How I became a password cracker

Cracking passwords is officially a "script kiddie" activity now.

by **Nate Anderson** - Mar 24 2013, 7:55pm CDT

Aurich Lawson

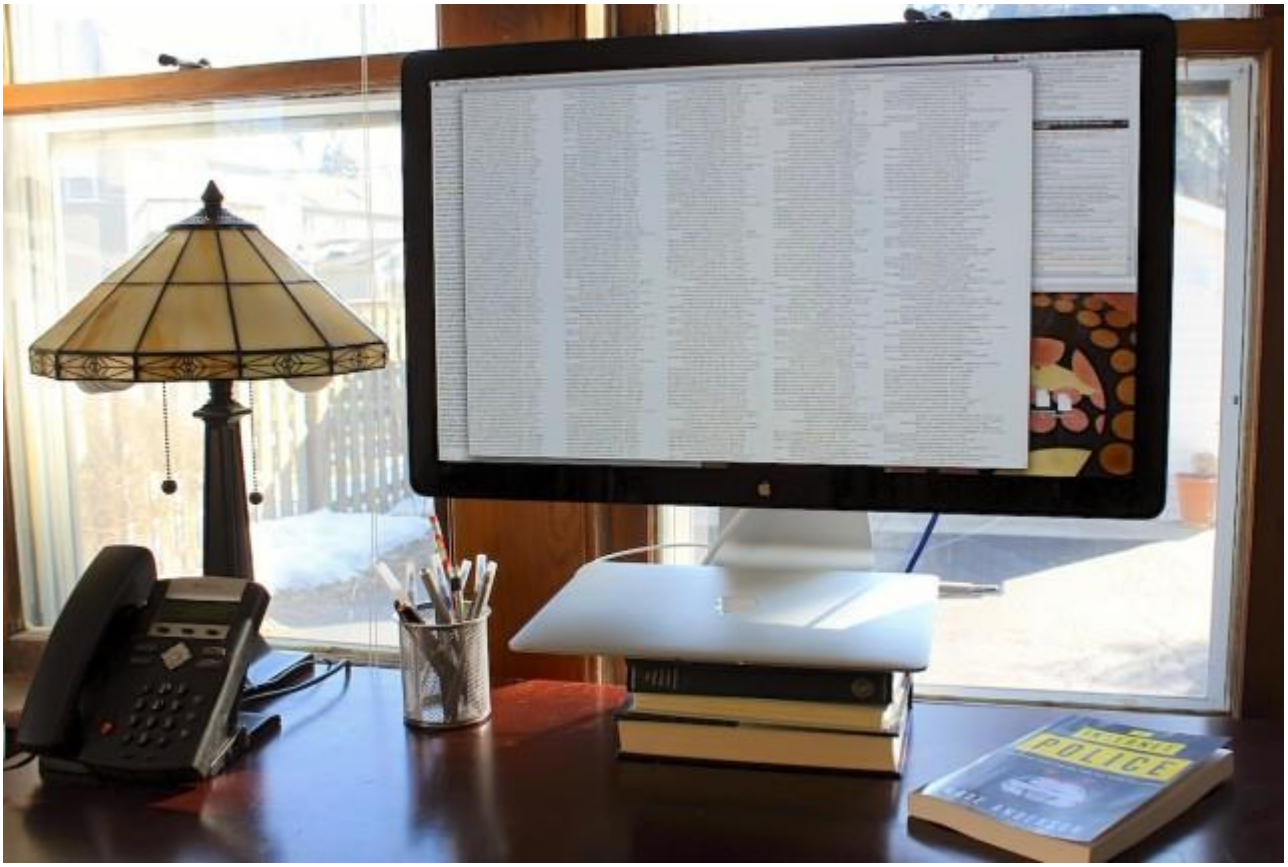
At the beginning of a sunny Monday morning earlier this month, I had never cracked a password. By the end of the day, I had cracked 8,000. Even though I knew password cracking was easy, I didn't know it was *ridiculously* easy—well, ridiculously easy once I overcame the urge to bash my laptop with a sledgehammer and finally figured out what I was doing.

My journey into the Dark-ish Side began during a chat with our security editor, Dan Goodin, who remarked in an offhand fashion that cracking passwords was approaching entry-level "script kiddie stuff." This got me thinking, because—though I understand password cracking conceptually—I can't hack my way out of the proverbial paper bag. I'm the very definition of a "script kiddie," someone who needs the simplified and automated tools created by others to mount attacks that he couldn't manage if left to his own devices. Sure, in a moment of poor decision-making in college, I once logged into port 25 of our school's unguarded e-mail server and faked a prank message to another student—but that was the extent of my black hat activities. If cracking passwords were truly a script kiddie activity, I was perfectly placed to test that assertion.

It sounded like an interesting challenge. Could I, using only free tools and the resources of the Internet, successfully:

1. Find a set of passwords to crack
2. Find a password cracker
3. Find a set of high-quality wordlists and
4. Get them all running on commodity laptop hardware in order to
5. Successfully crack at least one password
6. In less than a day of work?

I could. And I walked away from the experiment with a visceral sense of password fragility. Watching your own password fall in less than a second is the sort of online security lesson everyone should learn at least once—and it provides a free education in how to build a better password.



My not-particularly-l33t cracking setup: a 2012 Core i5 MacBook Air and a Terminal window. The five columns of text in the Terminal window are a small subset of the hashes I cracked by day's end.

“Password recovery”

And so, with a cup of tea steaming on my desk, my e-mail client closed, and some Arvo Pärt playing through my headphone, I began my experiment. First I would need a list of passwords to crack. Where would I possibly find one?

Trick question. This is the Internet, so such material is practically lying around, like a shiny coin in the gutter, just begging you to reach down and pick it up. Password breaches are legion, and entire forums exist for the sole purpose of sharing the breached information and asking for assistance in cracking it.

Dan suggested that, in the interest of helping me get up to speed with password cracking, I start with one particular easy-to-use forum and that I begin with "unsalted" MD5-hashed passwords, which are straightforward to crack. And then he left me to my own devices. I picked a 15,000-password file called MD5.txt, downloaded it, and moved on to picking a password cracker.

Password cracking isn't done by trying to log in to, say, a bank's website millions of times; websites generally don't allow many wrong guesses, and the process would be unbearably slow even if it were possible. The cracks always take place offline after people obtain long lists of "hashed" passwords, often through hacking (but

sometimes through legal means such as a security audit or when a business user forgets the password he used to encrypt an important document).

Hashing involves taking each user's password and running it through a one-way mathematical function, which generates a unique string of numbers and letters called the hash. Hashing makes it difficult for an attacker to move from hash back to password, and it therefore allows websites to safely (or "safely," in many cases) store passwords without simply keeping a plain list of them. When a user enters a password online in an attempt to log in to some service, the system hashes the password and compares it to the user's stored, pre-hashed password; if the two are an exact match, the user has entered the correct password.

For instance, hashing the password "arstechnica" with the MD5 algorithm produces the hash`c915e95033e8c69ada58eb784a98b2ed`. Even minor changes to the initial password produce completely different results; "ArsTechnica" (with two uppercase letters) becomes`1d9a3f8172b01328de5acba20563408e` after hashing. Nothing about that second hash suggests that I am "close" to finding the right answer; password guesses are either exactly right or fail completely.

Prominent password crackers with names like John the Ripper and Hashcat work on the same principle, but they automate the process of generating attempted passwords and can hash billions of guesses a minute. Though I was aware of these tools, I had never used one of them; the only concrete information I had was that Hashcat was blindingly fast. This sounded perfect for my needs, because I was determined to crack passwords using only a pair of commodity laptops I had on hand—a year-old Core i5 MacBook Air and an ancient Core 2 Duo Dell machine running Windows. After all, I was a script kiddie—why would I have access to anything more?

