CSE433S Final Project

Simulating a Timing Side Channel Attack

Shannon Coupland
Jason Oberstein

Github repo: https://github.com/shannon-coupland/cse433s.git within the final_project subfolder

Link to demo video:
https://wustl.box.com/s/81nuigigjh5x2uqzccmelhe8k4wqzz3w

Timing Attacks

Our area of focus is timing side channel attacks, specifically the following attack: when a server uses a byte-by-byte comparison function for checking credentials (i.e. a function that returns when it first encounters a mismatched character), a malicious client can recover a password for an arbitrary username one character at a time by timing server responses for different characters; when the server takes a suitably longer time to reply, the client knows it's hit upon a correct character and can move on to the next character. This is a much faster way to brute force a password than trying every possible password; while the latter would take exponential time, a timing attack of this kind will take linear time.

Problem Description

The server program we created has a data structure containing the usernames and passwords of two different users, "shan" and "jason." In our scenario, "shan" is the hacker -- she knows her own username and password, and she knows that the other user's username is "jason," but she does not know the other user's password or the contents of the server code. She has two C programs to complete her hack: *threshold.c*, which uses her own known username and password to repeatedly submit login attempts to the server and generate an average "threshold" value that describes how much the server response time increases for each correct password character; and *hack.c*, which is run with the result of *threshold.c* as a command line argument and yields the password associated with "jason."

The defense to this attack is twofold. The most direct way to counter it is to simply use a different kind of comparison function, one whose timing does not rely on the given passwords. Another way is to encrypt the passwords with a diffusive encryption scheme, such that a one-letter difference in the plaintext will not correspond to a one-letter difference in the ciphertext.

Development and Challenges

Development steps:

1. Created a basic socket-based server program to handle requests containing usernames and passwords.

   The server (*server.c*) and client programs are written in C and use INET socket-based communication. The server accepts only plaintext usernames and passwords (no encryption, to facilitate the attack), and it cannot multiplex connections. Once a client connects and disconnects, the server program quits. We could have changed this behavior by accepting multiple clients with a multiplexed connection strategy, but that would have introduced unnecessary complexity to what is ultimately a proof-of-concept project.

   After establishing a socket connection with the client, the server continually (in an endless while loop) reads a username from the client, reads a password from the client, then compares this password with the correct password (which it stores in a global data structure) and sends to the client the result of the comparison.

2. Created *client_test.c* to manually evaluate server behavior when queried with multiple passwords, and for use in debugging client and server behavior. **client_test.c is for our own reference during development and is NOT used in the final attack—please do not grade it.**

   This file is where we spent time developing most of our functions. In it, we used *timespec* structs to record the time from the MONOTONIC Linux clock before and after a *read()* call that reads the server's response to sending a password. We submitted passwords to the server with 0, 2, 4 . . . 28 correct characters of the password associated with "shan" to get a sense of timing, fine tune our querying methods, and guide our creation of *threshold.c*.

   Example output from client_test.c

```
[shannon.coupland@linuxlab007 final_project]$ ./client_test
connected
test1 (correct) average time (us): 1744.322000
test2 (incorrect, index 26 of 28) average time (us): 1570.266500
test3 (incorrect, index 24 of 28) average time (us): 1461.920380
test4 (incorrect, index 22 of 28) average time (us): 1345.585640
test5 (incorrect, index 20 of 28) average time (us): 1231.725640
test6 (incorrect, index 18 of 28) average time (us): 1118.779720
test7 (incorrect, index 16 of 28) average time (us): 1001.602080
test8 (incorrect, index 14 of 28) average time (us): 887.423900
test9 (incorrect, index 12 of 28) average time (us): 767.226200
test10 (incorrect, index 10 of 28) average time (us): 654.863280
test11 (incorrect, index 8 of 28) average time (us): 521.429140
test12 (incorrect, index 6 of 28) average time (us): 401.944180
test13 (incorrect, index 4 of 28) average time (us): 283.397640
test14 (incorrect, index 2 of 28) average time (us): 156.242580
test15 (incorrect, index 0 of 28) average time (us): 40.600840
```

