;login:

THE USENIX MAGAZINE

December 2003 • volume 28 • number 6

Focus Issue: Security

Guest Editor: Rik Farrow

inside:

USENIX

The Advanced Computing Systems Association

the end of crypt() passwords… please?

by Tom Perrine

Tom Perrine is the Infrastructure Manager for Sony's Playstation Product Development organization, where he watches over online games, software developers, artists, and musicians. In former lives he has been a computer security researcher, system administrator, and operating system developer for universities, companies, and government agencies.

tep@scea.com

Introduction

In the security realm, there is a tendency on the part of of some system administrators and many vendors to ignore "theoretical" vulnerabilities. Inertia is one of the driving forces behind the "full disclosure" movement, a group that believes the only way to get a vendor (or anyone else) to fix something is to release working exploits, so that vulnerabilities cannot be dismissed as theoretical and therefore safe to ignore. It is a recurring issue that it is often difficult to convince vendors (and some system administrators) that vulnerabilities are "real" before exploits appear in running code.

For example, for over 30 years the UNIX password protection system has depended on the UNIX crypt() function to protect 8-character ASCII passwords. Assumptions about computational and storage resources that were made 30 years ago are no longer valid. The UNIX crypt() function has been overtaken by computing technology and should no longer be relied on for password protection.

Although some UNIX and all Linux vendors now offer alternatives, such as longer passwords and stronger hash functions, crypt() passwords are still often required for compatibility with existing account management software or in multi-vendor environments where not all systems support those alternatives. Stronger systems are the norm in the open source operating system community, with Linux and the BSDs all supporting stronger hash functions such as MD5. However, there are still many commercial UNIX variants where crypt() is either the only option or the only option with full support from the vendor, or where using MD5 is incompatible with layered software for that platform.

Until now, quantifying the risks of continuing to depend on this outdated function has been via "back of the envelope" calculations based on key space size and theoretical performance figures for various hardware platforms. The evidence of Moore's Law and academic analysis of cryptographic algorithms have been insufficient to push vendors into supporting stronger systems by default. One of the goals of this project is to drive the last nail into the coffin of the UNIX crypt() function for 8-character passwords, using real-world results. Hopefully, the persuasive power of weaknesses demonstrated by running code and a large database of precomputed password hashes may accomplish that.

Toward this end, over the past few years system administrators and security practitioners at the San Diego Supercomputer Center (SDSC) have investigated the security of the crypt() function in light of ever-increasing computing and storage capabilities.

Our recent paper[1] describes the most recent large-scale password cracking project undertaken at the SDSC. This project examined the use of teraflop computing and petabyte storage capabilities to attack the traditional UNIX crypt() password system.

The paper presents results from applying high-performance computing (HPC) resources such as a parallel supercomputer, abundant disk, and a large tape archive systems, to precompute and store crypt()-based passwords that would be found using

1. Tom Perrine and Devin Kowatch, "Teracrack: Password Cracking Using Teraflop and Petabyte Resources," SDSC, 2003, *http://security.sdsc.edu/publications/teracrack.pdf* and *http://security.sdsc.edu/publications/teracrack.ps*.

common password-cracking tools. Using the Blue Horizon supercomputer at SDSC, we found that precomputing the 207 billion hashes for over 50 million passwords can be done in about 80 minutes. Further, this result shows that for about $10,000 anyone should be able to do the same in a few months using one uniprocessor machine.

This article provides a summary of that work, focusing on the results and implications instead of the technology. Full details of the computing hardware, software, and storage resources are available in the paper.

Project History, Motivation, and Goals

There have been two major password-cracking projects at SDSC. The first, Tablecrack, was intended to quickly determine which UNIX accounts, if any, had passwords that were easily guessed. Tablecrack was used for several years at SDSC to identify vulnerable passwords.

During the use of Tablecrack, it became apparent that there were other interesting questions to investigate, some of which are discussed below. The current project, Teracrack, explores some of these questions, as well as takes advantage of the advances in computing that have occurred since the original Tablecrack project began in December 1997.

Both Tablecrack and Teracrack exploit a novel time/space trade-off to take advantage of very large data storage capabilities, such as multi-terabyte disk systems, on password cracking.

The last and most recent goal of Teracrack was to pursue a "world land-speed record" for password cracking, combining multi-teraflop computing, gigabit networks, and multi-terabyte file systems.

For the purposes of this project, easily guessed passwords are defined as those in a specific list, which is generated using common, publicly available methods (e.g., Alec Muffet's Crack)[2] from publicly available word lists (dictionaries).

