IMPROVING THE DICEWARE MEMORABLE PASSPHRASE GENERATION SYSTEM

Marco Antônio Carnut (kiko@tempest.com.br)
Tempest Security Technologies

Evandro Curvelo Hora (evandro@tempest.com.br)
Universidade Federal de Sergipe – DCCE/UFS

ABSTRACT

This paper describes the Diceware method for making randomly chosen passwords easy to memorize and suggests several changes to the original scheme to make it even easier with little or no loss of security, along with a system to help users to recover from partial mistakes when recalling their passphrases. Two implementations in real world scenarios are discussed as well. It is argued that by using this method, ordinary dictionary attacks become uneffective.

1 INTRODUCTION

Passwords are still the primary mechanism most computer sytems employ to authenticate users; alternative methods like biometrics or tokens lag far behind in popularity and are often combined with passwords anyway. Over the years there has been significant evolution in password protection algorithms – in the Unix-like operating systems, the aging crypt(3) algorithm [1, 2] has been replaced by MD5Crypt [3], eksblowfish [4]; in Windows, the broken from the start LMHASH algorithm [5] has been replaced by Kerberos [6].

However, experience has shown that the users are the weakest link because they tend to choose easily guessed passwords. Several tools are available to perform guessing attacks on password databases, such as LMcrack [11], LC5 [12], John the Ripper [13], Crack [14]. System administrators have been trying to impose password restriction policies to try to foil those attacks, such as requiring that passwords satisfy certain minimum complexity rules (say, minimum of seven characters with at least one non-alphanumeric character and one numeric character) or requiring that passwords be changed often.

The effectiveness of these measures is debatable: the authors' practical experience with the aforementioned cracking tools has been that 20-30% of passwords in average can be recovered in a few hours when access to the encrypted password databases is gained; forcing users to change passwords too often also backfires: users tend to create simple password series such as "john03", "john04", and so on. Trying to force users to use too complex passwords often causes them to write them down in post-its stuck in their monitors.

This paper proposes a password suggestion system for helping users to create memorable random passphrases. It is an improvement of Arnold Reinhold's Diceware system [7]. The core of the idea is to use a dictionary to translate random numbers into common words user can more easily remember. Another particular characteristic of the Diceware method is that it is was originally meant to be made by hand, not by computers, using pencil, paper and common dice (such as the one used in board games or gambling). It is easy to underestimate how powerful this approach is to explain to non-technical users the importance of choosing their passwords randomly.

The rest of this paper is organized as follows. Section 2 reviews some terminology and attack scenarios and security design concerts for password systems. Section 3 presents the classical Diceware method and, out of discussing its shortcomings, proposes our modified version. Section 4 shows the method in action in two implementations, detailing the careful user interface details that help the user to memorize the passphrase both when choosing it for the first time and when using it to authenticate. This section also presents a proposed method to make the authentication correct small mistakes users typically make in their first attempts to recall a passphrase; it has been proven essential to foster our system's adoption. Section 5 wraps up presenting conclusions and ideas for follow-up work. The appendix shows our proposed Diceware dictionary.

2 PASSWORD SECURITY CRITERIA

2.1 Choice Sets

The larger the multitude of possibilities we draw our passphrase choice from, the less likely it is for an adversary to guess it and the larger the number of attempts she will have to make in a brute force attack.

The **choice set** is the set of all possible passphrases users can choose from in a specific computer system, program or usage context. For instance, the classical Unix login system accepts passphrases of at most eight characters that may be typed from a standard computer keyboard.

A fundamental operation is to compute or estimate the size of the choice sets. In this example, we assume for sake of simplicity there are about 96 typable characters (ASCII codes from 32 to 128 comprising all uppercase and lowercase letters, numbers, space and several punctuation signals). In this case, we would have $96^8 = 7.2 \times 10^{15}$ possible passphrases.

We will follow the common practice of assuming that users choose passphrases from the set equiprobably, at random. We know from experience that this is not true: users tend to choose common dictionary words, proper names, dates or phone numbers familiar to them. For all practical purposes, however, we can say that choosing nonrandomly from a large choice set *C*

(say, the acceptable Unix passphrase set with its almost 10^{16} elements) is equivalent to choosing randomly from a suitably chosen, much smaller set C' (say, the set of all dictionary words and proper names, estimated to have around 300,000 elements). The "familiarity" effect would be translated as a drastic reduction in choice set composition and size as much as to make the particular choice within that set look random to an outsider.

