

Ramifications of Strict Security Policies on Password-Authenticated Computer Systems

Andrew Seely

University of Maryland University College, European Division

18 January 2003

Abstract

It is widely understood that the least secure aspect of any computer system is the user of that system. The security community's success in reacting to and mitigating computer exploits has been partly technical and partly due to the development of sound policy. Good policy mitigates many of the user-oriented risks that technology simply cannot address. This paper demonstrates a negative relationship between policy and system security, showing that a policy considered to provide the greatest security in fact may effectively reduce the difficulty of a technology-based attack. Specifically, this paper shows how a strict user policy regarding passwords significantly reduces the time needed to successfully brute-force attack password encryption.

## Ramifications of Strict Security Policies on Password-Authenticated Computer Systems

### Introduction

Authorized access to modern computer systems is typically limited by a user account name that is commonly known and a password that is kept private. While new access control paradigms like biometrics (Kellner, 2001) and graphical interfaces (Paulson, 2002) are starting to change authentication rules in cutting edge environments, it remains that simple password-based challenge-response controls will be the standard for most computer systems for years to come. This common single point of access control makes the password an attractive target for anyone attempting to compromise a computer system quickly.

Recognizing this, security administrators attempt to protect the integrity of a password with several techniques. On modern computer systems, user passwords are always stored in an encrypted format, and this "ciphertext" is protected from user-level access by operating system level controls (Pfleeger, 1997). Policies like forcing users to change passwords at established intervals, preventing passwords from being reused, and requiring passwords to conform to a particular format add to operational security by making the password a moving target. (CERT, 2000)

These protection techniques help prevent the easiest exploits, such as guessing someone's password based on names of pets or birthdays of loved ones. Frequent

changes to passwords protect against passwords being observed. Operating system controls help to prevent more sophisticated attacks against ciphertext password storage. Taken together, all of these measures likely prevent a vast number of casual system compromises.

This paper will focus on the relationship between operational security policies and common password encryption techniques. Security administrators typically publish, at least within their authorized user community, password control policies. Operating system encryption techniques are almost always based on published encryption algorithms that have been rigorously tested for mathematical strength (Schneier, 1996). Passwords work well as an authentication technique because there are multiple layers of protection. The failure of the operating system to protect the ciphertext storage does not immediately lead to failure of the encryption algorithm. In other words, stolen ciphertext still needs to be cracked (Pfleeger, 1997).

Possession of ciphertext passwords and the encryption algorithm used is not considered a realistic compromise due to the large amount of time needed to attack an algorithm. Published encryption algorithms do not yield easily to mathematical shortcuts or exploits, so the only practical method of attack is the brute force method of trying every possible plain-text input to the algorithm to see if the output matches the stored ciphertext. Despite the increasing speed of modern computers, the magnitude of the problem prevents serial brute force attacks from being practical.

There is a noticeable decrease in the difficulty of cracking a password by brute force when strict security policies are in place. Assuming that the operating system protection of ciphertext has failed and the encryption algorithm is known, knowledge of password

policies can make the job of cracking passwords take less time. The remainder of this paper will demonstrate the time needed to discover a password through brute force both with and without policy controls, demonstrating the difference in time. Practical examples in Perl using the Data Encryption Standard (DES) method of password encryption on a standard SuSE Linux computer will be used, but this discussion is language, algorithm, and operating system independent.

### Passwords Defined

Passwords are made up of "plaintext," simple characters found on standard keyboards. The set of legal password characters is defined by the operating system. For this example a standard U.S. English keyboard character set is assumed, though the actual character set employed is irrelevant. U.S. English characters are easily divided into four groups: uppercase letters, lowercase letters, numbers, and "special" characters. The set of legal password characters for any given operating system is the keyspace,  $K$ . An example of  $K$  is shown in Figure 1.

$$\begin{array}{l}
 K = \{ \\
 \quad k_1 = \{ \text{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \} \\
 \quad k_2 = \{ \text{abcdefghijklmnopqrstuvwxyz} \} \\
 \quad k_3 = \{ \text{0123456789} \} \\
 \quad k_4 = \{ \text{~!@#\$%^&*()_+=-`[]\|}{';: "/. , < > ?} \} \\
 \}
 \end{array}$$

Figure 1. The set  $K$  of legal password characters.

The total number of members in  $K$  is  $K_{(\text{length})} = k_1(\text{length}) + k_2(\text{length}) + k_3(\text{length}) + k_4(\text{length})$ . In the example in Figure 1,  $K_{(\text{length})} = 26 + 26 + 10 + 32 = 94$ .