The effort in this specific dictionary attack is *not* an exhaustive search of all possible eight-character passwords. For our purposes, it is not necessary to try all passwords, just all those that are likely to be tried by an attacker using commonly available software. In real life, this set of passwords is defined by the software likely to be used by an attacker, e.g., Crack 5.0a or perhaps John the Ripper.[3] By testing the user's password against the set of guesses likely to be tried by this software, we can make a reasonable determination about the user's password falling to an intruder's attack.

The Time/Space Trade-Off

A novel part of both Tablecrack and Teracrack is the reversing of the time/space trade-off. Traditionally, it has been considered infeasible to precalculate (and store) all possible (or reasonable) passwords, forcing the attacker to generate (test) passwords on demand.

In fact, at least one earlier paper on password cracking[4] did discuss implementations of precomputed passwords. However, given the hardware (DEC 3100 CPUs and 8mm tape) of 1989, it took several CPU-weeks to produce hashes on 8mm digital tape for only 107,000 passwords, and it took several hours to check those tapes when searching for hashes. Clearly, it would have been impractical to try to store the hashes for millions of passwords, given disk and tape technologies available at the time.

2. "Crack, a Sensible Password-Checker for UNIX," *http://www.users.dircon.co.uk/~crypto/download/c50-faq.html.*

3. John the Ripper, *http://www.openwall.com/john/.*

4. D.C. Feldmeier and P.R. Karn, "UNIX Password Security — Ten Years Later," Proceedings of Crypto '89, in *Lecture Notes in Computer Science*, vol. 435, pp. 44–63.

It is obvious that the original UNIX crypt() is completely obsolete.

These original assumptions live on in most password-cracking software, including Crack and John the Ripper. These systems assume that there is limited (disk) storage and that each password hash should be calculated from a candidate password, checked against the actual target hash, and then discarded; the computed hashes are not saved for reuse.

Results and Conclusions

First, it is obvious that the original UNIX crypt() is completely obsolete. In today's computing environment, it should certainly not be the default password hash algorithm. It could be strongly argued that the algorithm should not be available at all, that only MD5 or stronger algorithms should be used.

WHAT ARE THE COMPUTATIONAL AND STORAGE REQUIREMENTS FOR SUCH ATTACKS?

We have shown that using the resources present at SDSC it is possible to precompute and save the 207 billion hashes from over 50 million of the most common passwords in about 80 minutes. This means that all hashes for the interesting passwords for a single salt can be computed in about 20 minutes. Further, on a per-CPU basis, the Power3 CPUs used in Blue Horizon are by no means the fastest available. In particular, the Intel Celeron provided very good numbers in our crypt timings. If, as we believe, the numbers above are related to the actual performance of Teracrack, then a modern x86 (1-2GHz) should be able to hash our word list for one salt in far less than 20 minutes.[5]

5. A test run indicates that this is the case.

To save 207 billion hashes requires 1.5 terabytes of storage. We had this much storage in a network file system, connected over high-speed network links. We also have a multi-petabyte tape archive to provide long-term storage for the hashes. The I/O bandwidth required for a full 128-node run is an average of 80 megabytes per second; however, in that time a single process only averages 80-82 kilobytes per second. Further, running the post-processing requires 2.27 terabytes of storage, and the resultant .pwh files will require 2.26 terabytes. Note that this space is not cumulative, but just represents different amounts at different times as the work is done.

These requirements are probably out of the reach of typical machines used by a single attacker. We were trying to exploit the resources available to us, and thus were trying to make this run in as little real time as we could. There are other methods we could have used that would have been less demanding. A more patient attacker could use fewer resources and still launch a successful attack in a reasonable amount of time, such as a few weeks.

ARE LARGE-SCALE DICTIONARY ATTACKS FEASIBLE FOR ATTACKERS WITHOUT ACCESS TO HIGH-PERFORMANCE COMPUTING RESOURCES? IS A DISTRIBUTED (COOPERATIVE) EFFORT FEASIBLE?

The computational requirements for precomputing all the hashes are high, but not prohibitively so. For a single Power3 CPU, the bulk encryption phase should take about 60 days. It should take a single CPU on a Sun-Fire 15K about 13 days to do the post-processing. Ignoring the storage and I/O bandwidth issue, the time required for bulk encryption and post-processing phases should decrease linearly for every CPU added to it.

The storage requirements are also "reasonable." With the current availability of IDE disk drives larger than 200GB, perhaps in IDE RAID arrays, the storage requirements

are also accessible. In fact, SDSC has experimented with low-cost "terabyte-class" file servers.[6] We have recently (November and December 2002) purchased 1.2-terabyte PC-based RAID file servers for around $5,000. We have been quoted prices below $10,000 for 2.8-terabyte file servers.

