

Assignment #2

Testing Access Control and Passwords

Security Measurement and Testing
Christopher Wood
12/18/12

Table of Contents

[Introduction](#)

[Exercise 1](#)

[Exercise 2](#)

[Password: bad \(7.8 bits\)](#)

[Password: information \(38.2 bits\)](#)

[Password: tgyh \(13.8 bits\)](#)

[Password: qzwxecrvtbynum \(58.2 bits\)](#)

[Password: w1n \(9.1 bits\)](#)

[Password: a\\$t3L1n# \(32.3 bits\)](#)

[Exercise 3](#)

[Discussion and Results](#)

[Exercise 1 Discussion](#)

[Exercise 2 Results](#)

[Exercise 3 Results](#)

[Exercise 3 Code](#)

Introduction

This report describes my work done to audit and crack passwords used to enforce access control to Windows and UNIX operating systems. I used the **Windows Password and Registry Editor** and **John the Ripper** tools to conduct the first two parts of this assignment. For the third part of this assignment I wrote a password cracking program that provides dictionary and brute force attack modes to attempt to break passwords. This report is partitioned into the experimental procedures and results for each part of the assignment.

Exercise 1

In this part of the assignment I followed the instructions given to break into Windows without knowing the password. Before getting started, I created a new account **caw** on the test machine (which was run as a virtual machine inside **VirtualBox**) with the following password:

qzwxecrvtbynum

With this account established I was required to enter the password to log into Windows, as expected. Then, I proceeded to download, burn, and boot the **Windows Password and Register Editor** tool. The tool automatically booted from the CD-ROM and then prompted me with the following questions and instructions:

1. Please select the partition number.
 - a. Windows 7 automatically partitions the hard-drive into the operating system and boot partitions, so I chose the operating system partition as this is where the passwords and registry information would be stored.
2. What is the path to the registry directory?
 - a. Since I did not do a custom Windows installation, the default registry directory of C:\Windows\System32\ was valid, so I entered this option.
3. Select which part of the registry to load, use predefined choices or list the files with space as delimiter.
 - a. For this question I chose the first option (1: Password reset)
4. What to do?
 - a. The only option available was 1: Edit user data and passwords, so I chose this option
5. Enter the username to change
 - a. I entered the 'caw' username as the target for this tool.
6. Manipulate the user information from the "User Edit Menu"
 - a. From this menu I chose 1: Clear (blank) user password, and the tool responded back, "Password cleared!"

After completing these steps I saved the modifications to the SAM file, closed out of the tool, and rebooted the virtual machine. Upon arriving at the Windows login screen I clicked on the caw account and was automatically logged in without being asked to enter a password. From this point I was able to change the password to any value I wanted within Windows. The tool worked as expected.

Now, to change the password from the tool (instead of making it blank to change it within Windows) I performed the same procedure, with the exception that step 6 was modified in order to change the password. Instead of clearing the password, I reset it to:

welcome

Upon completion, I saved the results to the SAM file, closed out of the tool, restarted the virtual machine, and then tried to enter the new password. However, the new password did not work (I repeated the process three times to verify it wasn't an error on my part). This is likely a problem with the tool, and the author's website mentions that there are some issues with modern versions of the Windows NT operating system (i.e. Windows 7). Attempting to change the password to a blank (i.e. just pressing the return key) does not work because the tool automatically assumes that no changes should be made. Therefore, I was only successful in clearing the password to then log in and change it to the desired value.

The only tools I used to complete this exercise were VirtualBox (running a Windows 7 VM) for virtualization and the Windows Password and Registry Editor tool provided in the assignment description. I did not need to gather any information to launch this attack since it was almost entirely automated with the exception of entering the target parameters.

As I previously mentioned, the only problem I encountered was trying to change the password from the provided tool. To circumvent this problem I simply cleared the password from the tool and then logged in to Windows to change it manually. Also, since all of this work was specific to Windows it does not carry over into the remaining parts of the assignment. As I discuss in the following sections, I used Ubuntu (a Linux distribution) to conduct my experiments, and this OS which handles passwords completely different from Windows.

The most important learning outcome from this part of the assignment was that, regardless of how complex a given password may be, there usually exists side-channel attacks that are capable of circumvent this access control mechanism to enable access to a protected system. This is true for every implementation of password-based access control systems, as we will see in the following sections.