In the authentication process, the user's password  $p$  is presented to the encryption algorithm which creates ciphertext that is unique for that password. The password is the "key" to the encryption algorithm's results. For example, on a typical Linux computer using DES encryption the plaintext password  $p = \text{"lemon"}$  may result in a ciphertext result  $c = \text{"azZJFZhMnQg6Q"}$ . Figure 2 demonstrates creating password ciphertext with Perl.

```
#!/usr/bin/perl
$plaintext = "lemon";
$salt = "az";
$ciphertext = crypt($plaintext, $salt);
print "The plaintext is " . $plaintext . ".";
print "The ciphertext is " . $ciphertext . "\n";

> ./testcrypt.pl
> The plaintext is lemon.
> The ciphertext is azZJFZhMnQg6Q.
>
```

*Figure 2, Using Perl to create ciphertext.*

Notice how the code in Figure 2 made a function call to `crypt()`. When a user logs into a system, the authentication process follows this same principle. The user provides  $p$  and the operating system does an encryption operation to produce  $c_{\text{intermediate}}$ , which is then compared to the stored ciphertext  $c$ . If  $c_{\text{intermediate}}$  and  $c$  match then the user is authenticated. (Garfinkel & Spafford, 1996)

## The Brute Force Method of Attack

The process of brute force attack is to repeat this operation for every possible  $p$  in the set of all possible passwords  $P$ . A brute force attack is not currently a viable threat since the encryption operation takes finite yet significant time. In Figure 3, time measurements before and after show how much time `crypt()` requires on the test system.

```
#!/usr/bin/perl
$plaintext = "lemon";
$salt = "az";
$before = (times)[0];
$ciphertext = crypt($plaintext, $salt);
$after = (times)[0];
$showlong = $after - $before;
print "The plaintext is " . $plaintext . ".\n";
print "The ciphertext is " . $ciphertext . ".\n";
print "Time taken for crypt() was " .
      $showlong . " CPU seconds.\n";

> ./testcrypt.pl
> The plaintext is lemon.
> The ciphertext is azZJFZhMnQg6Q.
> Time taken for crypt() was 0.05 CPU seconds.
>
```

Figure 3, Using Perl with time measurements to create ciphertext.

In general terms, the process of brute force cracking requires some amount of time per  $p$  tested. This time consists of the generation of the next password to try,  $p_{\text{next}}$ , the encryption operation, and the comparison of  $c_{\text{intermediate}}$  to  $c$ . For purposes of this discussion, the time taken for one whole cycle of generate/encrypt/compare will be  $t$  and the total time to try every  $p$  in  $P$  will be  $T$ :  $T = tP$ .

A simple brute force attack on  $c$  may be completed as shown in Figure 4. For this discussion, it is assumed that  $p$  has a maximum length  $p_{\text{maxlength}}$  of eight characters.

```

#!/usr/bin/perl
$maxlen=8;
@characters = (a..z, A..Z, 0..9, '\\', '\\~', '\\!', '\\@', '\\#', '\\$',
 '\\%', '\\^', '\\&', '\\*', '\\(', '\\)', '\\-', '\\_', '\\+', '\\=', '\\[',
 '\\]', '\\{', '\\}', '\\:', '\\;', '\\"', '\\\'', '\\.', '\\/', '\\<',
 '\\>', '\\?', '\\\\', '\\|');
for ($x=1;$x<=$maxlen;$x++) {
  foreach $j1 (@characters) {
    if ($x == 1) {
      &testit($j1);
      next;
    }
    else {
      foreach $j2 (@characters) {
        if ($x == 2) {
          &testit($j1 . $j2);
          next;
        }
        else {
          foreach $j3 (@characters) {
            if ($x == 3) {
              &testit($j1 . $j2 . $j3);
              next;
            }
            else {
              foreach $j3 (@characters) {
                if ($x == 4) {
                  &testit($j1 . $j2 . $j3 . $j4);
                  next;
                }
                else {
                  foreach $j5 (@characters) {
                    if ($x == 5) {
                      &testit($j1 . $j2 . $j3 . $j4 . $j5);
                      next;
                    }
                    else {
                      foreach $j6 (@characters) {
                        if ($x == 6) {
                          &testit($j1 . $j2 . $j3 . $j4 . $j5 . $j6);
                          next;
                        }
                        else {
                          foreach $j7 (@characters) {
                            if ($x == 7) {
                              &testit($j1 . $j2 . $j3 . $j4 . $j5 . $j6 . $j7);
                              next;
                            }
                            else {
                              foreach $j8 (@characters) {
                                &testit($j1 . $j2 . $j3 . $j4 . $j5 . $j6 . $j7 . $j8);
                              }
                            }
                        }
                    }
                }
            }
          }
        }
      }
    }
  }
}
sub testit {
  my $testval = $_[0];
  $cypher = crypt($testval, 'az');
  if ($cypher eq "azZjFZhMnQg6Q") {
    print "The password is $testval!\n";
    exit 0;
  }
}
}