The I/O bandwidth should not be a problem for the smaller machines which are more likely to be used by an attacker. Even a dual-processor 1.5GHz x86 machine would generate less than 2MBps of output. If the machine only writes to local disk, there will be no network-bandwidth problems.

There are also several strategies for coping with the total amount of storage space required. Most involve either distributing the computation or making a space/time trade-off.

- By distributing the computation across several machines, the storage space required on each machine would be greatly reduced. It would also allow using existing hardware, with the addition of a large IDE hard drive. For example, a cooperative effort with eight machines could add a 200GB IDE hard drive to each machine. This would provide 1.6TB of storage, which is enough to store the output of the bulk encryption phase. Using the next technique, it will be enough to handle the post-processing phase as well.
- The post-processed output requires 50% more storage space than the raw hashes. This is only because the output also contains a reverse pointer that allows retrieving the password as part of the table lookup. Having this pointer is not necessary, even for an attacker who is trying to recover the password. The post-processing is still needed to sort the hashes (and speed lookup times), but the final space requirement will be 1.5TB instead of 2.26TB if the reverse pointer is left out. Instead of using the precomputed hashes to retrieve passwords, the attacker can use them to figure out which passwords can be recovered with minimal effort. Here the attacker uses the precomputed hashes on a stolen password file to determine which hashes are in the attacker's dictionary. Once it has been determined which passwords will be found, it will take 20 minutes or less per salt to recover the password. As it only takes one password to compromise an account, a single 20-minute run should suffice. Using this method, however, an attacker can still recover over 30 passwords in a single day. The savings is from not wasting time attacking strong passwords.
- We did not investigate compression at all. However, in Tablecrack, it was found that an algorithm like the one used by Crack to compress dictionaries showed promise.
- While over a terabyte of disk storage is still expensive, 200GB is very affordable, and enough space to store precomputed hashes for 400 salts. While having 400 salts is not as good as having all of them, all it takes is one broken password.

Thus, it seems safe to say that large-scale dictionary attacks are feasible for either a very patient single attacker, an attacker with a farm of compromised machines, or a collective of cooperating attackers. What we were able to do in hours, a network of attackers could easily do in days.

ARE THERE COLLISIONS (MULTIPLE PASSWORDS THAT PRODUCE THE SAME HASH) IN THE PASSWORD SPACE?

We found two types of collisions. First, there were some words in our dictionary which contained characters with the high-bit set. There were 24 such collisions in each case

6. SDSC, "A Low-Cost Terabyte File Server," *https://staff.sdsc.edu/its/terafile/*.

Even a small set of . . . attackers, or an attacker with moderate resources, could independently duplicate our work.

the two colliding words only differed in one character. Also in each case the differing characters were the same in the lower seven bits. These collisions are due to the way crypt() makes a key from the password, by stripping off the high bit, and concatenating the lower seven bits of each byte to form a 56-bit key. The lesson from these collisions is that there is no benefit from including characters which use the high bit in a user password, even if your version of UNIX supports this.

Second, we found one "real" collision. By this we mean two words that differ in more than just the lower seven bits of each byte, which hash to the same value. This occurs with the words $C4U1N3R and SEEKETH, under the salt 1/. Both words hash to ChER-hgHoo1o. The lesson from this is that although there are collisions in the crypt algorithm, and they do reduce the usable key space, they are relatively rare and this is likely not a real-world concern. Quantifying the exact number of collisions would require a dictionary equal to the key size.

WHAT ARE THE NONTECHNICAL (SOCIAL, ETHICAL, AND LEGAL) ISSUES INVOLVED IN MAKING THE RESULTS OF THIS PROJECT PUBLICLY AVAILABLE?
This question has been at the heart of both the Tablecrack and Teracrack projects and was not adequately addressed by either.

With Tablecrack we examined the issue and made a decision to only store password hashes, and not include the "back pointers" to the original passwords. This decision was mostly to address storage concerns, but also made it at least slightly more difficult for attackers, even if they had access to our password hashes. At that time, we considered that the most likely misuse of the Tablecrack data would be by an insider, due to the difficulty in moving the large data sets outside of SDSC. We expected that even if the attacker could determine sets of crackable passwords, we would have a good chance of detecting the resulting attacks on the identified vulnerable passwords.

While designing Teracrack, we realized that data storage and CPU performance had advanced to the point that even a small set of cooperating attackers, or an attacker with moderate resources, could independently duplicate our work in days or weeks. This changes the issues considerably.