I started on the MacBook Air, which meant that I ~~had~~ got to use the 64-bit, command-line version of Hashcat rather than the Windows graphical interface. Now, far be it from me to sling mud at command line lovers, who like to tell me endless stories about how they can pipe `sed` through `awk` and then`grep` the whole thing about 50 times more quickly than those poor schlubs clicking their mice on pretty icons and menus. I believe them, but I still prefer a GUI when trying to figure out the many options of a complex new program—and Hashcat certainly fit the bill.

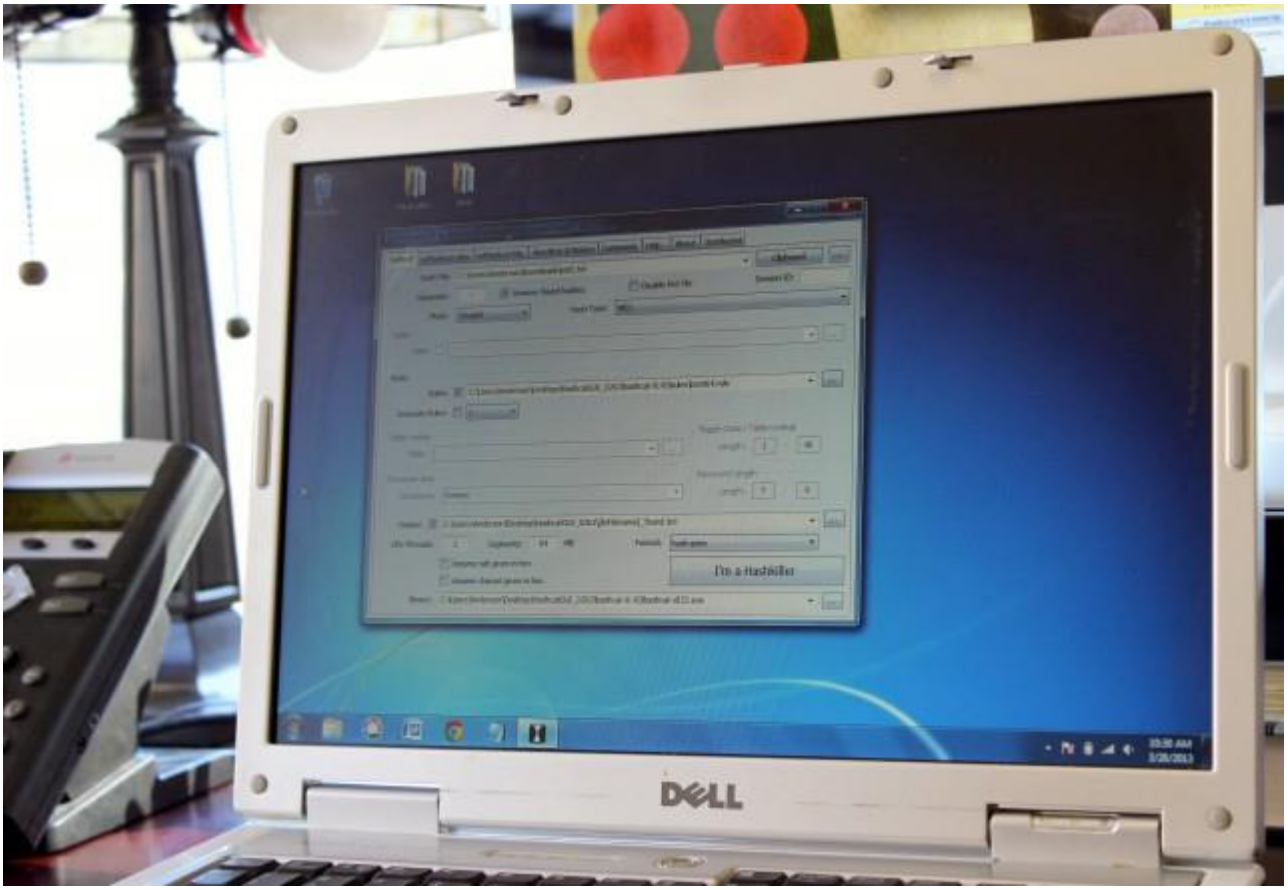
Still, this was *for science*, so I downloaded Hashcat and jumped into Terminal. Hashcat doesn't include a manual, and I found no obvious tutorial (the program does have a wiki, as I learned later). Hashcat's own help output isn't the model of clarity one might hope for, but the basics were clear enough. I had to instruct the program which attack method to use, then I had to tell it which algorithm to use for hashing, and then I had to point it at my MD5.txt file of hashes. I could also assign "rules," and there were quite a few options to do with creating masks. Oh, and wordlists—they were an important part of the process, too. Without a GUI and without much in the way of instruction, getting Hashcat to run took the best part of a frustrating hour spent tweaking lines like this:

```
./hashcat-cli64.app MD5.txt -a 3 -m 0 -r perfect.rule
```

The above line was my attempt to run Hashcat against my MD5.txt collection of hashes using attack mode 3 ("brute force") and hashing method 0 (MD5) while applying the "perfect.rule" variations. This turned out to be badly misguided. For one thing, as I later learned, I had managed to parse the syntax of the command line incorrectly and had the "MD5.txt" entry in the wrong spot. And brute force attacks don't accept rules, which only operate on wordlists—though they do require a host of other options involving masks and minimum/maximum password lengths.

This was a bit much to muddle through with command-line switches. I embraced my full script kiddie-ness and switched to the Windows laptop, where I installed Hashcat and its separate graphical front end. With all options

accessible by checkboxes and dropdowns, I could both see what I needed to configure and could do so without generating the proper command line syntax myself. Now, I was gonna crack some hashes!



Could an aging Dell laptop make me a "hashkiller"?

The first hit

I began with attack mode 0 ("straight"), which takes text entries from a wordlist file, hashes them, and tries to match them against the password hashes. This failed until I realized that Hashcat came with no built-in worldlist of any kind (John the Ripper does come with a default 4.1 million entry wordlist); nothing was going to happen unless I went out and found one. Fortunately, I knew from reading Dan's2012 feature on password cracking that the biggest, baddest wordlist out there had come from a hacked gaming company called RockYou. In 2009, RockYou lost a list of 14.5 million unique passwords to hackers.