3. Created *threshold.c* (modeled after *client_test.c*) to find the average of the differences between single-character password timing differences, to produce the average amount of extra time it takes for the server to respond to a password with 1 additional correct character from another password.
   In this file, we send to the server “s” as the password to “shan” NUM_ITERS times, and then record the average time the server takes to process the password. We then repeat this for “sh,” “sha,” . . . “shannon_password_super_secret” and store these averages as well. We then take all of these average times and return the minimum average as our final threshold to be used in *hack.c*.

4. Created *hack.c* to execute the hack. Using the threshold calculated from *threshold.c*, this file takes the username “jason” and finds its associated password.

   This is the meat of the timing attack. The file starts with an empty string, and goes through a set list of characters (all lower case letters, all upper case letters, plus some special characters), appending each to the empty list and querying the server NUM_ITERS times, and finally averaging these times. (It tests “a,” then “b,” then “c,” etc.) For each, it checks if this average time taken by the server to process the password is significantly longer than the last recorded average time. It uses the threshold value from *threshold.c* to determine this. If the time passes the threshold, the character is added to the password, and the file begins to guess the next character. It loops until a correct password is found, then returns this correct password.

5. Created a bash script *hack_script* that compiles the code for the attack, runs the server in the background, runs *threshold.c*, then runs the server in the background again and runs *hack.c* with its command line argument as the result of the *threshold.c* run. *hack_script* is the sole file that must be run to reproduce our hack. An example of its output is on the next page.

6. At this point, most runs of *hack_script* resulted in a successful hack, but some runs failed due to outliers among threshold times. We adjusted *threshold.c* to eliminate outliers in the minimum-difference calculation, and added a multiplier factor to the final threshold value, lowering it a little bit so that *hack.c* could break this threshold more easily.

hack_script example output

```
[shannon.coupland@linuxlab005 final_project]$ ./hack_script

Running server...

Running threshold.c...
Connected to server.
getting timing from password s
getting timing from password sh
getting timing from password sha
getting timing from password shan
getting timing from password shann
getting timing from password shanno
getting timing from password shannon
getting timing from password shannon_
getting timing from password shannon_p
getting timing from password shannon_pa
getting timing from password shannon_pas
getting timing from password shannon_pass
getting timing from password shannon_passw
getting timing from password shannon_passwo
getting timing from password shannon_passwor
getting timing from password shannon_password
getting timing from password shannon_password_
getting timing from password shannon_password_s
getting timing from password shannon_password_su
getting timing from password shannon_password_sup
getting timing from password shannon_password_supe
getting timing from password shannon_password_super
getting timing from password shannon_password_super_
getting timing from password shannon_password_super_s
getting timing from password shannon_password_super_se
getting timing from password shannon_password_super_sec
getting timing from password shannon_password_super_secr
getting timing from password shannon_password_super_secre
getting timing from password shannon_password_super_secret
Threshold is 37.919064 microseconds.
This is the amount of time used to detect that the server has processed an additional password character.

Running server...


Running hack.c...
Connected to server.
current guess is j
current guess is j@
current guess is j@s
current guess is j@sO
current guess is j@sOn
current guess is j@sOn_
current guess is j@sOn_S
current guess is j@sOn_Se
current guess is j@sOn_Sec
current guess is j@sOn_SecR
current guess is j@sOn_SecRE
current guess is j@sOn_SecRE7
current guess is j@sOn_SecRE7_
current guess is j@sOn_SecRE7_p
current guess is j@sOn_SecRE7_p@
current guess is j@sOn_SecRE7_p@s
current guess is j@sOn_SecRE7_p@sS
current guess is j@sOn_SecRE7_p@sSW
current guess is j@sOn_SecRE7_p@sSW0
current guess is j@sOn_SecRE7_p@sSW0r

Password for jason is j@sOn_SecRE7_p@sSW0rD. HACKED!

[shannon.coupland@linuxlab005 final_project]$
```

Development challenges:

- Though *strcmp()* is supposedly a byte-by-byte string comparison, we got timing results for different password lengths that were indistinguishable from one another when using it. Therefore, we wrote our own function, *string_compare()*, in the server to compare two strings byte-by-byte.