Our goal is to make the choice set so large as to render the attacker's efforts too time-consuming, while keeping its elements reasonably familiar to the users.

The size of choice set is often expressed as the number of binary bits needed to count all their elements; this is given by $\log_2 |C|$. Since we are assuming a random uniform distribution, this number is the same as the concept of **entropy** in information theory: it measures the "unpredictability" of the choice. Besides, this is convenient, as it avoids large numbers – for instance, the 7.2×10^{15} we calculated earlier become 57.2 bits of entropy.

2.2 Attacks

2.2.1. Computational attacks

Theoretically, it is always possible to discover which passphrase the user chose by means of the following algorithm, called the *exhaustive search* attack, or, more informally, the "brute force" attack:

```
algorithm exhaustive-search(C)
  for all p in C do
    if is_correct(p) then return p
  return NULL
```

That is, we try all possible plaintexts and until we find the one that tests to the value we know. The best case scenario is when the correct passphrase is the very first one attempted, when the algorithm finishes almost instantly. In the worst, case, however, it will take time $k|C|$, where k is the time it takes for the test algorithm to run once (and assuming it's constant, which, in practice, it usually is). This is why theoretical computer scientists say that this algorithm takes time proportional to the size of the key space, or, in other terms, exponential in the number of entropy bits.

The algorithm assumes that we know the choice set and that we have a function that tests whether a given attempt is correct or not. It is fair to assume that we always have access to the test function in the form of the process that actually tests the passphrase while authenticating to the desired service – be it the logon dialog in a window system, the PIN-entry screen in an ATM, the encryption function in a passphrase database (say, Unix's `/etc/passwd`, or `/etc/shadow`), or a network-based authentication protocol.

We assume that we can use the test function for as long and as many times as we want. This is not true in many scenarios: for instance, ATM PIN entry screens, certain system logon dialogs and a number of network protocols limit the number of attempts to just a few, like three or four, before locking out access for some time or requiring administrator intervention to reset the account. However, if we somehow obtain the passphrase database and the encryption function, our assumption holds – these are called *offline attacks*.

If the choice set we feed to the brute force algorithm is the same as the domain of the test function, the algorithm will always succeed if given enough time.

The special case where the choice set is a very small subset of the domain of the test function, say, based on a dictionary of common words, is often called the *dictionary attack*. It is still a "brute force" attack in the sense that it tries lots of possibilities, albeit far less than the exhaustive search. As any attack that doesn't try the full domain, it is not guaranteed to succeed.

Whether it's practical depends on several factors: how big the key/choice space being tested is; the time each attempt takes – the proportionality constant k – which might be improved by optimized implementations, better hardware or networking, etc. The realization of the brute force algorithm can vary immensely in sophistication, ranging in hardware from a single commodity PC to a massive parallel computer with dedicated ASICs implementing the test algorithm.

Resisting to offline attacks is much harder because it reduces the issue to a computational problem and the attacker can optimize the hardware to run the brute force algorithm at blazing speeds. In other words, the attacker isn't limited by any tricks the defender can put in place to make k deliberately large (say, imposing a delay between attempts or lock-outs) and can do her best to make k as small as possible by using better hardware and perhaps better algorithms as well.

2.2.2. Physical or environmental attacks

Passphrases can be captured when being typed, either when it's being chosen for the first time or when the user is being challenged to authenticate, using a variety of methods:

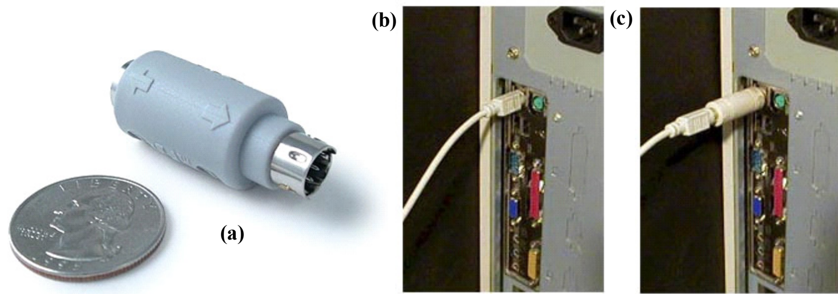


Figure 1: In (a), we see a miniaturized hardware keylogger with a quarter dollar coin for size comparison. In (b) we see the computer's keyboard cable before the device in installed and in (c) after. Since most of the time users don't care to look at their computer's cabling, such a device can pass unnoticed for long periods. It has 64KB of non-volatile memory that records all keystrokes coming from the keyboard to the computer. The attacker can later retrieve the device and examine the memory contents for passwords and similar sensitive data.