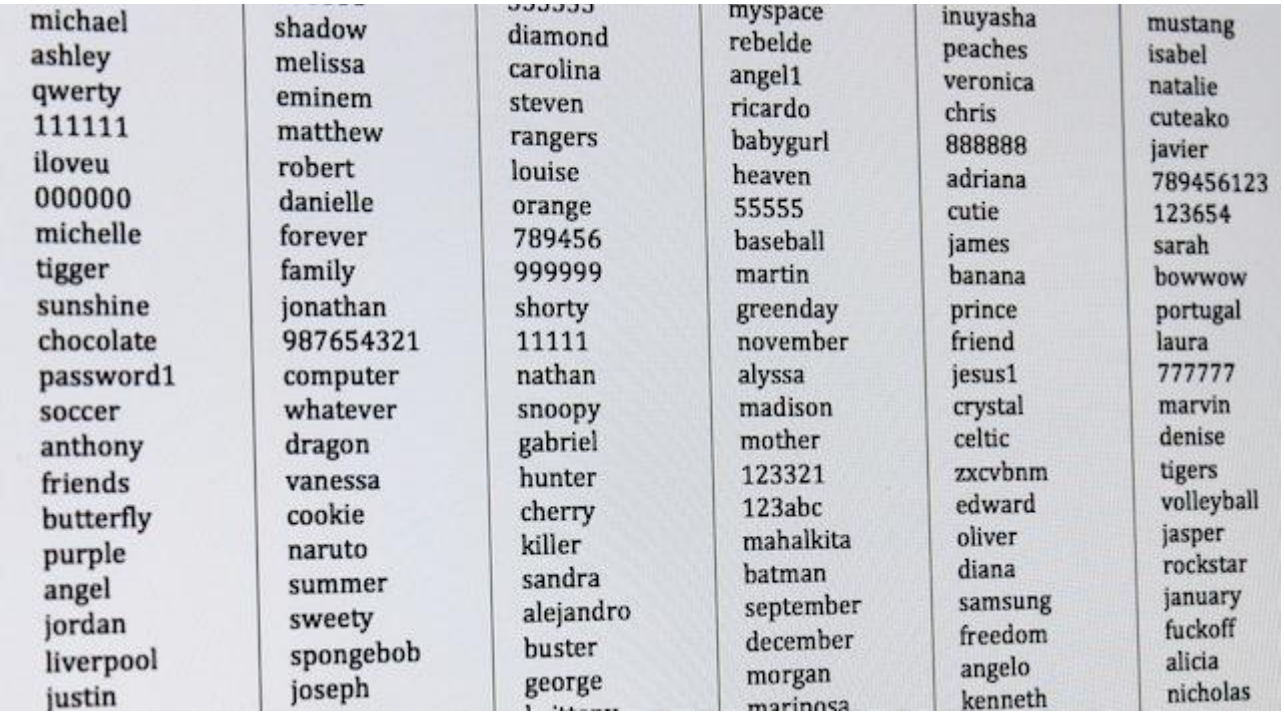
As Dan put it in his piece, "In the RockYou aftermath, everything changed. Gone were word lists compiled from Webster's and other dictionaries that were then modified in hopes of mimicking the words people actually used to access their e-mail and other online services. In their place went a single collection of letters, numbers, and symbols—including everything from pet names to cartoon characters—that would seed future password attacks." Forget speculation—RockYou gave us a list of actual passwords picked by actual people.

Finding the RockYou file was the work of three minutes. I pointed Hashcat to the file and let it rip against my 15,000 hashes. It ran—and cracked nothing at all.

At this point, sick of trying to puzzle out best practices by myself, I looked online for examples of people putting Hashcat through its paces, and so ended up reading a post by Robert David Graham of Errata Security. In 2012, Graham was attempting to crack some of the 6.5 million hashes released as part of an infamous hack of

social network LinkedIn, he was using Hashcat to do it, and he was documenting the entire process on his corporate blog. Bingo.

He began by trying the same first step I had tried—running the complete RockYou password list against the 6.5 million hashes—so I knew I had been on the right track. As in my attempt, Graham's straightforward dictionary attack failed to produce many results, identifying only 93 passwords. Whoever had hacked LinkedIn, it appeared, had already run such common attacks against the collection of hashes and had removed those that were simple to find; everything that was left presumably would take more work to uncover.



michael	shadow	diamond	myspace	inuyasha	mustang
ashley	melissa	carolina	rebelde	peaches	isabel
qwerty	eminem	steven	angel1	veronica	natalie
111111	matthew	rangers	ricardo	chris	cuteako
iloveu	robert	louise	babygurl	888888	javier
000000	danielle	orange	heaven	adriana	789456123
michelle	forever	789456	55555	cutie	123654
tigger	family	999999	baseball	james	sarah
sunshine	jonathan	shorty	martin	banana	bowwow
chocolate	987654321	11111	greenday	prince	portugal
password1	computer	nathan	november	friend	laura
soccer	whatever	snoopy	alyssa	jesus1	777777
anthony	dragon	gabriel	madison	crystal	marvin
friends	vanessa	hunter	mother	celtic	denise
butterfly	cookie	cherry	123321	zxcvbnm	tigers
purple	naruto	killer	123abc	edward	volleyball
angel	summer	sandra	mahalkita	oliver	jasper
jordan	sweety	alejandro	batman	diana	rockstar
liverpool	spongebob	buster	september	samsung	january
justin	joseph	george	december	freedom	fuckoff
			morgan	angelo	alicia
				kenneth	nicholas

A small slice of the infamous RockYou wordlist.

Nate Anderson

Perhaps the same culling had been done on my list of hashes, with the hacker—oops, I mean "password recovery specialist"—releasing a list of only those hashes he couldn't easily crack. This might explain my low success rate so far, and the thought gave me some much-needed encouragement after two fruitless hours of trying to convince myself I wasn't as bad at this as I clearly was. I was, after all, in the same boat as a noted security researcher like Graham, and here we were sailing together on the same journey, docking at the same ports, encountering the same difficult sea conditions.

I looked at Graham's next step, which was to apply "rules" to the RockYou dictionary. Such rules dramatically expand the usefulness of a particular wordlist by generating variations of those words that might be used to form someone's password. For instance, one rule might take every word in a wordlist and add two numbers to the end of it, starting at 00 and progressing up to 99; this would hit every password from "arstechnica00" to "arstechnica99." People commonly form passwords this way, so such a rule would likely produce many hits but at the expense of a 100x increase in required number of guesses.

All serious password cracking programs can handle such rules, but the format for creating them differs a bit—and besides, to create solid rules you first need to know the most common ways of forming passwords. While I could guess at a few of these, I was far from an expert and in any case had no wish to spend several hours learning how to translate my own terrible rules into proper Hashcat syntax. Fortunately, I didn't have to. Hashcat comes pre-

supplied with a number of complex rule sets, text files with names like "d3ad0ne.rule" and "T0XIC.rule." Graham chose to begin with "best64.rule," which as the name suggests contains the most productive known rules. I began with it, too.

In best64.rules, each word in the wordlist is first hashed without any alteration, then the fun begins. At first, the changes are simple: each word gets reversed, or each has its case switched. Next comes a "simple number append," which adds a single digit (0-9) to the end of each word. After that comes "special number append," which tries 14 specific number combinations that appear most commonly at the end of passwords:

```
## special number append
$0 $0
$0 $1
$0 $2
$1 $1
$1 $2
$1 $3
$2 $1
$2 $2
$2 $3
$6 $9
$7 $7
$8 $8
$9 $9
$1 $2 $3
```