```

Figure 4, Simple brute force program in Perl.

The maximum quantity of  $p$  in  $P$  is the number of one to  $p_{\text{maxlength}}$  character permutations possible using the character set defined in  $K$ . Thus, the maximum quantity of  $P$  is  $\sum_1^n K^n$ , where  $n = p_{\text{maxlength}}$ . The total time required is  $T = tP = t \sum_1^n K^n$ . For the test system,  $n = 8$ ,  $K_{\text{length}} = 94$ , and by demonstration  $t$  takes 0.05 CPU seconds. On this system, the time required to compare every possible password to a given ciphertext would be

$$\begin{aligned}
 T &= tP \\
 &= t \sum_1^n K^n \\
 &= 0.05 \text{ CPU seconds} * (94^8 + 94^7 + 94^6 + 94^5 + 94^4 + 94^3 + 94^2 + 94^1) \\
 &= 0.05 \text{ CPU seconds} * (6,095,689,385,410,816 + 64,847,759,419,264 + \\
 &\quad 782,757,789,696 + 7,339,040,224 + 78,074,896 + \\
 &\quad 830,584 + 8,836 + 94) \\
 &= 0.05 \text{ CPU seconds} * 6,161,327,320,574,410 \\
 &= 308,066,366,028,721 \text{ CPU seconds.} \\
 &= 5,134,439,433,812 \text{ CPU minutes} \\
 &= 85,573,990,564 \text{ CPU hours} \\
 &= 3,565,582,940 \text{ CPU days} \\
 &= 9,768,720 \text{ CPU years}
 \end{aligned}$$

In this example,  $t = 0.05$ ,  $P = 6,161,327,320,574,410$ . While the solution for  $T$  seems dramatic, it is important to note that the actual value of  $t$  acts as a constant and is less significant to this formula than the value of  $P$ .

The above value of  $P$  assumes that no policy restrictions are in place. Since it may be assumed that  $p$  is chosen at random, on average  $p$  will be found in half the time (Moore & McCabe, 1993). Using the value of the previous value of  $T$ , we can say

$$T_{average} = \frac{tP}{2} = \frac{t \sum_1^n K^n}{2} = \frac{t6,161,327,320,574,410}{2} = t3,080,663,660,287,205 =$$

$0.05 * 3,080,663,660,287,205 = 154,033,183,014,360.25$  CPU seconds required, on average, to discover a plaintext password using brute force on a system without password policies.

### The Impact of Strong Password Policies

What impact, if any, does a security policy have on  $P$  and ultimately  $T$ ? A sound security policy may include the following rules (compiled from: Garfinkel & Spafford, 1993; Pfleeger 1997; SANS Institute, 2003):

1. Passwords must be eight characters long.
2. Passwords must not be in a multilanguage dictionary.
3. Passwords must contain at least one character from every character grouping.
4. Passwords must be changed at a regular interval and not be re-changed for some length of time.

For  $T$  to decrease, either the size of  $P$  or  $t$  must be reduced. The quantity of  $t$  will be treated as a constant since it is dependent on computer hardware. Policy rule number one

immediately changes the size of  $P$  from  $\sum_1^n K^n = 6,161,327,320,574,410$  to  $K^n =$