- **Hardware keyloggers:** usually a small device that connects between the computer and the keyboard, recording in non-volatile memory the keyboard data. The attacker has to have physical access to the computer to install and retrieve them, but this is a small hinderance. Besides, since current versions of the device are quite small (about the size of a mini-DIN connector) and they do not interfere with the normal working of the computer, they're almost imperceptible – most users really don't look at the cables at the back of their computers very often. Thus, most users won't find it unless if specifically told to look for them. Figure 1 shows one such setup.
- **Software keyloggers:** computer programs that attack to the hooks the main event loops in window systems or the kernel keyboard drivers. Although they usually are presented as standard executable files, it is often easy to trick the users into executing them by embedding them within another program or package. Quite a few viruses and worms also have integrated keyloggers as part of their payload. Most of them also record context information, such as the title of the window to which the specific keystrokes events were delivered. They can usually send the captured data to the attacker over the network.
- **Shoulder surfing:** the term usually brings up the image of a nosey snooper ostensibly looking over our shoulders while we type the password. For our purposes in this paper, we also include in this category less perceptible variants of the attack, such as being filmed by a hidden video camera strategically positioned to have the keyboard and possibly the screen as well within its field of view. With the miniaturization of video recording and transmission hardware, this attack has become a big deal in the banking industry's ATM machines.

3 THE DICEWARE METHODS

The Diceware passphrase generation method [7] was originally designed to be performed by hand using dice (the ordinary 6-sided ones from board games or gambling) as random number generation. A set of *n* ordinary cubic dice are tossed and their numbers are used to pick a word from a table, called the **dictionary**. This method offers a few attractive advantages:

- **Easy to understand and explain:** as we saw above, it can be explained in a few sentences and makes intuitive sense even to laymen. As can be seen in the appendix section in the end of this text, the full dictionary and the algorithm fit in a single page. The use of dice pedagogically stresses the importance of the randomness instead of a psychological choice.
- **Simple to implement in a computer:** arguably the hardest part is the pseudo-random number generator; however, there are several good ones widely available. The rest is a trivial table lookup.
- **Generates easy to remember passwords:** with a properly designed dictionary it is possible to "translate" the random numbers into common words, making them easier to memorize.

The recommended setup uses a 5 dice toss per word, yielding 6^5=7,776 possible values, which are also the number of words in the dictionary. Using the recommended 5 words, we get that the size of the choice set is $7{,}776^5$, yielding 64.62 bits of entropy.

However, the dictionary recommended by the Diceware authors comprises words with different word lengths, ranging from 1 to 6 characters per word; the exact word length distribution is shown in the table below. This means that the resulting passphrase may be anything from 5 to 30 characters in length (or 10 to 35 if the words are separated by spaces; for most of this paper we will ignore word separators in passphrase length calculations).

Word length	# of words
1	51
2	784
3	853
4	2346
5	3111
6	631

Thus, the passphrase length can be used to partially deduce its composition and reduce the amount of possibilities a exhaustive search attack would require. Let's take an extreme example to illustrate the issue: suppose the passphrase we want to attack has 5

characters. The only way this can happen is all words are single-character ones. Since there are only 51 such words in the dictionary, have have only 51^5=345,025,251 possible 5-character passwords or mere 28.36 bits of entropy – trivial prey for a brute-force attack. However, the probability that such a passphrase may actually happen (again, assuming unbiased dice with uniform distributions) would be quite rare: $51^5/6^{25}=1.2 \times 10^{-11}$.

Proceeding with the example, suppose now we have a 6-character passphrase. This implies it is comprised of one 2-character word and 4 single-character words. Unlike the previous case, we have a non-uniform composition: the 2-character word may happen anywhere, so there are in fact five possible permutations.

Given that there are 784 words with length 2 in the dictionary and, as we saw earlier, 51 single-character words, we have 5 possible permutations times 784 times 51^4, yielding 26,519,587,920 possibilites or 34.62 bits of entropy for a probability of 9.3×10^{-10}.

