Distributed Password Cracking with John the Ripper

Computer Security – Tufts Comp116

Author: Tyler Lubeck

Email: Tyler@TylerLubeck.com

Mentor: Ming Chow

Contents

Abstract

Password Cracking can be an incredibly complicated process. A very common tool for this process is John the Ripper (JtR). JtR is free and Open Source, and is largely distributed in compilable source code form. It uses three main modes of attack: single, wordlist, and incremental. The Single and Wordlist attacks compute hashes for supplied password lists and check those hashes against the hashes in the password files. Incremental is a brute force attack, and this is where things get complicated. Brute forcing passwords of under 5 characters is a pretty trivial process. However, as the password length grows, the complexity of brute forcing those passwords grows exponentially. This paper will deal with detecting accessible network computers on a *nix based network and distributing JtR processes across those computers in an attempt to speed up the incremental attack. The distributed processes will all check passwords of different lengths (so that machine A will check passwords of length 5, machine B passwords of length 6, etc.). This project will not deal with parallelizing JtR processes on a single machine. The main deliverables of this project will be a script to detect available and accessible machines on a *nix based network, and automatically distribute JtR processes amongst those machines. It will also look in to alternatives to JtR.

Introduction

John the Ripper (JtR) is a widely known, widely available open source password cracking tool. It is primarily used for cracking Unix passwords. It is distributed primarily in source code form, and can be compiled with several different options [1]. There are also several different "community builds" that are managed by the community and contain a variety of extra features. JtR has three workloads: generating hashes of passwords, generating passwords, and comparing the generated hashes with the hashes of the passwords to be cracked. Of these three workloads the most computationally intensive is generating password hashes [2]. Since John is a Brute Force cracker, this makes sense. There are three different modes of operation: single, wordlist, and incremental. Single and Wordlist modes both try passwords that are presumably more likely to occur. Wordlist mode requires a wordlist to be supplied when JtR is run, and generates hashes for each of the strings in the wordlist before comparing those hashes to the hashes of the passwords to be cracked. Typically, JtR will run both single and wordlist mode before moving on to incremental mode. Incremental mode generates passwords on the fly, then creates hashes for them, and then compares those hashes to the hashes of the passwords to be cracked. This mode is the one that requires time and processing power [2]. It's this mode that we aim to conquer by distributing JtR processes across multiple machines.

JtR comes with two features that can be helpful here: parallelization with OpenMP and Incremental segmentation. Parallelization is out of the scope of this paper, but segmenting the incremental mode is a very powerful tool. Essentially we can tell a JtR process to only run an incremental mode based on certain parameters, such as minimum length and maximum length [1].

To the Community

I chose this topic because I've always been interested in password cracking. This is because I've locked myself out of a number of development machines, my *family* has locked themselves out of every single machine they've ever owned, and because every so often I get an interest in seeing how secure my own passwords are. Especially for these first two instances, speeding up password cracking is a phenomenal tool. For the last instance, if an attacker might have the ability to speed up their cracking, then I should test against similar scenarios. The main intent of this paper is to describe how to distribute John the Ripper processes across different machines, while also exploring alternative password cracking tools.

Assumptions

While working on this paper, I explored distributing John the Ripper processes across the computer science network at Tufts University. The specific segment of the network that I used is built using Red Hat Enterprise Linux releases 5 and 6. The way this network is set up, a user's files are available at the same file path regardless of which machine on the network is being used. This helps tremendously when distributing JtR processes as described later on.

Action Items

Parallelizing John the Ripper

While an in-depth analysis is beyond the scope of this paper, it is worth mentioning that JtR does come with support for parallelization using Message Passing Interface (MPI). MPI enables JtR to communicate across a network, passing messages back and forth between a master controller and child processes [3]. Enabling MPI in JtR can be as simple as modifying the makefile before building [2]. There have been several attempts at building further parallelization of JtR, but as of the time of writing, none

have been incorporated in to the main JtR build. This is mainly because these projects "do not split the workload as elegantly or efficiently as the development team would like, they introduce unneeded dependencies of external libraries, and their code quality is often inadequate" [2].

Incremental Mode Revisited

Incremental mode was discussed briefly in the introduction, but here we'll dive in to it in-depth. Essentially, Incremental mode allows a user to tell JtR to only generate passwords that match a certain criteria. For instance, a user could specify to only generate passwords with seven characters. A user can also specify information about the character set to be used, and add extra characters to be used while checking [4]. Incremental mode configuration is done in the *john.conf* file supplied by JtR. A typical configuration may look like this [2]:

```
[Incremental:All5]     #Specify a name for the mode
File = $JOHN/all.chr   #Specify a file to get the character set from
MinLen = 0             #Specify a Minimum Password Length
MaxLen = 5             #Specify a Maximum Password Length
CharCount = 95         #Specify the number of characters in the character set
```