Exercise 2

In this password audit exercise, I worked with John the Ripper, a cross-platform and open source password cracking tool, to test a variety of passwords with different levels of strength. As per the assignment description, we were asked to select six different passwords to test (or crack). The passwords I selected are listed below along with their measures of entropy, which is calculated using the equation $H = \log_2 N^L$, where H is the entropy, L is the length of the password, and N is the set of symbols used to create the password.

1. bad - 7.8 bits
2. information - 38.2 bits
3. tgyh - 13.8 bits
4. qzwxecrvtbynum - 58.2 bits
5. w1n - 9.1 bits
6. a\$t3L1n# - 32.3 bits

As is the case in all modern operating systems, passwords are not stored as plaintext. Rather, their hash digests are stored in a privileged file. For example, on UNIX machines, the password digests are stored in `/etc/shadow`, which is only accessible to users with root access. To avoid working with this file and running my machine with root privileges, I chose to make my own password files for each of the aforementioned passwords using the program **mkpasswd**. This program is the standard UNIX password generation tool that supports a variety of hash functions, including MD5, SHA-1, and SHA-2, and also salts every password to prevent the effectiveness of dictionary and rainbow table attacks. To create a password file with this program I ran the following in the terminal:

```
$> mkpasswd --method=crypt > passwd.db
```

Note that I used the **crypt** format, which on Ubuntu (the operating system upon which Linux Backtrack is based) is implemented as DES in which the encryption key is the candidate password and the salt is a two-character string chosen from [a-zA-Z0-9./]. At the prompt, I entered each of the aforementioned passwords and then saved their digests in an encrypted password file in the following format (encrypted with the MD5 hash algorithm):

```
username:obFEVLNmmASEw # the password hash digest
```

Also, to avoid downloading and compiling the source code for John the Ripper myself to facilitate the password cracking process, I used Linux BackTrack, a Linux distribution configured for penetration testers. The mkpasswd program did not come with the distribution, so I had to install it manually (sudo apt-get install mkpasswd).

The last thing to do prior to working on these password files was to write a script to time the password cracking attempts. The Perl code for this task is shown below (where the one command line argument is the path to the password file).

```
#!/usr/bin/perl -w

use Time::HiRes qw/ time sleep /;

my $start = time;
system("john --format=crypt $ARGV[0]");
my $end = time;
my $run_time = $end - $start;
print "Elapsed time: $run_time\n";
```

Notice that in this script I am specifying the crypt format, which is the same format used to generate the password digests with the mkpasswd program. Now, with the encrypted password files, John the Ripper inside Linux BackTrack, and the timing script, I set out to crack each of the passwords. My process for each password is described in the following sections.

Password: bad (7.8 bits)

Due to the simplicity of this word, John the Ripper was able to crack it in less than a second. I did not encounter any problems with this password.

Password: information (38.2 bits)

This dictionary word was not included in the default John the Ripper word list. Therefore, conducting a brute force attack was unsuccessful, and the modifying the mangling rules would not help. Due to the entropy of the password I immediately ruled out a brute force attack as it would take too much time. My solution was to replace the standard wordlist that is distributed with the John the Ripper source code with one commonly used by Cain and Abel, located at <http://www.skullsecurity.org/wiki/index.php/Passwords>. This password file has approximately

ten times as many words as the default dictionary for John the Ripper, making it much more comprehensive. After replacing the default word list with this new dictionary, I was able to successfully crack the password in a matter of seconds.

Password: tgyh (13.8 bits)

Since this is not a word in any common dictionary I decided to use John the Ripper's incremental mode (with the Alpha parameter to only use all 26 letters in the alphabet) to crack this password. Based on a compiler-defined option, only passwords of at most length 8 can be fully matched using this mode of operation. From my experiments, it appears as though the incremental algorithm is parameterized by this upper limit in such a way that it focuses on mainly passwords of this length (i.e. it does not perform an exhaustive check of all passwords of length 4 before going to longer passwords if the upper limit is 8). This hypothesis was verified when I ran the default incremental attack mode for more than ten minutes without any success. After this time passed I stopped John the Ripper, changed the upper limit on the passwords for incremental mode in the **john.conf** file (MaxLength = 4), and restarted the program. With this change I was able to crack the password in approximately 15 seconds. The command-line options for this mode of execution are as follows:

```
john --format=crypt --incremental=Alpha /root/Desktop/crack.db
```