A *k,l*-partition of a positive integer *n* is a set of *k* strictly positive integers equal to or less than *l* whose sum is *n*. As we saw in the first example above, there is only one 5,6-partition of 5: {1,1,1,1,1} or 5=1+1+1+1+1. The second example showed that there is also only one 5,6-partition of 6: {2,1,1,1,1} or 6=2+1+1+1+1.

Now, seven-character passphrases now confront us with two possible 5,6-partitions: 2+2+1+1+1 or 3+1+1+1+1. So, for the first one we have 784 × 784 × 51 × 51 × 51, all that times 10 (this last value is the number of permutations of {2, 2, 1, 1, 1}), yielding 815,347,330,560. For the second, we have 853 × 51 × 51 × 51 × 51 times the 5 possible permutations of {3, 1, 1, 1, 1}, resulting in 28,853,582,265. Adding these two values we have 844,200,912,825 for the size of the subset of all possible seven-character, five-word diceware passphrases, corresponding to an entropy of 39.62 bits and a probability of 3×10^{-8}.

More generally, the entropy estimate for a diceware passphrase with lengh *l* and *w* words is:

$$E(l) = \log_2 \sum_{j=1}^{n_j} k_j \prod_{i=1}^{w} d(p_{i,j}(l,w))$$

Where $p_{i,j}(l,w)$ is the *i*th element of the *j*th *w,m*-partition of *l* with exactly *w* elements and with *m* being the maximum word length and *d*(*x*) being the number of words with length *x* in the dictionary. The quantity k_j is the number of permutations of the corresponding $p_{i,j}(l,w)$ and n_j is the number of *w,m*-partitions of *l* there are.

Using those equations we can compute the entropies and probabilities for all passphrase lengths from *m* to *wm*; in the classical diceware method, for passphrases ranging from 5 to 30 characters:

Length	Entropy	Leak	Prob %
5	28.36	36.26	1.21E-09
6	34.63	29.99	9.33E-08
7	39.62	25.00	2.97E-06
8	43.71	20.91	5.06E-05
9	47.03	17.59	0.0005
10	49.65	14.97	0.0031
11	51.78	12.84	0.0136
12	53.69	10.93	0.0513
13	55.33	9.29	0.159
14	56.76	7.86	0.429
15	58.01	6.61	1.023
16	59.08	5.54	2.145
17	59.99	4.63	4.023
18	60.73	3.89	6.739
19	61.30	3.32	10.016
20	61.71	2.91	13.254
21	61.93	2.69	15.447
22	61.94	2.68	15.58
23	61.73	2.89	13.452
24	61.25	3.37	9.616
25	60.39	4.23	5.305
26	59.04	5.58	2.091
27	57.13	7.49	0.553
28	54.54	10.08	0.0921
29	51.13	13.49	0.00867
30	46.51	18.11	0.000352

The above table reveals that with a probability of 15.6%, the most common passphrase length is 22 characters and it has 61.94 bits of entropy – 2.68 short of the ideal 64.62. In order words, the fact that we know the passphrase length "leaks" a certain amount of its entropy: around 3 to 6 bits for the most common passphrase lengths, but sometimes a lot more.

The diceware method FAQ suggests that if we ever get a passphrase with 14 characters or less, we should scrap it and choose another one. This prevents most large leaks while making just 0.65% of the choice set unavaliable. However, it does not handle the considerable leaks for 27 or more characters – which, also happening to cover about 0.65% of the keyspace, are not all that rare.

This is the first drawback of the diceware method with variable-length dictionaries: they have less entropy than it may appear. In the specific case of the 5-word passphrase with the 7,776-word Beale wordlist, it is at a minimum $2^{2.68} \approx 6.41$ times weaker than it could be if its wordlist was made of fixed-length words.