Each one of the specified modes will be distributed to a different machine to run. Therefore, as the goal is to gain speed during cracking, it is important to plan the modes to be distributed fairly via password complexity and not necessarily password length. Given a character set containing uppercase and lowercase alphanumeric characters, a password with only two characters has only 3,844 possible combinations and can be cracked almost instantly. A password with five characters has 916 Million possible combinations, and can take anywhere from a full day to just a single minute to crack. A password with eight characters has 218 Trillion possible combinations and can take anywhere from 692 years to 253 days to crack [5]. Our basic JtR processes will usually be cracking at the faster ends of these scales. Given that the length of the password is unknown at the time of running JtR, JtR will have to

check all possible lengths of passwords. To evenly distribute this workload amongst JtR processes, we must split by complexity which, as shown above, does not correlate directly to password length. When choosing our incremental modes it will make sense to allow one mode to check all passwords with lengths between 0 and 3, and have all other modes check passwords of just a single length. If there are enough machines available, it would make sense to distribute these further and describe modes that will check only uppercase, or only lowercase.

Network Distribution

The real power of this process is that JtR can have multiple instances running from a single executable. Each instance will dump passwords in to a single *john.pot* file. This allows us to place JtR on the user's path, and run it from any machine that the user has access to. Parsing the configuration file for incremental modes and running each mode on an individual machine enables us to distribute JtR across a network, and gather all of the cracked passwords in a single location. On a network such as the one described in the assumptions section, this gives us everything we need.

To actually access the machines on the network, the user must first set up a ssh key. This allows us to use passwordless entry for machines that we have access to. Then, a network scan can be initiated. This network scan can go through all of the IP addresses in a range and determine if they are accessible or not. This scan can dump the available IPs to a file that we can later use as a queue of machines to use. Simple grab the first one on the list and send it a JtR process. This does not do any logic to determine which machines are faster and which machines are more capable of performing intense CPU work, but on a network that is as homogenous as the one that I am testing on this does not exhibit too much of a problem. The script could be improved in such a way that it checks the CPU load of a machine before accessing it, and essentially making a priority queue in the file, but that exercise is left to later exploration of this topic.

Alternate Tools

In addition to JtR, which is largely my favorite, other tools do exist for password cracking. These are split in to two main categories: Distributed tools and GPU powered tools.

Distributed Tools

Distributed tools are largely the same as JtR, and many are built off of JtR itself. Many of the ones that are built on JtR have been abandoned, but of the ones that are still in development DJohn is a large project and a powerful tool [2]. DJohn stands for "Distributed John" and is in fact just that. It capitalizes on JtR's MPI support, and contains a client to monitor the various child processes [6].

GPU Powered Tools

GPU powered tools are becoming more and more powerful as the graphics processors in machines become both more powerful and cheaper. In fact, Amazon Web Services allows for instances that have multiple high-powered GPUs running in tandem on a machine. This makes for incredibly high power computing, and can greatly diminish the time needed to crack passwords if the password cracking tool is optimized for this use. Of course, where there's a will there's a way and multiple tools of this nature have either been released or are in development.

oclHashcat is one of these tools, and can run on both NVidia and AMD graphics cards. It is possible to run oclHashcat on up to 128 gpus, and focuses on brute force cracking much like JtR does [7].

Windows Tools

While we're discussing the topic, it's worth mentioning tools that are made for cracking Windows passwords. The largest tool that I'm away of is ophcrack. It utilizes rainbow tables, and can be run from a LiveCD on the machine that has passwords that need to be cracked. This can be an issue, because it requires the machine to be powered down [8].

Works Cited

[1] Openwall, "John the Ripper password Cracker," Openwall, [Online]. Available: http://www.openwall.com/john/. [Accessed 2 December 2013].

[2] solar, "Parallel and distributed processing with John the Ripper," [Online]. Available: http://openwall.info/wiki/john/parallelization. [Accessed 27 October 2013].

[3] D. Howe, "Message Passing Interface," Free On-Line Dictionary of Computing, 2010. [Online]. Available: http://foldoc.org/MPI. [Accessed 10 December 2013].

[4] OpenWall, "Customizing John the Ripper," 27 February 2011. [Online]. Available: http://www.openwall.com/john/doc/CONFIG.shtml. [Accessed 10 December 2013].

[5] The Home Computer Security Centre, "Password Recovery Speeds," 10 July 2009. [Online]. Available: http://www.lockdown.co.uk/?pg=combi. [Accessed 11 December 2013].

[6] L. Parravicini, "DJohn," [Online]. Available: http://ktulu.com.ar/blog/projects/djohn/. [Accessed 26 October 2013].

[7] "hashcat," [Online]. Available: http://hashcat.net/oclhashcat/. [Accessed 27 October 2013].

[8] ophcrack, "ophcrack," [Online]. Available: http://ophcrack.sourceforge.net/. [Accessed 14 December 2013].