TMTO[dot]ORG: Project Overview

Author: Jason R. Davis
Site: TMTO[dot]ORG

Table of Contents

Foreword Page 2
Chapter 1 – Build: *Construct* Page 3
Chapter 2 – Results and Analysis: *Intellect* Page 4
Chapter 3 – Bruteforce: *Epiphany* Page 6
Chapter 4 – Words: *Saunter* Page 8
Chapter 5 – Data flow Page 9
Index Page 10

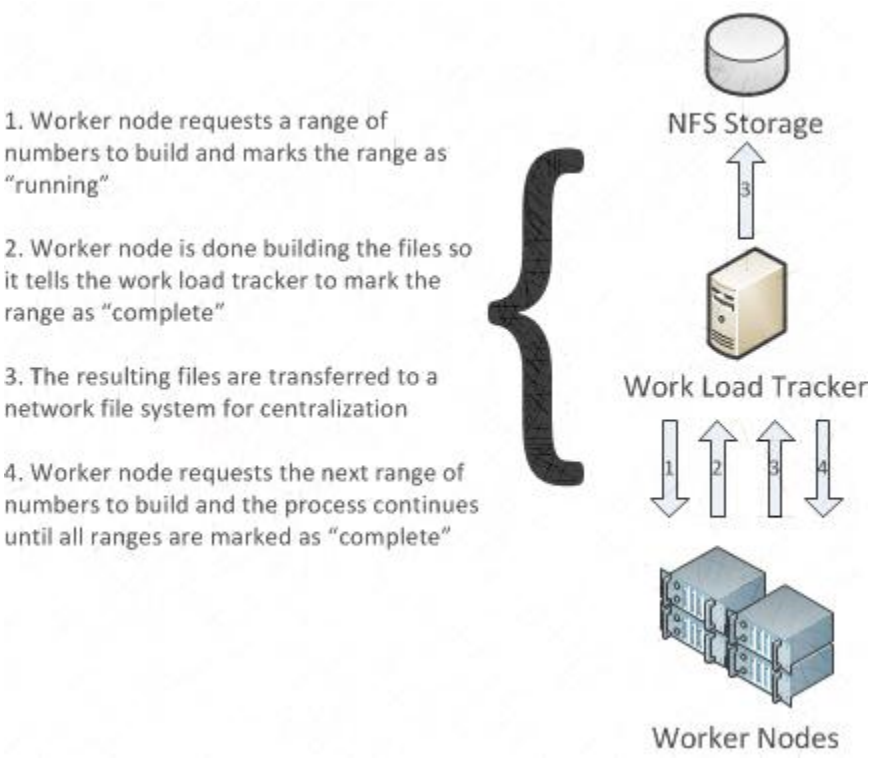
Created Date: November 22nd, 2010
Last Edited: November 22nd, 2010

Foreword

When I first started researching time-memory tradeoff in 2001, it was a selfish endeavor. I was dissatisfied with the existing methods of finding the clear text that a hash was created from. After a few months of researching new attack methods, I realized it wasn't the attack method that was the problem. It was the time frame in which an attack method could result in a success. This was a big sign that I needed to move towards a TMTO based attack method. For a long time I struggled with a lack of disk space. The raw data was simply too massive to store. I rolled in the piss water everyone was referring to as "rainbow tables" for a while. For me the creation time was too long for rainbow tables and their success rates weren't 100%. I wanted a "lossless" database that contained all the entries of a character set. One thing that came out of mulling was a way of generating strings by incrementing integers only. There was no need to store the hash or the entire string. The function creates a database of only numbers. This reduced the space requirements quite a bit. In result, the key space could be increased which meant more permutations could be stored. We were headed in the right direction.

Chapter 1 – Build: *Construct*

The goal of *Construct* was to efficiently distribute the build process. This would allow for much shorter creation times – which was the problem I had with rainbow tables. Things started out slowly with one node that had a dual core processor. I would launch two instances of the client and max out both CPU cores utilization. The client then checks in with a server and requests a work load. Once completed the raw files are moved to a network file system so that the results are centralized. Once the transfer is complete the client then requests more work. This is not a new way of distributing work via the client/server model, but it was a new way of building TMTO based tables.



Once all the files were copied to a central location a script that verified each files contents is correct was run. After verification, every file is prepared for import into the database. On huge tables this process proved to decrease build time exponentially. I am currently running *Construct* across 48 CPU cores and performance is holding solid. With each core the build time decrements – I'm sure that at some point performance would start to diminish, but I have not reached it yet.