Another drawback comes from the fact that many words in the dictionary are not widely-known words in English, but numbers, symbols and abbreviations like "25%", "3000", "2nd", "5/8", "9:30", etc. This is probably due to the fact that selecting seven thousand words with six characters or less while striving to choose only common ones turns out to be more challenging than it may appear at first glance (as any Scrabble player can confirm), so the dictionary authors

```
testuser@testbox:~$ dwpwd
Current passphrase: oldstyle ❶

❷Choose: below we have 12 passphrase suggestions (more if you think of columns
------- instead of just rows). Choose one you find the easiest to memorize:

    1: pion mega roll duck week give      7: bald arcs gust luck tang muse
    2: knee deli eros lacs lack side      8: wave arak hold pigs grew grid
    3: wait year navy said best yale      9: pelt wild palm dogy shot into
    4: pill anti cows jerk that peas     10: void bulk tony anil says word
    5: mail tied cyan baby kiwi moon     11: ruse rush oval pant hall know
    6: warn look pays acne cove debt     12: deco that dots dive axis hush

* Tip: take your time. Don't try to choose in a hurry.
* Spicing: if you like, add small variations to the suggestion you chose.

    --- Press ENTER when ready to practice or ^N for more suggestions ---

❸Practice: Type the full passphrase you chose (not just the choice number)
--------- a few times to help you and your fingers to memorize it.
          At any time: ^Y goes up in the suggestion history, ^V goes down,
                       ^N generates new suggestions, ^D aborts.

New passphrase: wait year navy said best ale
** Nice. It does look like one of the suggestions above.

Please confirm: wait year navy said beat ale ❹
** Odd. You didn't type the same thing exactly. Let's start over, shall we?

                              --- STARTING OVER ---

New passphrase: wait year navy said best ale ❺
** Nice. It does look like one of the suggestions above.

Please confirm: wait year navy said best ale ❻
** Good. Let's try just once again to make sure, ok?

Confirm  again: wait year navy said best ale ❼
** Congrats. You typed it consistently right.
** Passphrase updated succesfully.
```

Figure 2: The `dwpwd` utility, our drop-in replacement for the classical Unix `passwd` utility, in action. System output is shown in normal font, echoed user input is in bold and non-echoed user input is in bold-italic. In (1), the user types the previous "old-style" password, as usual. In (2) we see the choice selection phase with the twelve randomly chosen suggestions. Several are shown so that a casual shoulder surfer can't trivially tell which one the user chose from. In (3) we see the the beginning of the practice phase with instructions for navigating the suggestion history and generating new suggestions. There we see the our test user choose a passphrase inspired by suggestion 3 (which by chance and with some interpretation freedom may be regarded as a meaningful sentence), but which a small variation or *spice*. Step (4) shows the user making a typo and having to start over. Steps (5) to (7) show the user carrying on to successful completion.

used those abbreviations to fill in the missing gaps up to the required 7,776 words.

The same reasoning reveals similar drawbacks in John Walker's JavaScript-based passphrase generator [8]: first, its 27,489-word dictionary has words from 1 to 8 characters. Computing the maximum entropy from the word length distribution yields 85.34 bits, 3.14 bits shy from the ideal 88.48 for the recommended 6-word passphrase. While 88.48 bits seems more than enough to resist offline attacks for the forseeable future, the average length of those passphrases is a whooping 39 characters – it is hard to convince users to type such long passphrases.

3.1 The Modified Diceware Method

We then introduce a few modifications to the diceware method to address its shortcomings while preserving its essence – in particular, its link with a real world physical process. The changes are as follows:

- **Smaller dictionary:** we use a wordlist with 6^4=1,296 word, yielding $\log_2 6^4 \approx 10.34$ entropy bits per word. The smaller number of words makes it a lot easier for dictionary designers to select more common, familiar words, not only in English but with other languages as well.
- **Fixed-length words**: as a result of the smaller dictionary, we could make all words have only four characters. This makes the full passphrases always 24 characters long, for a fixed entropy of 62.04 bits – slightly more than the best entropy of 61.94 from the classic method. The disadvantage is that passphrases are now two characters longer than the average classic ones.

The resulting regular structure of the passphrase brings other advantages. First, it helps memorization: when the user is in doubt whether how a particular word is spelled or whether it is in singular or plural form, he/she can count on choosing the version with four characters. Second, as the words are purely alphabetic, with no numbers or accents, it is easy to become proficient at typing them very quickly, helping to defeat shoulder surfers and providing a slight but non-negligible edge against filming.

The question of whether 62.04 bits of entropy is acceptable is certainly debatable. In [9], several expert cryptographers recommend 75 to 90 bits as the minimum size a cryptographic key should have to offer commercial-grade security. It is important to put this advice in perspective: they were referring to the raw keys used in low level encryption algorithms. Modern password-hashing algorithms include deliberate slowdowns to make brute force attacks harder; MD5Crypt, for instance, runs the passphrase through 1000 iterations of the MD5 algorithm. Most password-based encryption systems use a *key derivation function* (KDF) with many encryption

rounds both to "fill up" the low level encryption key to the maximum number of bits allowed by the specific algorithm; this is the case of the OpenSSL KDF used then encrypting RSA or DH private keys.