Password: qzwxecrvtbynum (58.2 bits)

Out of all my selected passwords, this one has the most entropy and is certainly not anywhere close to a word in the dictionary. Therefore, the brute force attack mode was the only appropriate technique to crack this password. Also, since it only consisted of letters, I used the 'Alpha' parameter again to limit my brute force search by the 26 characters in the alphabet, and ran John the Ripper as follows (crack.db is the file containing the encrypted password):

```
john --format=crypt --incremental=Alpha /root/Desktop/crack.db
```

Unfortunately, based on the compiler-defined limit of 8 characters for brute force mode, this password could not possibly be fully matched. However, there are instances where John the Ripper can match part of a password (as per the online documentation), especially with the DES-based crypt hash function, and in an effort to find such a match I let John the Ripper run in its Alpha incremental mode for 24 hours. However, it was not successful within this time frame, and I was not able to crack this password.

Password: w1n (9.1 bits)

This is the first password that uses alphanumeric symbols in the character set. Therefore, I tried the same approach from the past two passwords attempts, with the exception of this time using the 'All' incremental mode, which includes letters, numbers, and special characters in the character set. Even after bumping down the maximum password length from 8 to 4 symbols, as I did in the third password experiment, John the Ripper was still unable to crack this password

within the timeframe of 20 minutes, which I felt to be too long for such a short and simple password. Therefore, since this password is close to the dictionary word *win*, I decided to add a mangling rule to the **john.conf** file that replaces all 'i' characters with '1', as follows:

```
[List.Rules:WordList]

# Replace all 'i' character instances with '1'
si1
```

With this rule, I ran John the Ripper in its default dictionary mode as follows:

```
john --format=crypt /root/Desktop/crack.db
```

With this new substitution rule I was able to crack the the password in approximately 5 seconds. Because this technique was effective, I kept this mangling rule for the next password experiment.

As a second attempt to crack I then tried the 'Alnum' incremental mode, as this includes all alphanumeric characters in the character set. I limited the maximum word length to 4 characters and ran John the Ripper as follows:

```
john --format=crypt --incremental=Alnum /root/Desktop/crack.db
```

With these parameters, John the Ripper was able to crack this password in less than 10 minutes.

Password: a\$t3L1n# (32.3 bits)

Based on the previous password, I extended my mangling rules to include the following:

```
[List.Rules:WordList]

# Perform common character substitutions
si1
se3
ss$
ss5
st7
```

I then tried the dictionary attack mode even though this password did not resemble any word in a dictionary. Unfortunately, as expected, this did not work, as I was forced to terminate the program after 6 hours passed with no success.

Therefore, 'All' incremental mode was the only remaining alternative, so I tried this approach to finish the experiment. Unfortunately, I was not able to crack the password, and was forced to terminate the execution of John the Ripper after 14 hours elapsed.

Exercise 3

From the previous exercise, dictionary and brute force seemed to be the most successful (and only useful options) provided by John the Ripper for this assignment, so I decided to replicate these attacks in my code. However, unlike John the Ripper, I do not account for salted passwords, and my program, Wedge (available at <https://github.com/chris-wood/Wedge>), generates its own password files at runtime. By making the hash function a format passed into the program from the command-line I could easily test and verify the correctness of my program.

As for the details of Wedge, it is a cross-platform password cracking tool written in Python. It is configurable in that it allows the users to specify the type of hash function to use for the password cracking, as well as the word list to use in the dictionary attack. It uses the **hashlib**, **crypt**, and **threading** Python libraries to provide the necessary cryptographic routines and threading support. Furthermore, it comes packaged with the Cain and Abel password cracking dictionary. No other technologies were used in the making of this program.

