CS 426: Computer Security
Fall 2023
Homework 9: Cryptography

Due Dec 2nd, 2025 @ 11:59 PM

1 Overview

The goal of this assignment is to help you gain hands-on experience on real-life cryptographic problems.

1.1 Environment Setup and Logistics

This assignment can be done in any environment of your choice. We recommend Linux, but it is totally up to you.

In this assignment, you will be 'cracking' the cryptographic protocols/keys provided to you and released in public (e.g., as a research paper) for academic and research purposes. Attempting the same kinds of attacks against others' systems without authorization is prohibited by law and university policies and may result in *fines, expulsion,* and *jail time*. Do not attempt to crack other entities' passwords without their permission.

Moreover, do not attempt brute force any problems of this assignment, especially on the server provided for Section 1.6. This will only make the server unusable for everyone and you will not receive any credit for it.

1.2 Setup

Files for the rest of the challenges are provided in Brightspace as `hw9_crypto.tar.xz`. This tarball consists of the following four subfolders:

- `dictionary`: for Problem 1 (Section 1.3).

  `rockyou.txt`, `passwd.txt`, and `shadow.txt`.

- `length`: for Problem 2 (Section 1.4).

  `pymd5.py`, `intro_to_pymd5.py`, and `length_extension.py`.

- `md5`: for Problem 3 (Section 1.5)

  `fastcoll_Makefile.mk`, `prefix`, and `suffix`.

- `rsa`: for Problem 3 (Section 1.6)

  `rsa_chal_server.py`

Do not modify any of these files unless the instruction tells you to do so.

1.3 Problem 1: Dictionary Attack (10 points)

You are given two text files, `shadow.txt` and `passwd.txt`, that are copies of the files `/etc/shadow` and `/etc/passwd` of a running Ubuntu system respectively. In a typical Linux-based OS, the user account information is saved in the `/etc/passwd` file, and the hash values of their passwords are saved in the `/etc/shadow` file. As we learned in the lecture, finding a preimage of a hash function is difficult and computationally expensive (except for ones like MD5, which we will discuss later).

However, just as everyone has their own favorite vocabulary that they like to say and use in real life, everyone also has their own favorite choice of passwords. But apparently, the 'pool' of passwords everyone chooses from is not as diverse and it leads to some unfortunate situations such as the one that this news article describes. There are also many (and massive!) lists of commonly used passwords, one of the most popular ones being `rockyou.txt` from the company name RockYou (if you would like to know about its history, see this Wikipedia page). There is even a free password cracking tool called John the Ripper and it works well with `rockyou.txt`!

Task 1. For this problem, you will be cracking passwords of five users of a Linux system: `milad`, `qi`, `kazem`, `cs426`, and `root`. You will need to crack all five passwords to receive full credit for this problem. Here are the suggested steps:

1. Download John the Ripper. If you are using Linux, run `sudo apt-get install john` on a terminal. On macOS, run `brew install john`. We've also installed John on the armor0 server in HW6 (`armor0.cs.purdue.edu`) for your convenience.
2. Read the John the Ripper documentation and crack the five hashed passwords.
   - Specify the hash type explicitly using the `--format` option since John may not always detect the correct format automatically.
   - Start with the simple cracking modes, such as the single crack mode and the wordlist mode, before attempting other more complex and time-consuming modes. On most machines, it takes less than 1 minute to crack all passwords if you use correct cracking modes.
   - All cracked passwords are saved in `~/.john/john.pot`, which you can view using the `--show` option. Delete this file if you want John to re-crack all passwords from scratch.

Save your cracked passwords in a file named `passwords.csv` using the following format:

```
username,password
root,<cracked_password>
cs426,<cracked_password>
milad,<cracked_password>
qi,<cracked_password>
kazem,<cracked_password>
```

Your writeup should include a brief description of your process such as the scripts and commands you used. Include a screenshot as a proof. Suggest a fix for this vulnerability.