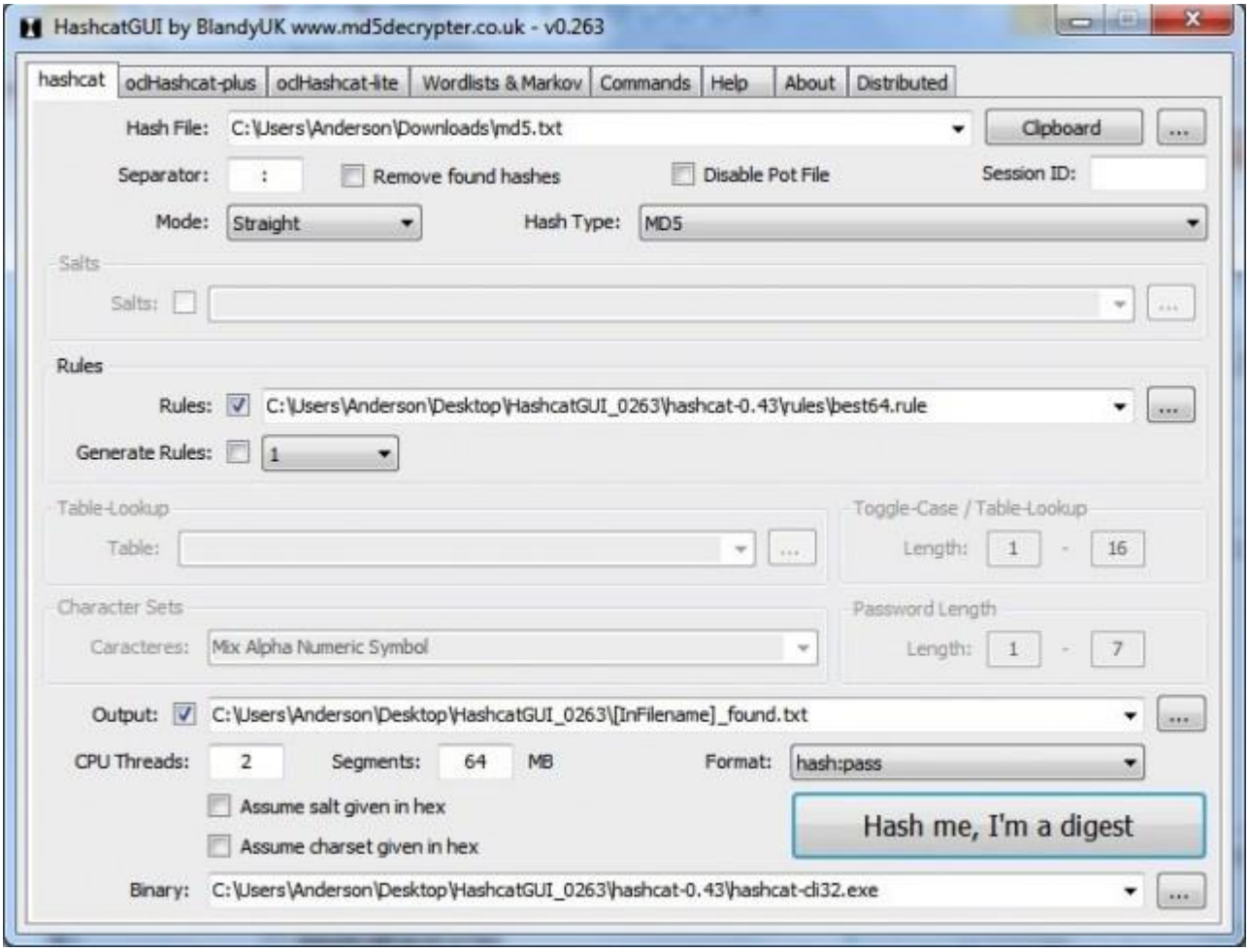
Each entry in the wordlist then has "e" added to the end, then "s." Next comes a "high frequency overwrite" replacing the last few letters of each word with the most common endings found in passwords (-er, -y, -123, -man, -dog...). High frequency prepends are next (most common first character: "1") and then high frequency overwrites at the start of words. Each wordlist item is "leetified" by replacing "o" with "0" and "i" with "!" and "e" with "3." Words are "undoubled," words have their most common suffixes removed, the letters in each word are rotated...

The final set of rules is labeled only "unknown" and shows how complex the rule syntax can get:

```
## unknown
*04 +0 '4
*05 x03 d '3 p1
+0 +0 +0 +0 +0 +0 +0 +0
+0 +0 +0 x12
Z4 '8 x42
Z5 '6 x31 ] p1
Z5 *75 '5 { x02
d x28 Y4 '4 d
f *A5 '8 x14
p2 '7 p1 x58
x14 d p2 '6
```

I double-checked that my attack method was "straight," made sure to select the RockYou wordlist, and set the rules to best64.rule. If this attack failed to produce a single hit, I resolved to spend the rest of the day drinking. I clicked the cheekily labeled button "Hash me, I'm a digest" to start the attack and sat back to wait.

Three seconds later, the complete set of rule-based variants had been run against my MD5.txt hash file—and I had scored a hit against my hash file. I eagerly opened the text file containing the list of discovered hashes to see my first cracked password: "czacza."



The settings for my first successful attack. Note that wordlists are selected on a separate tab.

I was a god among men, a password-cracking colossus striding through the world of mortals. The Secrets of the Hash were mine!

But as I exulted in the exercise of my new powers, doubt crept quickly in: shouldn't someone who knew the Secrets of the Hash be able to crack more than 1 out of 16,051 hashes using a rule-varied RockYou dictionary attack? Graham, by contrast, had nailed 688,000 using the exact same technique, and that was on a hash dataset already stripped of most common passwords. "Czacza" felt a little thin by comparison.

Now that I had the system up and running, it was time to find out why it wasn't running *well*.

Let's get cracking

I began a series of experiments designed to isolate the problem. The first involved my hash database; perhaps it had simply been stripped of all the easy-to-find passwords and all that remained inside were 12-digit passwords requiring a six-month brute force attack to crack. So I returned to my "password recovery" forum and downloaded a new list of 17,000 MD5 hashes and got to work.

The results were the same. Running the RockYou dictionary through a "straight" attack modified by the best64.rule file produced only a single result: "tawtaw." This was faintly unsettling for a reason I couldn't quite name. A nagging voice in the back of my head suggested that only being able to crack passwords that followed a single specific pattern—in this case, three characters, repeated twice—was a Bad Sign. I couldn't decide why this might be a bad sign, exactly, so I shushed the nagging voice and returned to work.

New attacks on my 17,000 hashes produced slightly better results but at the cost of dramatically increased cracking times. A "combination" attack would have taken an estimated 14.5 hours to complete, though it did crack three passwords in the three minutes I let it run. A "combinator" attack cracked a single non-patterned password ("cp2009") in seconds. And returning to a "straight" attack but using the d3ad0ne.rule file instead of best64.rule earned me two more hashes ("1234567" and "aaaaa1") after the six minutes it took to run. I now had six hashes cracked. Progress!

Hashcat in operation—even the GUI versions launch the command line executable to do their actual cracking.

The results still seemed absurdly low, and the fact that I continued to crack largely patterned passwords gave me pause. Had both of these hash files really been stripped of every single RockYou password along with their rule-based variants?

Since wordlist attacks, even using rules, were producing few hits, I decided to simply "brute force" the hash file. A brute-force attack simply iterates through every available option with minimal intelligence, trying "aaaaa" and then "aaaaab" and then "aaaaac" until it every possible permutation has been tried. I was extraordinarily thankful for the Hashcat GUI here, which simplified the configuration options needed to mount a brute-force attack from the command line.

A brute-force attack requires numerous options, including the lengths of the attempted passwords and a mask built up from character sets like these:

```
?l = abcdefghijklmnopqrstuvwxyz
?u = ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d = 0123456789
?s = !"#$&'()*+,-./:;<=>?@[\]^_`{|}~
?a = ?l?u?d?s
?h = 8 bit characters from 0xc0 - 0xff
?D = 8 bit characters from german alphabet
?F = 8 bit characters from french alphabet
?R = 8 bit characters from russian alphabet
```

I could certainly put together a mask to brute force all passwords that begin with, say, one uppercase letter, follow with five lowercase letters, and end with a symbol (?u?l?l?l?l?s), but the GUI made it even simpler to select options like "lowercase alphanumeric" with particular length constraints.

Brute forcing can take extraordinary amounts of time as the password length increases; indeed, each additional character in a password exponentially increases the brute force cracking time, and passwords over nine or 10 characters could take weeks or months to crack on consumer hardware. To have any chance of success, I began with six-character passwords because I had already found some of this length in the hash file. I also limited myself to lowercase alphanumeric characters and symbols—leaving uppercase off for now because few people voluntarily create SHOUTY PASSWORDS.

The complete run of six-character alphanumeric symbols would take five hours, Hashcat informed me, but it started showing results almost immediately. Two minutes in, I cracked 22 hashes. Four minutes in, it was 28. After six minutes, I was up to 32. Many of the cracked hashes consisted of nothing but lowercase letters, so I canceled the attack and aimed for something more efficient by throwing out the numbers and symbols. The resulting run went faster, cracking 334 hashes in one minute and revealing passwords like "violet" and "ludwig" and "august" and "peanut."

