack

The collision attack lets us generate two messages with the same MD5 hash and any chosen (identical) prefix. Due to MD5's length-extension behavior, we can append any suffix to both messages and know that the longer messages will also collide. This lets us construct files that differ only in a binary "blob" in the middle and have the same MD5 hash, i.e. $prefix \parallel blob_A \parallel suffix$ and $prefix \parallel blob_B \parallel suffix$.

We can leverage this to create two programs that have identical MD5 hashes but wildly different behaviors. We'll use Python, but almost any language would do. Copy and paste the following three lines into a file called `prefix`: (Note: writing below lines yourself may lead to encoding mismatch and the error may occur while running the resulting python code)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
blob = """
```

and put these three lines into a file called `suffix`:

```
"""
from hashlib import sha256
print sha256(blob).hexdigest()
```

Now use `fastcoll` to generate two files with the same MD5 hash that both begin with `prefix`. (`$ fastcoll -p prefix -o col1 col2`). Then append the suffix to both (`$ cat col1 suffix > file1.py; cat col2 suffix > file2.py`). Verify that `file1.py` and `file2.py` have the same MD5 hash but generate different output.

Extend this technique to produce another pair of programs, `good` and `evil`, that also share the same MD5 hash. One program should execute a benign payload: `print "I come in peace."` The second should execute a pretend malicious payload: `print "Prepare to be destroyed!"`. Note that we may rename these program before grading them.

What to submit Two Python 2.x scripts named `sol_2.3_good.py` and `sol_2.3_evil.py` that have the same MD5 hash, have different SHA-256 hashes, and print the specified messages.

2.4 Guess the number (20 points)

The goal of this task is to guess a number that we will announce in class on Friday, September 18.

Note: You MUST complete this section and submit `sol_2.4_hash.hex` to svn **before 3.30PM on Friday, September 18**.

Here are the rules:

1. At the start of class on September 18, Professor Bailey will announce the winning number, which will be an integer in the range $[0, 63]$.

2. Prior to the announcement, each student may register one guess. To keep everything fair, your guesses will be kept secret using the following procedure:

   (a) You'll create a PostScript file that, when printed, produces a single page that contains only the statement: "*your_netid* `guesses` *n*".

   (b) You'll register your guess by submitting MD5 hash of your file to `sol_2.4_hash.hex`. **We must receive your hash value before the professor announces the pick**.

   (c) You and your project partner may collaborate and produce a file that includes both of your netids, or you may choose to work independently.

3. If your guess is correct, you can claim the prize by submitting your PostScript file to `sol_2.4.ps`. The professor will verify its MD5 hash, print it to a PostScript printer, and check that the statement is correct.

Hint: You're allowed to trick us if it will teach us not to use a Turing-complete printer control language with a hash function that has weak collision resistance.

Checkpoint 2: Submission Checklist

Inside your mp1 directory svn, you will have the auto-generated files named as below. Make sure that your answers for all tasks up to this point are submitted in the following files before **Friday, September 18** at **6:00pm**:

SVN Directory

`https://subversion.ews.illinois.edu/svn/fa15-cs461/NETID/mp1`

File Format

All submitted files should be submitted as plantext. The autograder runs on Unix, so students are encourage to validate in EWS that the files can be read as you intended. Any lines in your submission that begins with '#' will be ignored.

example content of .txt solution file

```
SPN WMKTQIW QR SPBW HQGRSEMW HQVS QY VEKW ENAENWNRS VEVZQR
```

example content of .hex solution file

```
3dab821d92b5ca7f48beee066996b8abc82f7e5646a0561710ea5bc11c80d
```

Team Members

`partners.txt` : a text file containing netIDs of both members, one netid per line. Place the student's netID, whose directory contain your project submission, at the top of the file.

example content of partners.txt

```
netid1
netid2
```

List of solution files that must be submitted for checkpoint 2

- `partners.txt`
- `len_ext_attack.py`
- `sol_2.1.2.txt`
- `sol_2.2.hex`
- `sol_2.3_good.py`
- `sol_2.3_evil.py`
- `sol_2.4_hash.hex` (MUST be submitted by **3:30PM** on Friday, September 18)
- `sol_2.4.ps`