- We did all of our development in the WashU Linux cluster so we could use the Linux *glock_gettime()* function. However, when we initially ssh'd into the cluster, we found that timing varied quite a bit between runs. Running *qlogin -qall* to gain access to a dispatched Linux lab machine before running *hack_script* greatly improved the hack's accuracy.

- We still had several issues with inconsistent timing between runs, likely due to varying overhead from OS process scheduling, the time taken for library function calls, etc. So we took the liberty of adding a 1-microsecond sleep for every character checked on the server side. This choice is definitely one that would not be made in a genuine server program, but it's possible that real-world servers could have additional overhead associated with checking each password character. Since this is a proof-of-concept exercise, we think this is acceptable, and this change stabilized our timing enough to enable us to reliably perform the attack.

- *perform_attack()* in *hack.c* will endlessly spin if it detects a character it thinks is correct, but is actually wrong (which happens if the threshold is too low or the number of averaged iterations is too low)

- As mentioned above, while *threshold.c* ran beautifully most of the time, some of the runs would produce a threshold that was wildly off, and sometimes even negative. This was due to unavoidable variability of timing on the server (note that we choose the minimum average time in *threshold.c*), and so we had two choices: either increase the number of iterations to discourage these outliers, or expand the file to ignore outliers. While the cleaner way would be to increase the number of iterations, this would cause the attack to take far too long (perhaps on the order of 15 minutes), and so we decided to ignore outliers. We did this by calculating the average, and then ignoring everything less than 0.75 times this average in our calculation. After this change, we never experienced any weird thresholds again.

Reproducing our Results

To reproduce our attack, perform the following steps:

1. Log into the WashU Linux lab cluster using the following terminal commands. (You only need to do this in one terminal window; the server that you run will run in the background.)
   ```
   ssh <wustl_key>@shell.cec.wustl.edu
   qlogin -qall
   ```

2. Pull the Github repo at https://github.com/shannon-coupland/cse433s.git into the Linux lab cluster, and navigate to the `final_project` folder within the repo.

3. Use the `ifconfig` command to determine your IP address.

4. Change the variable `ip` in both *threshold.c* and *hack.c* to match your IP address.

5. Run the bash script *hack_script* within the folder. This script will compile *server.c, threshold.c,* and *hack.c* and run them accordingly. (You may need to *chmod* the script to make it executable.) You will see print statements as these programs run that indicate what passwords have been sent to the server in *threshold.c*, the calculated threshold, and when a new character of the hacked password is found in *hack.c*. The script should take 2 minutes or less to run; most of the time will be in running the hack itself rather than training the threshold.

6. If you wish, you may add your own username/password pairs to the `CREDS` global variable in *server.c*. You may also train the threshold on different data by updating the `USERNAME` and `PASSWORD` global variables in *threshold.c* —however, to get a well-trained threshold, you need to provide a long password (say, at least 20 characters). However, usernames and passwords may not exceed 255 characters and cannot be the empty string, and passwords must only contain the characters a-z, A-Z, 0-9, !, @, and _.

7. If you would like to hack your own password, you can add a new entry to the `CREDS` global in *server.c* and change the `USERNAME` global in *hack.c* to match the username of the new entry and run *hack_script*. (Usernames and passwords may not exceed 255 characters and cannot be the empty string, and passwords must only contain the characters a-z, A-Z, 0-9, !, @, and _.)

Note that to save time, we have lowered the number of iterations in *threshold.c* and *hack.c*, but this comes at the price of some fluke runs. If the hack doesn't work the first time, try a few more times -- it should work if given enough runs. If you have more time to spare, you may increase the NUM_ITERS globals in *threshold.c* and *hack.c* to increase the chances of a successful attack.