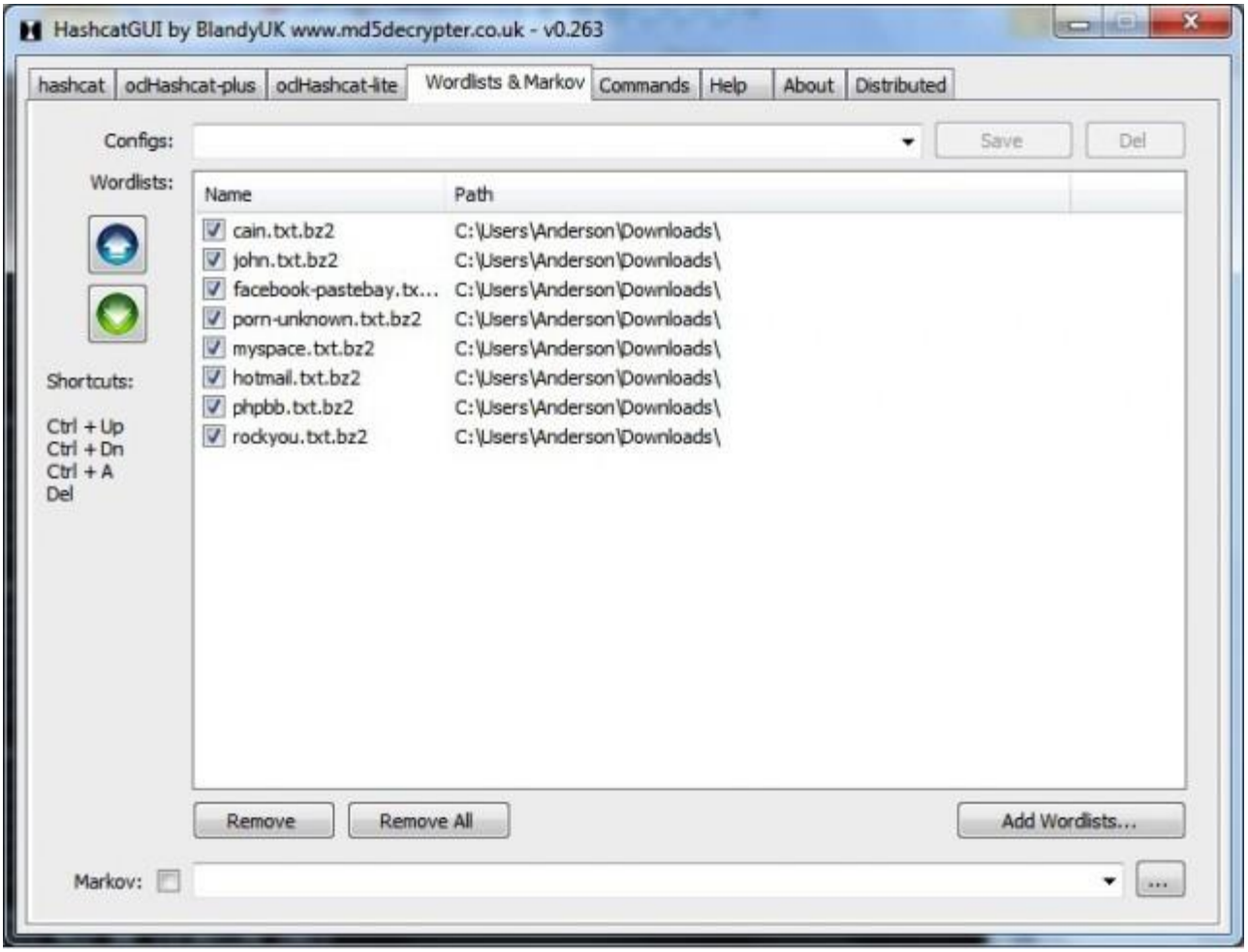
Boom! Who's a script kiddie now?

Many of the cracked passwords were people's names, while others were common English words. It did seem odd that the vaunted RockYou wordlist wouldn't contain things as obvious as "august" or "violet," but I dismissed the thought and went hunting for further wordlists to see if I could increase my success rate. I grabbed an English dictionary wordlist along with a huge collection of first and last names slurped up from that massive corpus of data known as Facebook. Just for the heck of it, I grabbed a German dictionary too.

I ran them in a straight attack against my 17,000 hashes, expecting massive results. Instead I got nothing—not even passwords like "violet," which I knew were in both the wordlist and the hash file.

The nagging voice in my head got louder, the one which had first suggested some time ago that I had a more fundamental problem than I cared to admit. Indeed, as I peered more closely at my collection of files, I began to suspect that somewhere along the way I had made a huge mistake—and that I had not in fact cracked a single hash using a wordlist.

The walk of shame



What could possibly be wrong?

Remember when I said that I was no expert at this stuff? Well, I'm about to prove it.

Each of the wordlist files I had downloaded were ".bz2" files—that is, bzip2 compressed files. Now, I had recognized this, but I had not bothered to decompress the files before using them (especially in the case of Rock You; the file was quite large). This didn't appear to be a problem because the GUI version of Hashcat threw no errors when I loaded it up with these .bz2 wordlists. Everything appeared to operate normally, and I assumed that Hashcat simply contained an internal version of the open-source bzip2 decompressor and unpacked the wordlists before using them.

But the failure of my English-language wordlist in particular convinced me that something had gone wrong here, so I decompressed all the .bz2 wordlists and added them back into the program. Perhaps, I reasoned, my modest

success so far as a password cracker was due only to rules or to brute force, both operating in the absence of an actual wordlist—which might explain why the early passwords I cracked followed patterns.

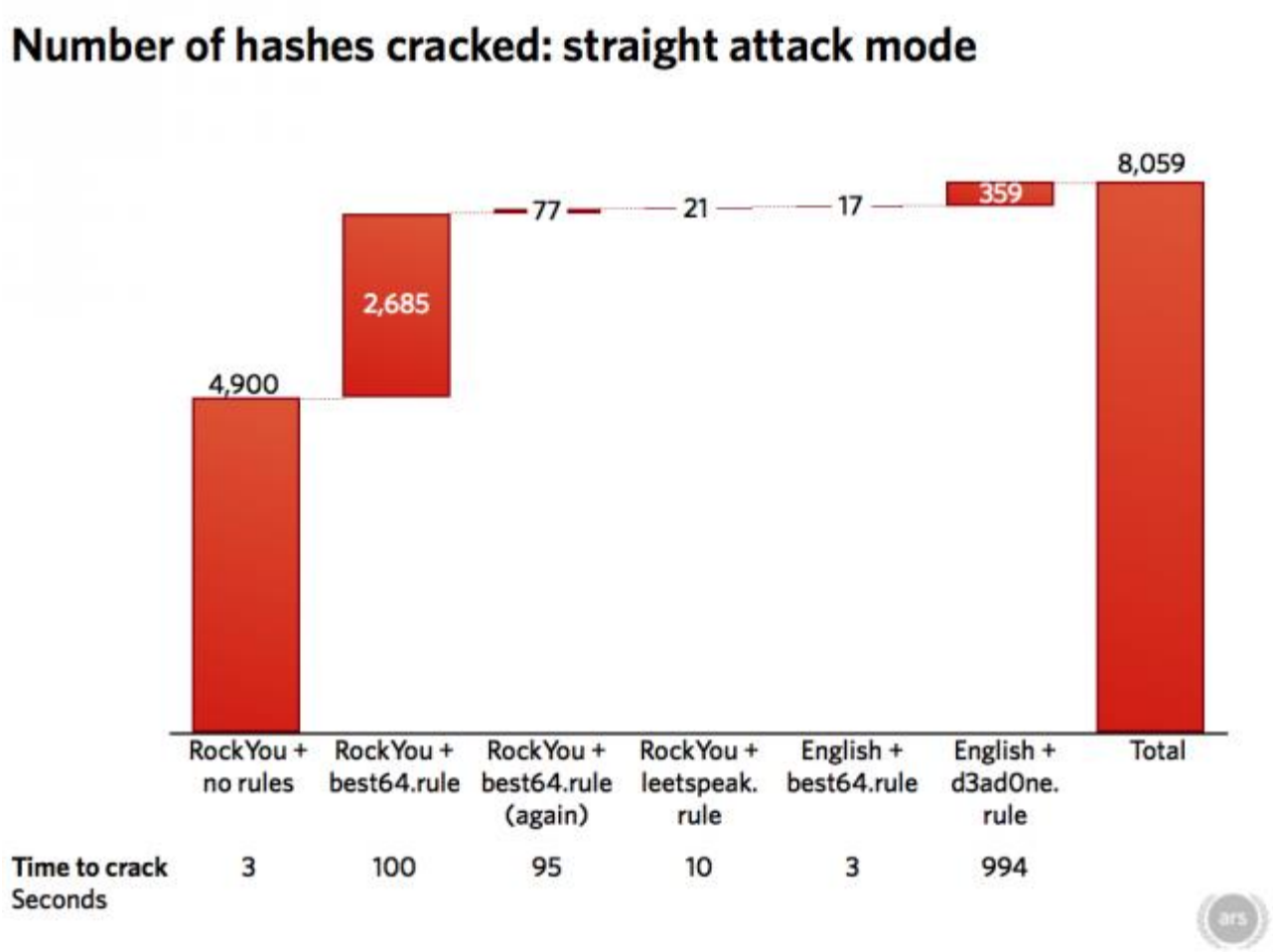
I re-ran a "straight" attack on my 17,000 hash file using each of my wordlists with no additional rules whatsoever.