With a few databases built and with a way to quickly generate new databases, I was ready to move onto the next phase: *Intellect*.

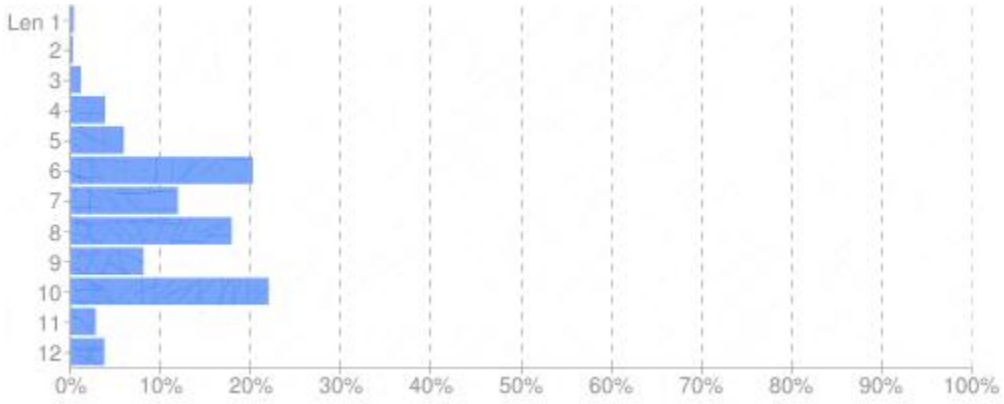
Chapter 2 – Results and Analysis: *Intellect*

The core purpose of intellect is to analyze the results data from all the queries against a TMTO database and track characteristics of the found results. These characteristics are the character set and string length of each found password. Why would I want to know these things? Simple: if I know the most common character sets and the string lengths, then I can build a more effective TMTO database based on those parameters.

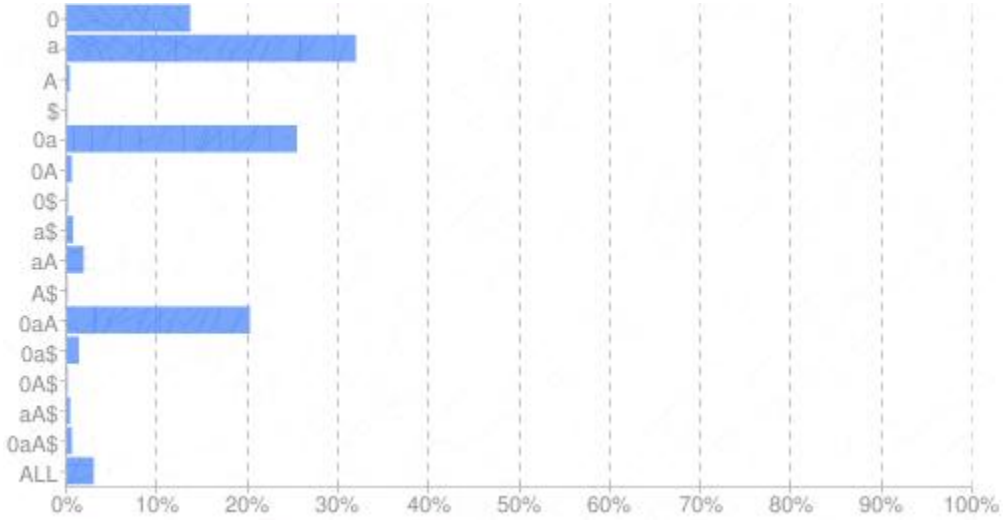
Blindly building a TMTO database with no supporting data is ineffective. Most of the entries will never see the touch of a query. Building future intelligence into *Construct* is one of the reasons *Intellect* exists.

Based on a current (2010-12-01) data set of approximately 462,000 found results it is quite clear where the string length distribution and character sets are most dominant.

String Length Distribution



Character Set Distribution



See Index 1.0 for character set definitions

The charts before outline the basic characteristics of the entire results set and provides a guideline for what kind of TMTO database should be built next. The bulk of the found results fall into the 6-10 string length with the “0”, “a”, “0a”, and “0aA” character sets. The “0aA” character set actually contains the “0”, “a”, and “0a” sets, so overall this can be simplified down to a TMTO database with the character set of “0aA” containing all permutations from 6 to 10 length. This is a speculated conclusion. Not an exact mathematical equation. There were 1.4 million results and 462,000 were found. This means there is a 66% (1,400,000/462,000 = 0.33) bias on the data and nearly a million results that need to be reversed to remove that bias. You cannot soundly make decisions unless you have analyzed the entire results set.