$6,095,689,385,410,816$ , reducing the number of operations required to search the

password space by approximately 1.07%. From a password cracking perspective this does not significantly reduce the amount of work to do, but it does make writing cracking software shorter and easier, as shown in Figure 5.

```
#!/usr/bin/perl
$maxlen=8;
@characters = (a..z, A..Z, 0..9, '\`', '\~', '\!', '\@', '\#', '\$',
'\%', '\^', '\&', '\*', '\(', '\)', '\-', '\_', '\+', '\=', '\[',
'\]', '\{', '\}', '\;', '\:', '\'', '\"', '\', '\.', '\\/', '\<',
'\>', '\?', '\\', \|');
for ($x=1;$x<=$maxlen;$x++) {
  foreach $j1 (@characters) {
    foreach $j2 (@characters) {
      foreach $j3 (@characters) {
        foreach $j3 (@characters) {
          foreach $j5 (@characters) {
            foreach $j6 (@characters) {
              foreach $j7 (@characters) {
                foreach $j8 (@characters) {
                  &testit($j1 . $j2 . $j3 . $j4 . $j5 . $j6 . $j7 . $j8);
                }}}}}}}}}
}
sub testit {
  my $testval = $_[0];
  $cypher = crypt($testval, 'az');
  if ($cypher eq "azZJFZhMnQg6Q") {
    print "The password is $testval!\n";
    exit 0;
  }
}
```

Figure 5, Modified brute force program in Perl.

The enforcement of rule three means that there will be no homogeneous passwords. Passwords that are exclusively made up of only  $k_1$ ,  $k_2$ ,  $k_3$ , or  $k_4$  are immediately excluded from membership in  $P$ , as are passwords made up of only two or three different categories of  $K$ . The set of passwords that are only made up of three categories of  $K$  will automatically include those made up of two or one categories, so determining a three-category exclusion  $P_{\text{exclusion}}$  will define a  $P$  that contains only passwords that include members from all four  $K$  categories.

The number of passwords made up purely of one, two, or three categories can be determined by combinations of three from the four categories, subtracting duplicates created along the way:

$$\begin{aligned}
 P_{\text{exclusion}} &= [(k_1 + k_2 + k_3)^n + (k_1 + k_2 + k_4)^n + (k_1 + k_3 + k_4)^n + (k_2 + k_3 + k_4)^n] - \\
 &\quad [(k_1 + k_2)^n + (k_1 + k_3)^n + (k_1 + k_4)^n + (k_2 + k_3)^n + (k_2 + k_4)^n + (k_3 + k_4)^n] + \\
 &\quad [(k_1)^n + (k_2)^n + (k_3)^n + (k_4)^n] \\
 &= [218,340,105,584,896 + 2,479,758,911,082,496 + \\
 &\quad 457,163,239,653,376 + 457,163,239,653,376] - \\
 &\quad [53,459,728,531,456 + 2,821,109,907,456 + 128,063,081,718,016 + \\
 &\quad 2,821,109,907,456 + 128,063,081,718,016 + 9,682,651,996,416] + \\
 &\quad [208,827,064,576 + 208,827,064,576 + 100,000,000 + \\
 &\quad 1,099,511,627,776] \\
 &= 3,612,425,495,974,144 - 324,910,763,778,816 + 1,517,265,756,928 \\
 &= 3,289,031,997,952,256
 \end{aligned}$$

The original  $P = \sum_1^n K^n = 6,161,327,320,574,410$ . Enforcement of rule one changed  $P$  to  $K^n$ , a reduction of 65,637,935,163,594 from the original. Enforcement of rule three reduced  $P$  by  $P_{\text{exclusion}} = 3,289,031,997,952,256$ . Thus, the effect of rules one and three are to reduce  $P$  from 6,161,327,320,574,410 to a  $P_{\text{restricted}}$  of 2,806,657,387,458,560, a reduction of approximately 55%. Given an attack algorithm that could differentiate between passwords in  $P$  and  $P_{\text{restricted}}$ , a more reasonable estimate of the security of a system's ciphertext passwords can be stated as  $P_{\text{restricted}} = K^n - P_{\text{exclusion}}$ .

Policy rule two is implied by rule three, since any word with numbers and special characters will not be in a standard dictionary. However, proactive security analysts may

construct a modified ‘cracking’ dictionary from  $P_{\text{restricted}}$  using common substitutions like  $i=1=I$  or  $o=0=O$ , etc. A reasonable multilanguage dictionary for cracking purposes may be assembled from the various single-language dictionaries available at Internet sites like The Center for Education and Research in Information Assurance and Security at Purdue University (<ftp://ftp.cerias.purdue.edu/pub/dict/dictionaries/>). Removing these passwords from  $P_{\text{restricted}}$  means they will not be tested in a crack situation, leaving a final equation of  $P_{\text{restricted}} = K^n - P_{\text{exclusion}} - \text{dictionary}$ .