The result: I cracked 4,976 hashes in one minute, most of them coming from the RockYou wordlist, which clearly lived up to its hype now that it was uncompressed and working. Even without rules, I had cracked far more complex passwords than before, things like "softball24" and "butterfly5."

So I added best64.rule back into the mix and let Hashcat rip. In only 16 minutes it had applied every rule to every word in every wordlist I had, smashing through 7,553 hashes. After my dumb misstep, I had finally solved the password-cracking puzzle. I stared in awe at the huge list of passwords and short amount of time need to crack them. This was getting fun.

Refining the technique

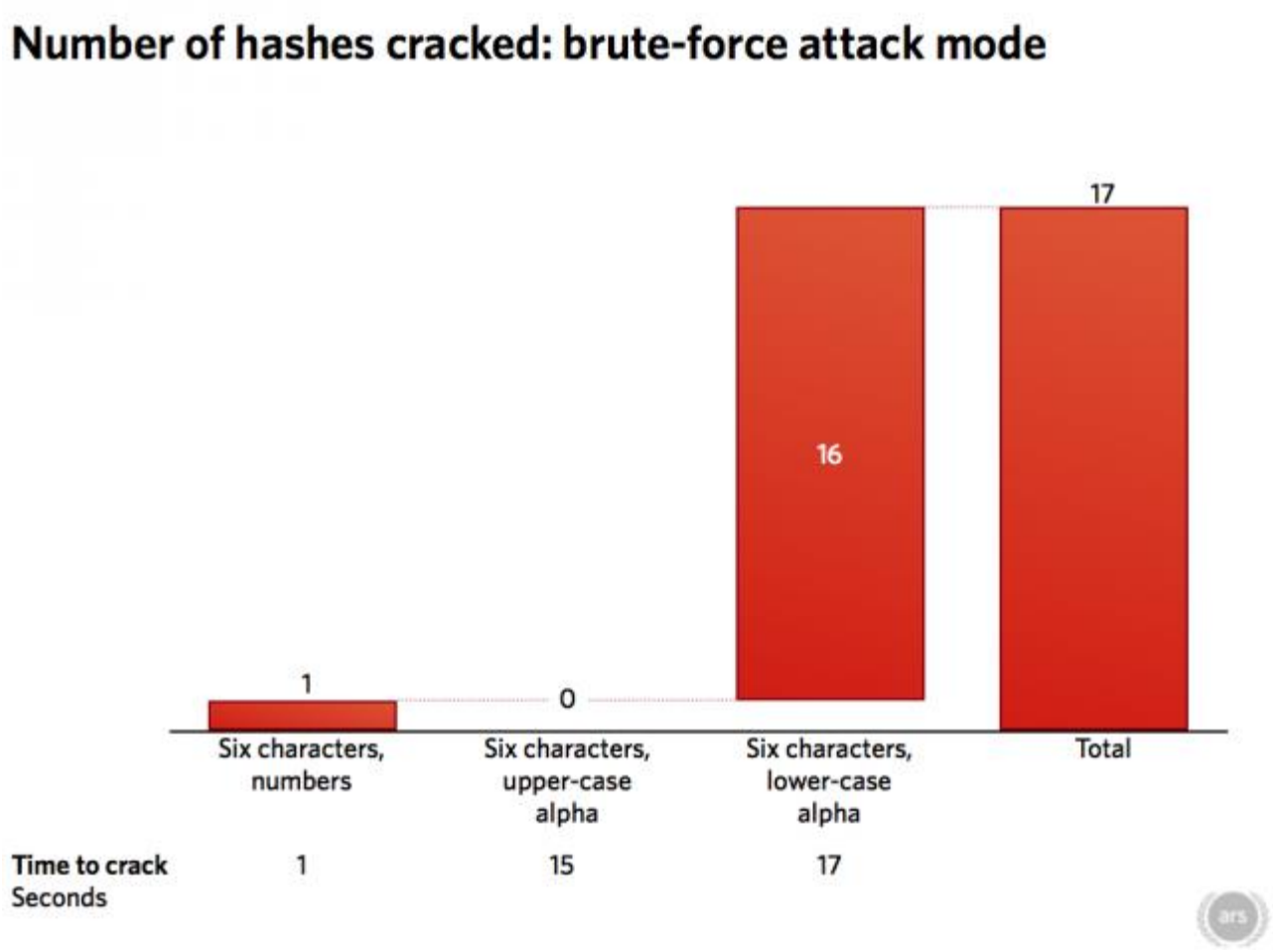
By this point I had puzzled out how Hashcat worked, so I dumped the GUI and switched back to the command-line version running on my much faster MacBook Air. My goal was to figure out how many hashes I could crack in, say, under 30 minutes, as well as which attacks were most efficient. I began again on my 17,000-hash file, this time having Hashcat remove each hash from the file once it was cracked. This way I knew exactly how many hashes each attack solved.