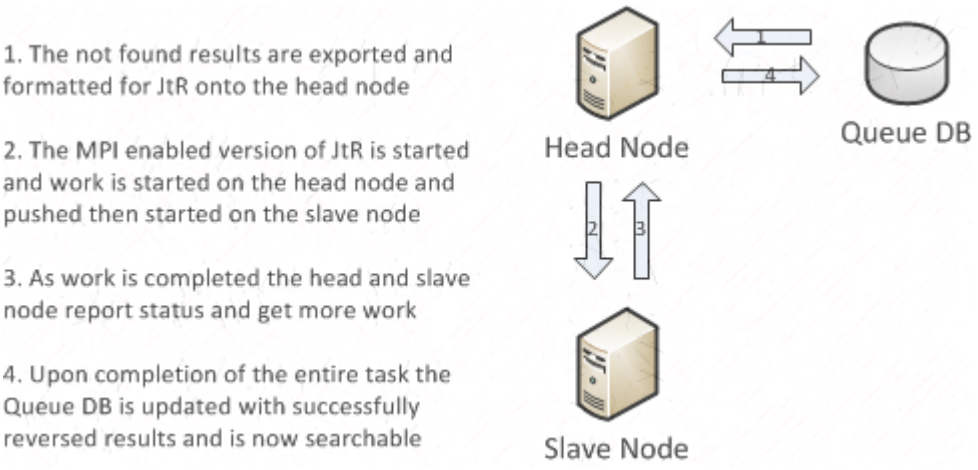
I started brain storming of ways to take the results that were not found and run an array of classic attack methods against it. Then update the results data with successfully reversed values and re-run the *Intellect* application against it. Over time it would decrease the bias and increase the overall effectiveness of the existing databases – which was a twofold benefit. This is how *Epiphany* was born.

Chapter 3 – Bruteforce: *Epiphany*

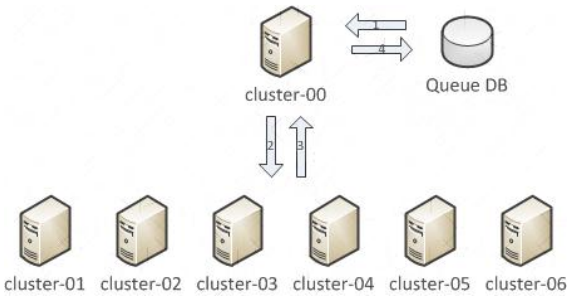
I want to start by saying that *Epiphany* was not always the 48 core and dual GPU accelerated cluster of hash cracking nodes it is today. Epiphany started with two nodes with a total of 4 cores – which were not very fast. It didn't matter to me though, I needed to start somewhere. The code base that would support everything was what I wanted to make rock solid and scalable. I knew that the better I coded it, the bigger it could get without running into stupid scalability problems. I made it modular so I could add new nodes, tweak an XML file, and then restart the job to increase performance.

The first item to solve was to decide what application I would use to bruteforce the key space that the TMTO databases didn't include. It had to be distributed and scalable to many CPU cores across many nodes. Initially I was quite lost. In order to find some direction I decided to choose a library that was designed to support embarrassingly parallel work. That was when I realized I had been using MPI (Message Passing Interface) for about 2 years to develop *Construct*. I knew I wanted to use MPI, so if I couldn't find a bruteforce application that was MPI capable – I would code one of my own. I eventually decided to use the MPI patched version of John the Ripper. I had always liked JtR, and it's capabilities met all the baseline requirements I need.

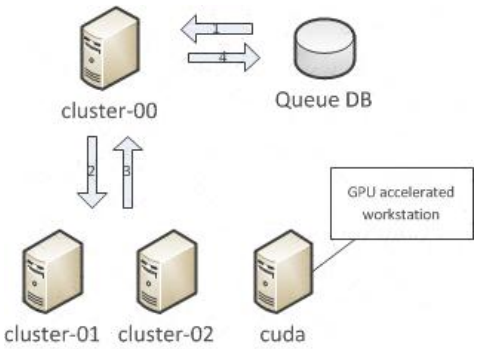
The first version of *Epiphany* was simple and worked great. I've diagramed the simple topology below.