Note 1. This assignment could be computationally challenging for some machines. You must NOT use the Purdue data servers directly for running the password cracker if you need more computation powers. You should submit your jobs to Jacobi machines via `srun` from `queue.cs.purdue.edu`.

In order to run your jobs on Jacobi machines, you should first ssh into `queue.cs.purdue.edu`:

```
ssh <purdue_username>@queue.cs.purdue.edu
```

Afterwards you need to create a bash script that contains your commands to run the password cracker. Make sure your script is executable by running:

```
chmod +x ./your_script.sh
```

Then you can submit your job on Jacobi machines by running:

```
srun -A jacobi --nodes=1 --ntasks=1 --cpus-per-task=1 ./your_script.sh
```

However, similar to what was said in Section 1.1, DO NOT abuse any of them, for example by running multiple John's at once.

1.4 Problem 2: Length Extension Attack (10 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. Many common hash functions use a design called the Merkle-Damgard 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-size 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 $b_{n+1}, b_{n+2}, \cdots$ that we want to add. This process is called length extension, and it can be used to attack many applications of hash functions.

1.4.1 Introduction to `pymd5` and Length Extension Attack

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 are given the `pymd5` module as `pymd5.py` already and you can learn how to use it by running `pydoc pymd5`. To follow along with these examples, run Python in interactive mode and run the command from `pymd5 import md5, padding`.

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

```
from pymd5 import md5, padding
m = "Use HMAC, not hashes"
h = md5()
h.update(m)
print(h.hexdigest())
```

and confirm that the output is `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 the 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:

```
h1 = md5(state=bytes.fromhex("3ecc68efa1871751ea9b0b1a5b25004d"), count=bits)
```

See the `pymd5` module for more technical details. Now you can use length extension to find the hash of a longer string that appends the suffix "Good advice". Simply run:

```
suffix = "Good advice"
h1.update(suffix)
print(h1.hexdigest())
```

to execute the compression function over `suffix` and output the resulting hash. Verify that it equals the MD5 hash of

```
m + padding(len(m)*8) + suffix
```

which can be done as follows:

```
h2 = md5()
pad = padding(len(m)*8)
h2.update(m)
h2.update(pad)
h2.update(suffix)
print(h2.hexdigest())
```

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 part of the assignment 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.

1.4.2 Challenge

Length extension attacks can cause serious vulnerabilities when people mistakenly try to construct something like an HMAC by using `hash(secret || message)`. The National Bank of CS 426, which is not up-to-date on its security practices, hosts an API that allows its client-side applications to perform actions on behalf of a user by loading URLs of the form:

```
http://bank.cs426.purdue.edu/hw5/api?token=e99e333b393af0f3827c6f7497c8fd55&user=
kazem&command1=ListSquirrels&command2=NoOp
```

(for accessing this link, see Note 2 below) where token is

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

Using the techniques that you learned in the previous section and without guessing the password, apply length extension to create a URL ending with `&command3=UnlockAllSafes` that would be treated as valid by the server API.

Task 2. For this problem, you will be writing a Python 3 script named `length_extension.py` that does the following:

1. Accepts a valid URL in the same form as the one above as a command line argument.
2. Modifies the URL so that it will execute the `UnlockAllSafes` command as the user.
3. Prints the new URL to the command line.

Your write-up should include your code with a sufficient amount of comments (anyone should be able to understand what you are doing just by reading your code). Suggest a fix to this vulnerability. Proof of success for this problem is not necessary as we can see your progress on Gradescope.

Note 2. Because of its bad security practices, the National Bank of CS 426 has taken down its website. So you will be using our course Gradescope (**length_extension**) to test your code attack URL.

Hint 1. For testing: The input URL will have the same form as the sample above, but we may change the server hostname and the values of `token`, `user`, `command1`, and `command2`. These values may be of substantially different lengths than in the sample. The input URL may be for a user with a different password, but the length of the password will be unchanged.