$$\text{Revisiting the example with an unlimited } P: T_{\text{average}} = \frac{tP}{2} = \frac{t \sum_1^n K^n}{2} =$$

$$\frac{t6,161,327,320,574,410}{2} = t3,080,663,660,287,205 = 0.05 * 3,080,663,660,287,205 =$$

154,033,183,014,360.25 CPU seconds required, on average, to discover a plaintext password using brute force on a system without password policies. Substituting  $P_{\text{restricted}}$

for  $P$ , and assuming no dictionary words have been prepared:  $T_{\text{average}} = \frac{tP}{2} = t(P_{\text{restricted}})/2 = 0.05 * 2,806,657,387,458,560 / 2 = 70,166,434,686,464$  CPU seconds. While this time is still far too long to be practical, manipulation of  $t$  through software optimization, multi-processing, and improved processor speeds could take advantage of this reduction of  $P$  and make brute-force cracking a reasonable activity in the near future.

### Conclusions

Potentially, the total time required to crack a password complying with known and strict policy is on average 55% less than if no policy at all is used. The challenge for the

attacker is to produce a function that returns the subset of  $P$  that is representative of the restricted password space without taking as much time to run as a regular encryption operation. Development of this function is left to the reader.

This risk, while relatively arcane, may be mitigated by security administrators and operating system designers by modification of the fourth policy rule to implement random policy enforcement. The problem with short passwords is not that they are more vulnerable by nature, but that a reasonable expectation of short passwords means that a cracker can launch a shorter and easier attack. In the same way, knowledge that no short passwords are allowed also leads to a shorter attack as described in this paper. If the attacker could make no assumptions about the size of content of a password, then a brute-

force attack would be forced to run at or near  $T = \frac{1}{2} \sum_{n=1}^{\infty} K^n$ .

Simply removing policy rules is not sufficient, as attackers could reasonably assume that users would consistently choose shorter, easier to remember passwords. The modified policy rule would require short, varied-component passwords randomly. Including a random change interval would add a sense of urgency to the cracking operation without creating any more annoyance to a user than already exists.

The proposed change to rule four would include these restrictions:

1. The length of a password will be between three and eight characters, with the actual length determined randomly at the determined password modification time.

This length will be enforced, i.e. the user will not be able to choose 'better' security by creating a longer password.

2. Content of a password will be chosen and forced from random combinations of  $K$  categories, with the combinations being between two, three, or four elements of  $K$ .
3. The time interval for mandatory password change will be randomized between ten days and 120 days. The time interval restricting password re-changing serves more to alert a user of a compromised password than to stymie an attacker, so it should be left unchanged.

This new rule would force crackers to

1. Search a maximum sized  $P$ .
2. Search on maximum permutations while losing on most dictionary attacks.
3. Not be able to assume that any hard-won password will be valid for any length of time.

Random policy enforcement may be beneficial to any system desiring high security. Simply hiding a security policy is not enough; though it may not hurt to keep it secret, security should not depend purely on the hope that no one guesses. As this study has shown, knowledge of a policy, even one designed to increase security, allows an attacker a foothold towards lessening the total amount of work that must be done to compromise a system. Creation of a policy that makes use of a random element retains the strengths of a good system and eliminates unhealthy reliance on obscurity while reducing the benefits published policy can provide to attackers.

References

- CERT (2000). Configure computers for user authentication. Retrieved 11 January 2003 from <http://www.cert.org/security-improvement/practices/p069.html>
- Garfinkel, S. & Spafford, G. (1996). Practical unix & internet security. Sebastapol, CA: O'Reilly & Associates, Inc.
- Kellner, M. A. (2001). Fingerprint ID devices are ready to make their mark. Retrieved 11 January 2003 from: [http://www.ibgweb.com/in\\_the\\_news/gcn.html](http://www.ibgweb.com/in_the_news/gcn.html).
- Moore, D. S., & McCabe, G. P. (1993). Introduction to the practice of statistics (2nd ed.). New York: W. H. Freeman and Company.
- Paulson, L. D. (2002). Taking a graphical approach to the password. *Computer* (35) 7. P. 19.
- Pfleeger, C. P. (1997). Security in computing. Upper Saddle River, NJ: Prentice-Hall.
- SANS Institute (2003). The SANS security policy project. Retrieved 11 January 2003 from: <http://www.sans.org/resources/policies/>.
- Schneier, B. (1996). Applied cryptography. New York: John Wiley & Sons, Inc.