I started the first few jobs using the many results that had been piling up since 2006. I ran those on this exact configuration for weeks. It took a few months, but in March, 2010 the topology grew to the diagram below.



The cluster had a total of 12 CPU cores and ran at 1.2 billion requests a second using the single MD5 algorithm. During this period of growth I modified the code of the TMTO[dot]ORG website to essentially queue the not found result into *Epiphany*. When the next job started, it would include new hashes and begin cracking. Eventually the queue grew as not found results were pouring in at a high rate. I had to implement a "retired" status for hashes that had been in the queue for more than 20 days. Hashes marked as retired would not be included in the next job. This kept the workload at a reasonable rate. During the month of October, 2010 I made the decision to drastically increase the infrastructure of the entire website and the *Epiphany* cluster. Most of the existing hardware was Intel based and the decision was made to move to AMD based hardware. In November, 2010 the new hardware was installed and the new website went live. The new *Epiphany* topology went into production use and is shown below.



The total number of nodes was reduced from 7 to 3, but the number of CPU cores increased from 12 to 48. There also was a substantial increase in the GHz each core ran at as well. Power consumption was substantially reduced until a GPU accelerated workstation was added to the cluster. The first iteration was a pair of NVIDIA GTX480 GPU cards, which consumed nearly 600W and generated a lot of heat at full speed. After a few weeks the NVIDIA GTX580 was released. I immediately purchased two of them and shortly after they were installed. There were two reasons for the immediate upgrade; 1. The power requirements and heat footprint were reduced 2. There is a 15-20% performance increase in the GTX580 compared to the GTX480.

The current CPU based cluster runs at 9.4 billion requests per second (single MD5). The current GPU based workstation runs at 3.1 billion requests per second (single MD5).

Chapter 4 – Words: *Saunter*

The next code iteration of *Epiphany* was to add the ability to process word lists on the GPU accelerated workstation – specifically hybrid attack methods, where each dictionary word is bonded with a bruteforce attack to create a dictionary word with a few numbers and/or letters and/or special characters on the left and/or right side – yes, I know that is a lot of "and/or". The hybrid attack method is generally much more effective than a simple traditional dictionary based attack method. This was a "must have". After gathering a whole bunch of word lists – which I combined, then sorted, and removed duplicates – I realized how sloppy and contaminated the entire list became. I needed a true word list, no special characters and no digits, that I could use as a base. Upon export of all the words, create "leet speak" permutations, uppercase first and/or/then last character permutations, and so forth from the list not vice a versa. This was a unique problem to solve. Where can I gather words (with the option of different languages) that are not contaminated (or at least 99% pure) or altered? Also I needed that source to be constantly updating; staying current so that new words are available on a regular basis. I decided to use the internet – specifically medical, news, technology and science web sites. This was the catalyst to the development of *Saunter*: an internet crawler that saunters webpages and parses out words that were clean and unaltered.

Saunter is integrated into *Epiphany* only by an import of the words – otherwise it is a separate project. That integration is diagramed in the next chart.

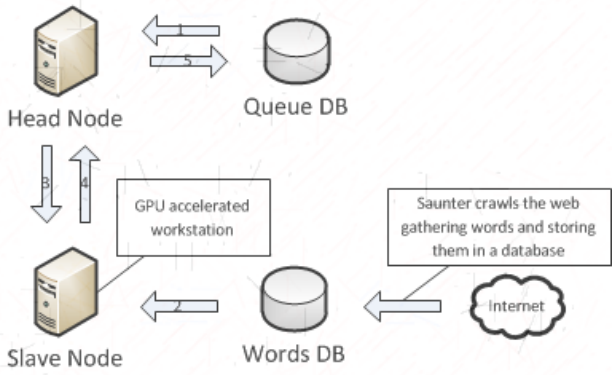
1. The not found results are exported and formatted for the CUDA application onto the head node

2. All words are exported on the GPU accelerated workstation – this includes additional permutations resulting in semi-hybrid style words

