Jordan Bayles
May 9, 2014

Homework 4
ECE 478: Network Security

0. Disclaimer

This submission reflects my own understanding of the homework and solutions. All of the ideas are my own, unless I explicitly acknowledge otherwise.

1. Cookie contents and format

The `hashhw` cookie has its value set to the username, creation timestamp, and a hex encoded MAC generated from the account name plus the timestamp ran through a md5 hash, truncated, and then reran concatenated with a key from a secret keyfile. Once a user enters the website while having a cookie, the cookie is checked by utilizing the same username, timestamp, and key to generate what the cookie value should be and then checking it to make sure the MAC is valid.

2. Account token

You completely control the username portion of the token, and can predict the timestamp that will be associated with it.

3. Token contents for ADMIN access

Must have the username set to "admin," and the MAC address must match the MAC generated by `admin.<timestamp>` using the key. The timestamp doesn't really matter in and of itself.

4. MD5trunc collision yields ADMIN? How much effort?

We know that the username must be admin, and we know that the mac must match admin plus some timestamp, and must use the secret key only known to the server. Thus, we must create a collision between `admin.<some timestamp>` and
`<some username != admin>.<some predicted or known timestamp>`. We can make it a birthday attack instead of a brute force attack by using a list of alternative usernames with known hashes, then calculating the MD5trunc of each username to create a list of possible collisions to check against.

A naive attack is 2^n in complexity, but if we use a birthday kind of attack instead of a second preimage, it makes the attack only $2^{n/2}$ in complexity.

It is important to note that we do NOT control the timestamp... it is a server side variable. Thus we have two options for finding collsions: use a known timestamp that occurs in the future (annoying, but what I chose to do), or pregenerate a ton of MACs with known

timestamps to make results less time dependent (great for me, not so great for not hammering the server...)

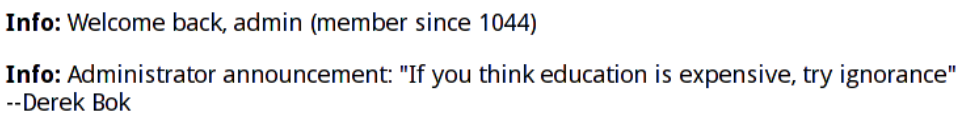
5. MD5trunc collision and approach

I found a collision using my python script, written to utilize the birthday problem. Using a timestamp of `090520140313` and a maximum length of 4 for both the username and admin timestamp, I found that the following values collide:

```
admin.1044          - 9bb5b59c8419244da574199300324198

fkkc.090520140313 - 9bb5b59c3433b0de8013e70d848ab799

Resulting MAC: cabe6ac811eb07a59e7cb30fd4597009
```



6. Mechanics of the final attack

The final attack used the collision above: at 03:13AM on May 9th, 2014, I created a user with the username `fkkc` on the server. This created a cookie with the MAC called "Resulting MAC" above, which I then modified to hold the value `admin.1044.cabe6ac811eb07a59e7cb30fd4597009`. A quick page refresh resulted in the admin message:

> "If you think education is expensive, try ignorance" –Derek Bok

7. Extension to different truncation lengths

This type of attack is very sensitive to the length of the truncation utilized, as it becomes increasingly difficult to match longer sections of the hash sum. For example, using timestamps and usernames of both length $[1, 4]$ results in many instances of no collisions for a truncation of length 5, whereas it has a very high probability of working for truncation of length 4. This set of usernames, timestamps is quite large, resulting in $26^4 * 10^4 = 4,569,760,000$ pairwise comparisons (using a linear traversal on a finite array). For a length 5 truncation, to have a reliable chance of success, we need to use timestamps and usernames of length 5 or more.

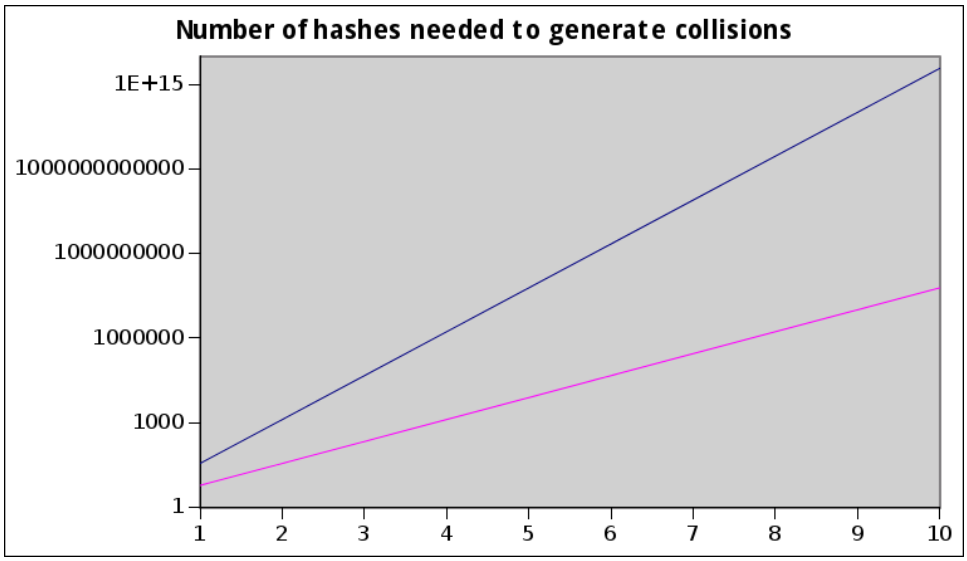


Figure 1: *Number of hashes needed to generate a collision*

Appendix

0.1 Collision finding script

collide.py

```
#!/usr/bin/env python

import hashlib
import sys

valid_chars = "abcdefghijklmnopqrstuvwyxz"
valid_ints = "0123456789"
truncate = 5 # characters

def generate_fields(element_list, max_len):
    li = list(element_list)
    index = [0,0]

    for _ in range(1, max_len):
        index = [index[1], len(li)]
        for string in li[index[0]:]:
            for character in element_list:
                li.append(string + character)

    return li

def generate_hashes(hhmm, max_len):

    # timestamp format DDMMYYYYHHmm
    user_timestamp = "09052014" + hhmm

    usernames = generate_fields(valid_chars, max_len)
```

```
    timestamps = generate_fields(valid_ints, max_len)

    # generate the hashes with timestamp=08052014hhmm
    user_hashes = [None]*len(usernames)
    for index, value in enumerate(usernames):
        data = value + "." + user_timestamp
        user_hashes[index] = hashlib.md5(data.encode('utf-8')).hexdigest()
            [:truncate * 2]

    # generate the hashes with username=admin
    admin_hashes = [None]*len(timestamps)
    for index, value in enumerate(timestamps):
        data = "admin." + value
        admin_hashes[index] = hashlib.md5(data.encode('utf-8')).hexdigest
            ()[:truncate * 2]

    return usernames, timestamps, user_hashes, admin_hashes

def find_collisions(hhmm, max_len):
    users, timestamps, usersh, adminsh = generate_hashes(hhmm, max_len)

    # run through and check for collisions
    for i, userh in enumerate(usersh):
        for j, adminh in enumerate(adminsh):
            if userh == adminh:
                print("Found collision")
                print(users[i], timestamps[j])
                return users[i], timestamps[j]

    return None

def main():
    print("MD5trunc clipping to", truncate)
    find_collisions(sys.argv[1], int(sys.argv[2]))
    return 0

if __name__ == "__main__":
    main()
```