Unfortunately, as I have said, it does not support salted passwords (due to a lack of time to implement the program). Given more time I would have explored:

1. Rainbow table attacks (precomputing hash digests)
2. Multi-threaded brute force attacks that equally partition the candidate passwords among the number of available cores using an executor service to manage the threads (similar to Java's `ExecutorService` class).
3. Salted passwords to match real-world password generation and storage techniques.
 - a. There are many tutorials available online that explain the proper password salting technique, as well as those describing how to crack salted passwords. Here is one such tutorial: http://www.exploit-db.com/wp-content/themes/exploit/docs/Cracking_Salted_Hashes.pdf

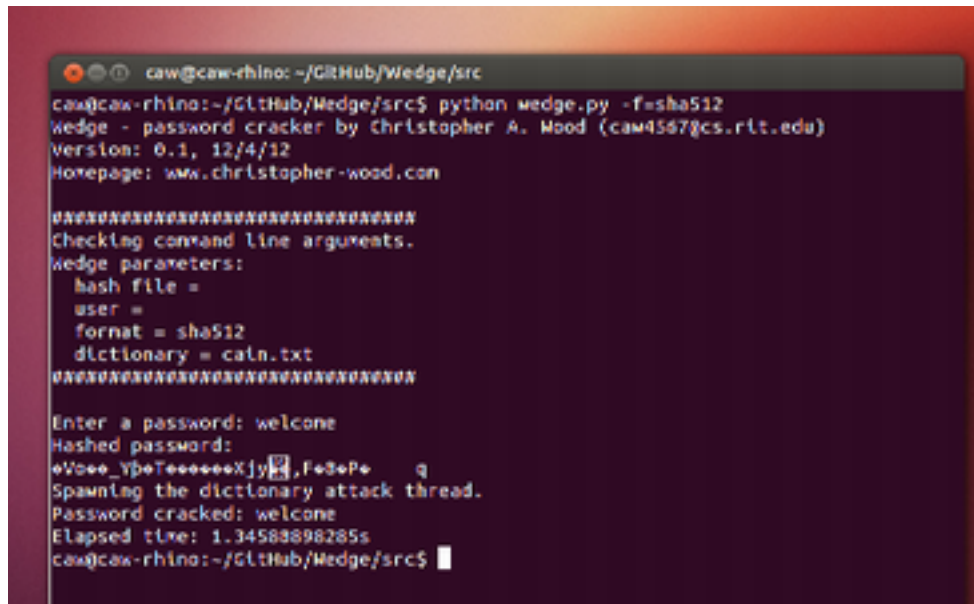
The basic control flow of Wedge is as follows:

1. Prompt the user to enter a password to encrypt (hash).
2. Encrypt the entered password using the specified hash function.
3. Launch a dictionary attack that attempts to crack this password using the specific hash function.
4. If the dictionary attack fails, launch a brute force attack that attempts to crack this password using the specified hash function. Depending on the pre-defined time limit for this brute force attack, Wedge may terminate unsuccessfully if it does not find a matching password in the allotted time or if it successfully finds a match.

5. Display the attack results and elapsed time.

Based on this main control flow, Wedge is capable of cracking unsalted passwords for any of the specified hash functions.

A screenshot showing its execution is below.



```
caw@caw-rhino: ~/GitHub/Wedge/src
caw@caw-rhino:~/GitHub/Wedge/src$ python wedge.py -f=sha512
Wedge - password cracker by Christopher A. Wood (caw4567@cs.rlt.edu)
Version: 0.1, 12/4/12
Homepage: www.christopher-wood.com

#####
Checking command line arguments.
Wedge parameters:
  hash file =
  user =
  format = sha512
  dictionary = caln.txt
#####

Enter a password: welcome
Hashed password:
eV0ee_Yp0TeeeeeXjy[REDACTED],Fe8ePe  q
Spawning the dictionary attack thread.
Password cracked: welcome
Elapsed time: 1.3458898285s
caw@caw-rhino:~/GitHub/Wedge/src$
```

Figure 1: Sample execution of Wedge.

The only information gathered by Wedge is the solved password (if the attacks were successful) and the elapsed time for the attack. Also, the program itself is very modular, allowing for extra attack modes to be easily implemented and plugged in for testing purposes.

Discussion and Results

This section describes my results from the three exercises in this assignment.

Exercise 1 Discussion

I was able to very easily clear the password for the Windows account I created. Although the password for this account was encrypted in the SAM password file stored in the Windows registry directory (Windows\System32\config), the Windows Password and Registry Editor tool was able to delete the password in its entirety from the SAM file. Thus, I had the freedom of either selecting a new password of my own choice or keeping it null from within Windows.

Unfortunately, the tool was not able to change the password to something new that was not null (blank). Perhaps this is an indication that the SAM file for Windows 7 (and newer) operating

systems uses extra precautions when storing passwords. This is a good step forward for Windows users.