Hint 2. You might want to use the `quote()` function from Python's `urllib.parse` module to encode non-ASCII characters in the URL. Also, consider using `bytes.fromhex()` when passing in the state to `md5()`.

1.5 Problem 3: MD5 Collisions (10 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.

1.5.1 Generate Collisions Yourself!

The first known collisions were announced in 2004 by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Here is 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
```

Copy the above hex strings into `file1.hex` and `file2.hex`. Convert each group of hex strings into a binary file: on Linux, run `xxd -r -p file.hex > file`. You can verify that those two messages indeed return the came MD5 hash values by running `openssl dgst -md5 file1 file2`

Wang et al. in 2004 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, called `fastcoll`, written by Marc Stevens that uses a more advanced technique.

We have already built `fastcoll` for you on the course server `armor0.cs.purdue.edu`.

You may also download `fastcoll` from the official HashClash website:

- Windows executable: https://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5.exe.zip
- Source code: https://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5-1_source.zip

If you choose to compile `fastcoll` from source, you can use the provided makefile (`fastcoll_Makefile`). You must also have the Boost development libraries installed:

- Ubuntu/Debian: `sudo apt-get install libboost-all-dev`
- macOS (Homebrew): `brew install boost`

Task 3. Answer the following questions on your write-up.

Q1. Generate your own collision with this tool. How long did it take? (You can use this command: `time ./fastcoll -o file1 file2`.)

Q2. What are your files? Copy and paste the content of the hex dump files. Are they the same or different? (You can use this command: `xxd -p file`)

Q3. What are their MD5 hashes? Are they the same or different?

Q4. What are their SHA-256 hashes? Are they the same or different? What does this tell us about MD5 (just one sentence-long explanation is sufficient)?

1.5.2 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 || blobA || suffix`" and "`prefix || blobB || suffix`".

We can leverage this to create two programs (shell scripts) that have identical MD5 hashes but wildly different behaviors. We're using shell scripts, but this could be done using a program in almost any language. The file name `prefix` contains the following:

```
#!/bin/bash
cat << "EOF" | openssl dgst -sha256 > DIGEST
```

and the file name `suffix` contains the following (make sure it starts with a blank line):

```
EOF
digest=$(cat DIGEST | sed 's/(stdin)= //' )
echo "The sha256 digest is $digest"
```

Now use `fastcoll` to generate two files with the same MD5 hash that both begin with `prefix`. This can be done via:

```
fastcoll -p prefix -o col1 col2
```

Then append the suffix to both, which can be done with the command:

```
cat col1 suffix > file1.sh; cat col2 suffix > file2.sh
```

You should be able to see that `file1.sh` and `file2.sh` have the same MD5 hash but generate different output.

Task 4. Extend this technique to produce another pair of programs, `good.py` and `evil.py`, that also share the same MD5 hash.

- `good.py` should print: `I mean no harm.`
- `evil.py` should print: `You are doomed!`

We will test your programs using **Python 3.9**, which is backward compatible with most Python 3.x code. Moreover, since `fastcoll` generates arbitrary binary data, which includes NUL bytes, you may need to specify a single-byte encoding like `latin-1` (tutorial) and use Python versions no newer than Python 3.9. You are restricted to using the standard Python library (e.g., `hashlib`).

Your writeup should describe (in detail) the steps you took to produce `good.py` and `evil.py`. Attach screenshots of your `good.py` and `evil.py` as well as the proof of success.

1.6 Problem 4: 'Leaky' RSA (10 points)

A team of penetration testers was tasked to steal a piece of sensitive information (hereafter referred to as the 'flag') from the armor server `armor3.cs.purdue.edu`. After a series of reconnaissance and scanning, they discovered that port number `8080` is open. Upon connection (`nc armor3.cs.purdue.edu 8080`), they were greeted by an interactive program that reads the text file that contains the flag. However, the program is (obviously) designed to not give out the plaintext of the flag; instead, it encrypts the flag differently at every iteration and returns the encrypted flag instead. The program is also designed in a way that it gives you some information about `p` and `q` (whatever those are supposed to mean). See Figure 1.

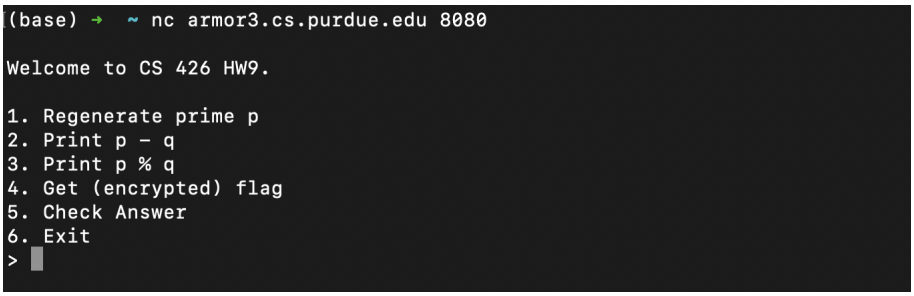

Figure 1: `nc armor3.cs.purdue.edu 8080`

After more penetration testings, the team has managed to extract some portion of the code that the server is running: `rsa_chal_server.py`, but they still got yelled at by their boss because they failed to extract the actual plaintext of the flag. However, from the code `rsa_chal_server.py`, they discovered that the flag is possibly encrypted using RSA, and the `p` and `q` mentioned above were the prime numbers that the server is using to encrypt the flag. They have recently heard that you are taking CS 426 at Purdue. Knowing that Purdue is a great school and CS 426 is a great course, you must be a great-great security expert, they have requested that you help them decrypt the flags for them.

Task 5. Your task is to recover the plaintext of the flag by interacting with this server (that you can access by entering `nc armor3.cs.purdue.edu 8080` on your terminal). Save the obtained plaintext in `leaky_rsa_flag.txt`, and submit to Gradescope. Your writeup should include the steps you have taken to find the flag in detail, and indeed the flag itself. If you are making any mathematical statement, you will also need to leave a proof of your statement. Suggest a fix to this vulnerability. Does this code suggest anything regarding the security of RSA (such as, how it should be implemented) or even other cryptographic protocols? Attach any scripts you used and screenshots as necessary.

Note 3. If you cannot connect to the server via `nc armor3.cs.purdue.edu 8080`, make a private post on the course EdStem page ASAP. Please make sure you are using Purdue VPN for connection.

2 Deliverable

Your write should be a single PDF file named `report.pdf`. It should be submitted to Gradescope under `Homework 9 (Report)`. As mentioned multiple times already, the write-up should contain sufficient instructions and details to prove the **integrity, validity, and reproducibility** of your work.

Also, submit all cracked secrets and codes to **Homework 9 (Code)** on Gradescope for autograding.

Refer to Table 1 for a complete list of required submission files.

File	Description	Submission
`report.pdf`	Your report/write-up as described in Tasks 1 to 5.	**pdf** on Gradescope [Homework 9 (Report)]
`passwords.csv`	Passwords cracked in Task 1.	All remaining files should be submitted to Gradescope under [Homework 9 (code)]
`length_extension.py`	Solution script to Task 2.	
`good.py`	Code generated in Task 4.	
`evil.py`	Code generated in Task 4.	
`leaky_rsa_flag.txt`	Flag obtained in Task 5.	
`other files`	Any additional files or programs created to support your efforts.	

Table 1: List of required submission files

3 Grading and Policy

3.1 Grade Breakdown

Each of the four challenges in this homework is worth 10 points, with 5 points allocated to the secrets/programs, and 5 points allocated to the write-up.

In total, this homework is worth 4% of your final grade.

3.2 Important Rules and Reminders

You are more than welcome to use any external resources, including the ones available online besides lecture slides. However, the academic integrity policy still holds. **Do not ask someone else to do your work for you** or copy and paste answers. **Use your resources reasonably, and do not cross the line**. See the **Academic Integrity** section of the syllabus (course webpage) for more details.