2. The CUDA application is started on the GPU accelerated workstation using the latest export of the words

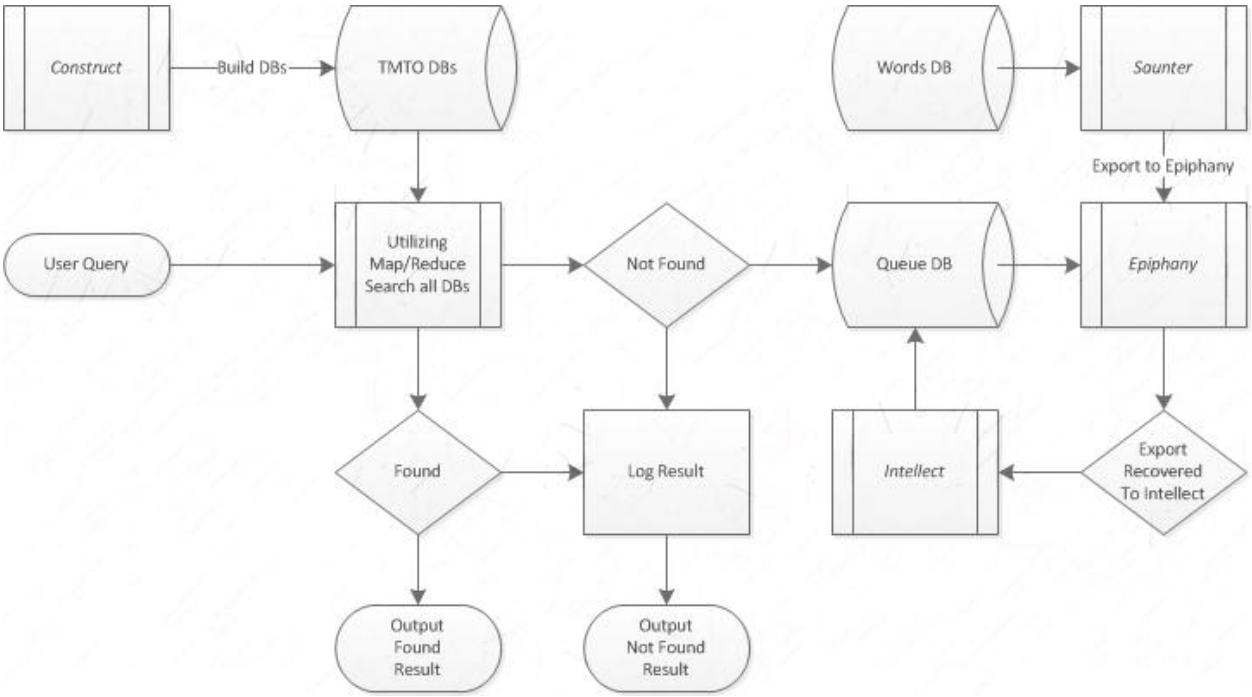
3. As work is completed the GPU accelerated workstation sends results back to the head node and starts the next block of work

4. Upon completion of the entire task the Queue DB is updated with successfully reversed results and is now searchable



Chapter 5 – Data flow

I believe that TMTO[dot]ORG should be adaptive and almost biological in its design. In that the software should learn and the data should grow on its own, at its own pace. At this point all the different components of the site could continue growing perpetually without interference or further development. Users need to understand just how powerful each query is on TMTO[dot]ORG. The higher the number of hashes that get processed the more TMTO[dot]ORG is allowed to grow and learn. Not found results are actually the most impacting part of the entire system. They create work for *Epiphany* and boost effectiveness in the long run. They also lower bias within *Intellect*. In essence the most powerful part of TMTO[dot]ORG is it's abilities to learn and reverse hashes on its own. I am in no hurry to create the largest and most elaborate database in the world. TMTO[dot]ORG is an adaptive tool that teaches me how users create passwords. At some point the application will have gathered enough data to help me make very educated decisions on the next iteration of TMTO database to build.



Index

1.0 - Character Set Definitions:

0: 0123456789
a: abcdefghijklmnopqrstuvwxyz
A: ABCDEFGHIJKLMNOPQRSTUVWXYZ
$: !@#$%^&*()-_=+
0a: 0123456789abcdefghijklmnopqrstuvwxyz
0A: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0$: 0123456789!@#$%^&*()-_=+
a$: abcdefghijklmnopqrstuvwxyz!@#$%^&*()-_=+
aA: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
A$: ABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()-_=+
0aA: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
0a$: 0123456789abcdefghijklmnopqrstuvwxyz!@#$%^&*()-_=+
0A$: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()-_=+
aA$: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()-_=+
0aA$: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()-_=+
ALL: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()-_=+[{]}\|;:'",<.>/?`~ and [SPACE]