Exercise 2 Results

The results from part 2 of the assignment (John the Ripper) are shown in the table below. Note that all of the passwords were encrypted using the default **crypt** hash routine because this is the default for modern *NIX operating systems.

Table 1: John the Ripper password cracking results with crypt hash formats.

Original Password	Digest Format	Successful?	Time Duration
bad	crypt	Yes	0.548s
information	crypt	Yes	1.199s
tygh	crypt	Yes	16.798s
qzwxcvrtbynum	crypt	No	24 hour time limit exceeded
w1n	crypt	Yes	4.835s
a\$t3L1n#	crypt	No	14 hour time limit exceeded

Exercise 3 Results

Due to the fact that I was cracking unsalted versions of my passwords, I could not test with the **crypt** hash mode. Therefore, for completeness, I tested my passwords with both the MD5 and SHA-512 hash functions. The results from these experiments conducted with Wedge are shown in the tables below.

Table 2: Wedge password cracking results with MD5 hash formats.

Original Password	Digest Format	Successful?	Time Duration	Attack mode
bad	MD5	Yes	0.0147s	Dictionary
information	MD5	Yes	0.434s	Dictionary
tygh	MD5	Yes	2.466s	Brute force (Alpha mode)
qzwxecrvtbynum	MD5	No	24 hour time limit exceeded	Brute force (Alpha mode)
w1n	MD5	Yes	1.08s	Brute force (Alnum mode)
a\$t3L1n#	MD5	No	24 hour time limit exceeded	Brute force (All mode)

Table 3: Wedge password cracking results with SHA-512 hash formats.

Original Password	Digest Format	Successful?	Time Duration	Attack mode
bad	SHA-512	Yes	0.163s	Dictionary
information	SHA-512	Yes	0.446s	Dictionary
tygh	SHA-512	Yes	2.022s	Brute force (Alpha mode)
qzwxecrvtbynum	SHA-512	No	24 hour time limit exceeded	Brute force (Alpha mode)
w1n	SHA-512	Yes	0.946s	Brute force (Alnum mode)
a\$t3L1n#	SHA-512	No	24 hour time limit exceeded	Brute force (All mode)

Given the complexity of my fourth and sixth passwords, my program was not able to crack them in a reasonable amount of time. However, as John the Ripper was not able to crack them either, I feel that my program is fairly capable at cracking “normal” passwords. Also, generally speaking, my program (Wedge) is more efficient at cracking simple passwords. This is due to the complexity of John the Ripper and its internal rules engine. Wedge has no such component, and thus is able to check more candidate passwords in less time.

The screenshots below show the time limit being exceeded for the two failed passwords using the MD5 hash format.

```

caw@cam-rhino: ~/GitHub/Wedge/src
caw@cam-rhino:~/GitHub/Wedge/src$ python wedge.py -f=nd5 -l=alpha
Wedge - password cracker by Christopher A. Wood (cam4567@cs.rlt.edu)
Version: 0.1, 12/4/12
Homepage: www.christopher-wood.com

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Checking command line arguments.
Wedge parameters:
  hash file =
  user =
  format = nd5
  dictionary = cata.txt
  character set = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', '
x', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Enter a password: qzxecrvtbynun
Hashed password:
1f+ceYb+++F]de1k
Spawning the dictionary attack thread.
Spawning the brute force attack thread.
Brute force failed. We give up.
caw@cam-rhino:~/GitHub/Wedge/src$

```

Figure 2: Time limit exceeded for the first failed password.

```

caw@cam-rhino: ~/GitHub/Wedge/src
caw@cam-rhino:~/GitHub/Wedge/src$ python wedge.py -f=nd5 -l=alpha
Wedge - password cracker by Christopher A. Wood (cam4567@cs.rlt.edu)
Version: 0.1, 12/4/12
Homepage: www.christopher-wood.com

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Checking command line arguments.
Wedge parameters:
  hash file =
  user =
  format = nd5
  dictionary = cata.txt
  character set = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', '
x', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Enter a password: qztlttne
Hashed password:
+1P+eW1k4s/1k:
Spawning the dictionary attack thread.
Spawning the brute force attack thread.
Brute force failed. We give up.
caw@cam-rhino:~/GitHub/Wedge/src$

```

Figure 2: Time limit exceeded for the second failed password.

Exercise 3 Code

My source code is contained within the archive for this assignment submission.