Ars Technica

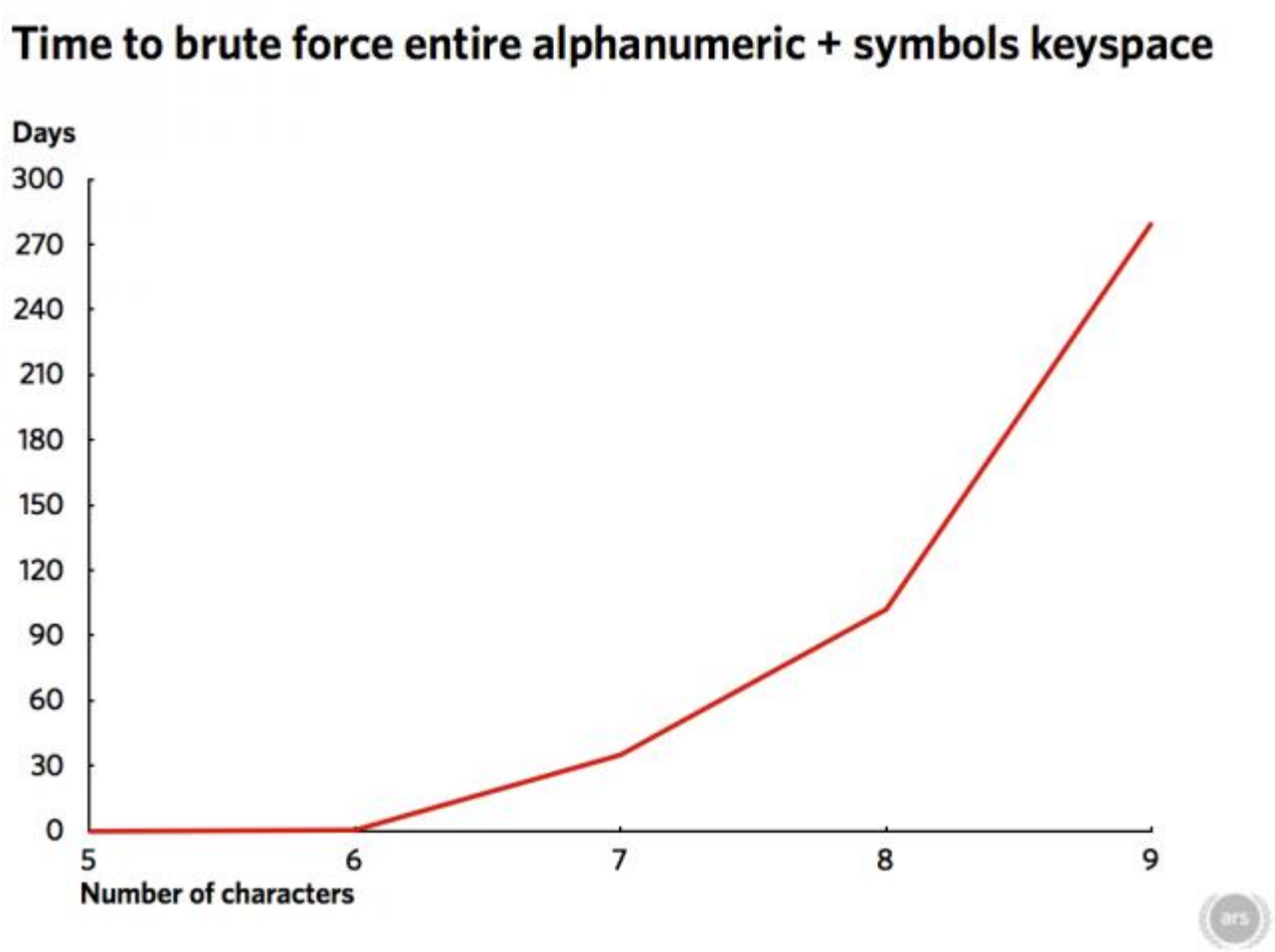
This set of attacks brought the number of uncracked MD5 hashes down from 17,000 to 8,790, but clearly the best "bang for the buck" came from running the RockYou list with the best64.rule iterations. In just 90 seconds, this attack would uncover 45 percent of the hashed passwords; additional attacks did little more, even those that took 16 minutes to run.

Cracking a significant number of the remaining passwords would take some much more serious effort. Applying the complex d3ad0ne.rule file to the massive RockYou dictionary, for instance, would require more than two hours of fan-spinning number-crunching. And brute force attacks using 6-character passwords only picked up a few additional results.



Ars Technica

Why did I limit myself to six characters? In a word, time. To see just how quickly brute force attacks could spiral out of control, consider the following chart. It shows the initial time estimates Hashcat gave for running full alphanumeric + symbols attacks on my 8,790 remaining hashes as I increased the character count of the passwords.



The amount of time needed to brute force passwords increases quickly when using my laptop's CPU.

Ars Technica

The lesson was clear: I *could* crack every last hash in the file—but I'd probably need the better part of a year to do it, assuming my machine didn't simply collapse under the strain.

But who needs to be a completist about cracking hashes? I was at least in a position now to crack thousands of passwords in mere minutes. I could get everything from common passwords (iloveyou1, iloveyou13, iloveyou19, iloveyou81) to odd passwords (hahapoop3) to long passwords (rangefinder12) to passwords incorporating mixed case characters, numbers, and symbols (Jordan2!). Had I been the one who "liberated" this particular set of hashes, I would have been well-placed to wreak havoc on thousands of accounts—more than enough for some real mischief.

Speedups and slowdowns

A small selection of MD5 hashes, followed by the password that produced the hash.

Nate Anderson

While I quickly gained both a conceptual and practical grasp on password cracking, I remain firmly in the realm of the script kiddie. Cracking a set of hashes generated by a single pass through the MD5 algorithm is computationally modest, even on aging laptop CPUs. A variety of changes to either my own setup or the hashing technique could make this process faster still—or make it exponentially slower.

Faster

More powerful CPUs. My little experiment was conducted on commodity laptop hardware better suited to a writer than a number cruncher. That was by design, as I wanted to see just how feasible password cracking was without unusual amounts of power, but it's important to remember that any serious cracker will use a rig sporting more memory and a more powerful processor than the Core 2 Duo and Core i5 CPUs I used.

And there's no reason to limit oneself to the kind of hardware found in discrete computers, either. With the rise of cloud computing, an ambitious hacker could rent some potent processing power from Amazon and deploy Hashcat in numerous virtual machines, each running a different attack on the hash database being targeted.

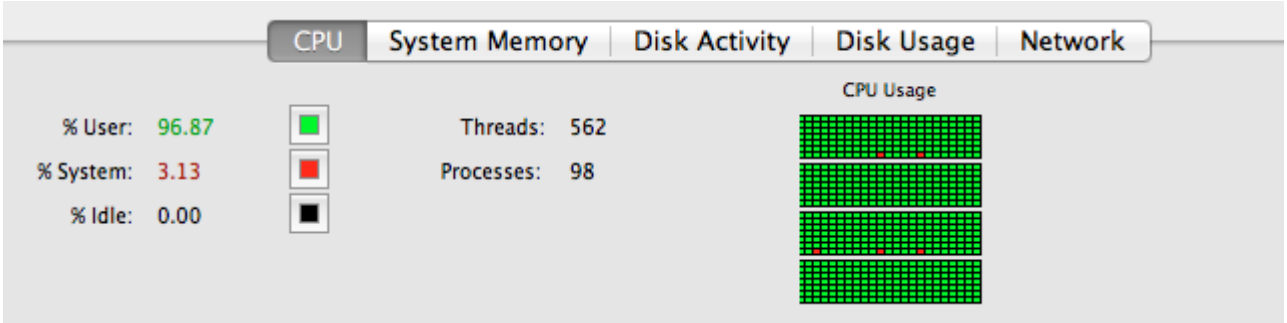
Enter the GPU. My own experiment differed in a more dramatic way from the practices of most serious password crackers, since I relied wholly on the CPU. But the best password cracking programs today can perform calculations on a GPU as well, generally at far higher speeds than CPU-based cracking alone. (Hashcat has two variants designed solely for use on computers with an OpenCL or CUDA-compatible GPU.) Those who really want to blast through the hashes (and raise the ambient temperature of the room around them) rely on GPU-based cracking.

Better cracking techniques. My own experiment touched only on two simple attacks, "straight" wordlist attacks (sometimes using rules to generate additional words) and "brute force" attacks. But other attacks exist, including "combination" and "permutation" attacks. Attacks can even skip the processor-intensive step of generating new hashes for each item in a wordlist and simply use a massive precomputed table of hashes, often many gigabytes in size, to reduce password cracking to essentially a database lookup exercise mixed with some fancy math. (Hashcat does not support such "rainbow table" attacks; to understand more about how they work, see our primer.)

The attacks I used can also be run more efficiently. Markov chains, for instance, rely on some statistical number-crunching to determine the odds that a given character will appear after another character in a password, using probability to make brute force attacks much faster.

Slower

Better hashing algorithms. Those charged with securing passwords have many ways to fight back against this cunning onslaught, however. First up, they can use better algorithms. I chose to start with a set of unsalted MD5 hashes because the MD5 hash function was built for speed—which means that basic MD5 hashes are also easy to crack (and are generally not recommended for passwords any longer by security professionals). Plenty of other "speedy" algorithms offer better protection, including the far more secure bcrypt or scrypt. (The amount of time needed to crack hashes increases dramatically as the speed of the hashing algorithm slows.)



Hashcat will peg every processor core you give it.