We have investigated several ways to make our results available for system administrators to check their own password hashes for weak passwords. None of the ways is completely satisfactory, for various reasons:

1. We started by looking at providing a Web interface, such as a search engine: submit a Web form with a UNIX password hash and we could tell you whether or not the hash was from a weak password. This has problems in terms of back-end lookup performance, online storage, and our complete inability to prevent this system from being abused by an attacker. Such an interface, if it existed, would quickly succumb to the dreaded "slashdot effect" and become useless.
2. It was suggested that we could improve the Web interface idea by occasionally returning "weak" for a strong password. This would cause any attacker to occasionally waste CPU time trying to crack an "un-crackable" password. For the legitimate user, we would occasionally influence them to change a password that was not weak. It can be argued that it is never a bad idea to influence a user to change a password, but this is only true if they don't replace a strong password with a weak one.
3. We then decided that the problem was authenticating the user of any system we might build. We decided that we could probably find a way to manually vet users,

registering people whom we could identify and decide we trusted as users of our system. Although we could generate a list of well-known individuals, and individuals who were personally known to us, this obviously does not scale. If this is still accessed via a Web server, the problem of ad hoc query performance remains.

4. Our last thought experiment combines the ideas of user registration and bulk lookups. We could register the PGP keys of people we trust not to abuse the system. These people could send formatted email messages, signed with the registered keys. We could batch all the requests from all the users in each 24-hour period into a subset of lookups. This would have several advantages. It would allow us to batch queries by salt, so that we would have to make only one pass through each salt's file. Additionally, if the hash files were stored in HPSS, we would only need to retrieve the files of the salts that were in at least one query. With fewer than a thousand or so queries in a batch, it is likely that we would not need to retrieve more than half of the per-salt files.

Unfortunately, this system still suffers from problems of scale in handling user subscriptions, problems of policy in determining who may use the system, and the cost of actually operating such a system.

In the end, it is not clear how we can make the resulting data publicly accessible. Even if we can satisfy our own concerns, there would be liability issues if it could be shown that our system was used by an attacker to mount a successful attack against someone's weak passwords.

Yet all of this may be moot, as we have shown that this work can be recreated by a determined, patient attacker.

For ourselves, using a batch mechanism to submit the information from SDSC's password files will allow us to find weak passwords in a timely fashion, until we have completely eliminated the use of crypt() passwords on all our systems.

Future Work

There are six main areas in which we would like to pursue this project further:

1. Performance tuning for better scalability. We would like to attempt a few methods for reducing or eliminating the runtime connection to the number of nodes. Also, there are performance tweaks which show promise but were nonfunctional at publication time. Some of these include asynchronous I/O and dividing the word list rather than the salts.
2. Moving the software to emerging hardware platforms. SDSC is currently installing a new system that may offer up to 34 teraflops, with scaler integer performance at least 30 times that of Blue Horizon. This system will have very different cache, main memory, and network and I/O performance.
3. Public access to check for weak passwords. We would like to allow subscribers to check their site's passwords against our precomputed hashes. Thus subscribers could verify that they have no passwords in the Crack dictionary, without needing to invest in the resources to run Crack. Even though we have identified some of the problems above, there should be some way to make this service available for legitimate use.
4. Measurement of different-sized word lists. We would like to try measuring runtimes for word lists which are both larger and smaller. The larger word list would

likely come from adding foreign-language dictionaries to the initial dictionary. The smaller word lists would simply be subsets of our current word list.

5. Algorithms other than the traditional DES-based UNIX crypt(). We would like to try precomputing an effective number of passwords for other algorithms, including SHA-1 and MD5. Precomputing the Microsoft "LANMAN" hashes would be particularly easy, as the passwords are limited to uppercase and the hash is not salted. This would effectively be a massively parallel "L0phtCrack."
6. Investigating crypt() performance on x86. The x86 architecture seems to run crypt() very quickly, but just how quickly depends on a variety of factors. We would like to examine factors such as cache and CPU core versions. However, since we are arguing that crypt() should be eliminated, we should actually focus on the performance of MD5-based hashes instead.

Acknowledgments

This article is based on the Teracrack paper, which would not exist without the significant Teracrack development work by Devin Kowatch, and the contributions by the system administration staff at SDSC, especially Jeff Makey, the creator of "Tablecrack."

Availability

The code used for this paper is publicly available at *http://security.sdsc.edu/software/teracrack*. It is covered under the U.C. Software License, which allows source code access and is free for noncommercial use.