Nate Anderson

More iterations. Another common way to slow down password crackers relies on running a hash through the hashing algorithms multiple times. This can slow down the cracking of even something like MD5, and it can be done as many times as desired. Running a password through SHA256, for instance, and then feeding the hashed output back through the algorithm another 500 times would certainly make wordlist and brute force attacks slower by requiring every attempted crack to follow the same time-consuming set of iterations.

Finally, more iterations alone won't stop table lookup attacks. Though the tables will take much longer to generate, once they're generated, attacks that use them will proceed at the same speed as with any other table lookup attack. Which brings us to...

Salting. Salting attempts to defeat table lookup attacks by adding several characters to the password before passing it through the hashing algorithm, and these characters don't have to remain secret.

Imagine, for instance a website that secures millions of passwords with a single iteration of SHA256. Imagine that 25 percent of them use the same weak password of "123456." Unfortunately for the hacker (and fortunately for the website and its users), each password will have to be cracked individually, even though 250,000 of them are the same. Because the website appended a unique, randomly generated "salt" such as "0hJ3l1" to each password, the resulting hash will contain a different value. In addition to preventing large numbers of passwords from being cracked all at once, salting also thwarts cracking techniques that rely on rainbow tables.

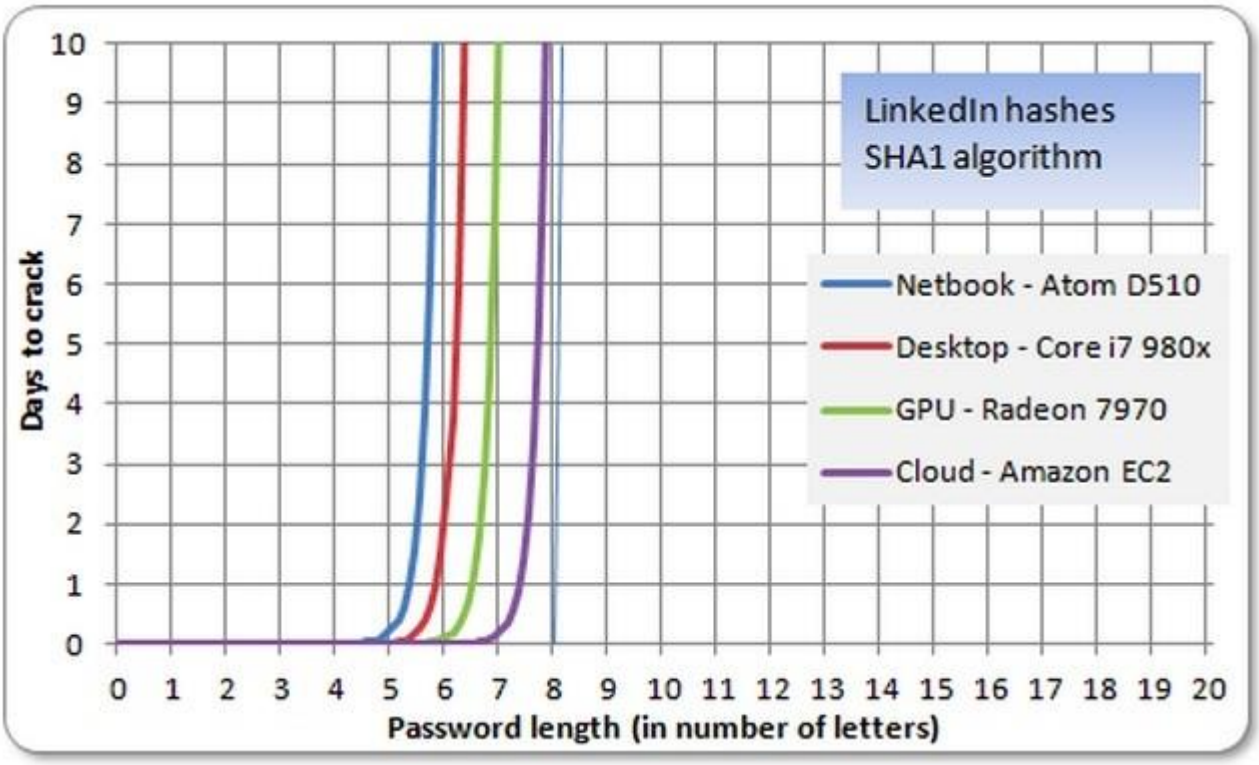
Because salt is often not hidden, but stored right with the hashed passwords, it may provide no defense against traditional wordlist attacks—which is why password security often depends on a combination of all of these techniques.

Password length. Finally, passwords can get longer. This can have a dramatic effect on password cracking times, as noted above, especially if the additional characters are random ("123" hardly counts).

As Robert David Graham puts it, "The thing that most people don't understand about passwords is that brute-force is an exponential problem. The amount of time it takes quickly grows out of all reasonableness.

"People have the misconception that massive increases in performance lead to massive differences in password cracking, but it doesn't really. Moving from my desktop processor to a GPU that's 20 times faster only slightly increases the length of password I'm able to brute-force. Even going to a 1,000 instance Amazon EC2 cluster with super-computer performance doesn't dramatically increase password lengths that I'm able to crack."

His post includes the following graph to show just how quickly additional characters can make brute-forcing impossible.



Additional password characters can thwart even massive computing resources.

Robert David Graham

Turning the weapon on myself

The unsettling experience of cracking so many passwords so easily led me to one final question: how secure were my own key passwords? Opening Terminal in OS X, I created an MD5 hash of my computer account password, which I had not changed in many years:

```
echo -n "password_here" | md5
```

I pasted the resulting hash into a text file and ran it through Hashcat, using the RockYou dictionary and the best64.rule variations. It broke in under a second.

A quick investigation showed just how it had failed—my password was actually in the RockYou dictionary—but the news wasn't *that* bad. For one thing, my hash had not been leaked publicly, and retrieving it would require someone to break into my computer. Even that was more challenging than it used to be because OS X Lion, which I'm still running, moved password hashes into a protected directory only accessible to "root," but it kept the root account off by default (attempts to access the directory with "sudo" or "su" therefore fail unless root is actively switched on by the user). Beyond that, OS X hashes passwords using SHA512 and adds a salt to prevent simple table lookup attacks.

Still, "All hashes have been recovered" isn't the sort of thing one wants to see after running a password cracker on one's own password. While passwords can be made more complicated without getting longer—mixing case or adding symbols or resorting to pure randomness—adding random characters is the easiest way to protect a password. I added several more characters to my password, made sure the result wasn't in the RockYou dictionary, then ran it against the RockYou + best64.rule combo to make sure that it was safe from the most common permutations. It was.

I next checked my online banking passwords using the same system; none were easily cracked. I remain under no illusions that beating a RockYou wordlist/best64.rule attack means I am secure—but any password that protects financial information or other sensitive data should at *least* pass such a test.

After my day-long experiment, I remain unsettled. Password cracking is simply too easy, the tools too sophisticated, the CPUs and GPUs too powerful for me to believe that my own basic attempts at beefing up my passwords are a long-term solution. I've resisted password managers in the past over concerns about storing data in the cloud or about the hassle of syncing with other computers or about accessing passwords from a mobile device or because dropping $50 bucks never felt quite worth it—hacks only happen to other people, right?

But until other forms of authentication take root, the humble password will form a primary defense of our personal information. The time has come for me to find a better solution to generating, storing, and handling them. (Look for future features on the subject in the near future.)

Password cracking proved surprisingly addictive—it's the ultimate mathematical puzzle, a lock that can be picked with only a single precise key that you have to uncover from a pile of billions of similar keys. Finding the fastest way to sort this pile is the game, and it takes only hours to learn. But mastering it is the real challenge, and you don't need to be a dark-hearted black hat to fall for it. After all, what true puzzle lover would be content with cracking only half the hashes?

http://arstechnica.com/security/2013/03/how-i-became-a-password-cracker/