On the other hand, 62.04 bits is 9.3 bits (about 600 times) stronger than the 52.7 for 8-typable character password we calculated in section 2. We thus argue that 62.04 bits is a reasonable balance between the several conflicting goals we set out, with a explicit bias towards usability by a non-technical user base, even at the cost of computational security. There is no point in making the system extra-secure if the users don't buy it.

Extra security, however, can be easily attained by at least two methods: first, we can always add more Diceware words, accumulating entropy at a rate of 10.34 bits per word. Another is by **spicing** the passphrase: the user introduces small changes to the passphrase, like deleting, adding or replacing one or two characters, effectively sending it out of the set of diceware passphrases.

Even small changes add considerable entropy. Consider the simple case of deleting a random character: including spaces between words, the deleted character can be one in 29, or about 4.8 bits. Appending an alphanumeric character (either uppercase, lowercase or a number) to the end of the passphrase adds 5.8 bits of entropy, while inserting it at a random position adds about 4.8 bits for the uncertainty of the position, yielding 10.5 bits. So, in a strict sense, a 6-word diceware passphrase spiced by changing a single character is stronger than a 7-word diceware passphrase, however counterintuitive this may sound.

Of course, we know that users aren't good at making random choices. When asked to spice their passwords, most users do obvious things like changing "o"s (the letter "o") by "0"s (the number zero) or the like. So we usually recommend that the user spice their passphrases by deleting one character and changing another. If chosen randomly, that would yield 15.3 more entropy bits. In our empirical tests, we could often find the spiced version from the non-spiced in around 1000 "mini-brute-force" attempts, or slightly less than 10 bits of entropy.

It is worth to notice that if an attacker adopts the naive idea of specifying the choice set based on characters, his work becomes much larger: if he only considers uppercase characters, he would have 26^{24} possibilites or more than 112 bits of entropy.

4 APPLICATIONS AND IMPLEMENTATIONS

4.1 Application in a console environment

We wrote a program called `dwpwd` to be a version of the classical Unix `passwd` utility that uses our modified Diceware method for suggesting passphrases and test the user's reactions in a text-based console environment. Figure 2 shows an example for a `dwpwd` session. After verifying that the user knows the current password, it takes the user through two-phase process: choosing a passphrase suggestion and practicing his/her choice.

It is important to notice that the program actually accepts whatever passphrase the user types, even if the user's choice doesn't have anything to do with the suggestions, as long as it complies with the minimum quality restrictions set in the utility's configuration file.

For sake of brevity, figure 2 didn't show a few other tricks the program implements. For instance, it measures user's typing speed and if it is found to be over 15 characters per second, a message is shown saying that copying and pasting won't help memorization – after all, the program may be run from a terminal that supports clipboard operations. The program also prints a warning message if the CAPS LOCK key is on – now a common trait in graphical password dialogs but that we haven't seen before in console programs.

By default, the program clears the screen at the start of the practice phase – forcing the user to actually recall the passphrase from memory instead of just reading it onscreen and retyping it. All suggestions are held in a history buffer and the user can navigate up or down as desired; furthermore, a new set of suggestions can be generated at any time, so that the user can peruse them at will and choose a passphrase he/she deems easier to memorize.

4.2 Application in a graphical environment

Figure 3 shows a GUI version of the passphrase choice process in the scenario we originally intended: encrypting private keys stored in disk. The most noticeable difference is that the process is split in two windows: the first is a classical passphrase change window with a quality meter and a button to go to the second window with the password suggestion and training system. Thus the user adopts the diceware passphrases only if he/she wishes.

The suggestions window sports most of the features we've already discussed for the console version also work: the user is warned about the CAPS LOCK key, cheating by using copy-and-paste is frowned upon, the user can spice the suggestion at will or not follow them at all. A minor difference is that instead of clearing the screen, the suggestions become asterisks every time the user types a character in the practice area. The user can see them again by clicking on the appropriate toggle.

A confirmation message shows whether the user has typed the exact same password as in the previous attempt even before the user presses ENTER to

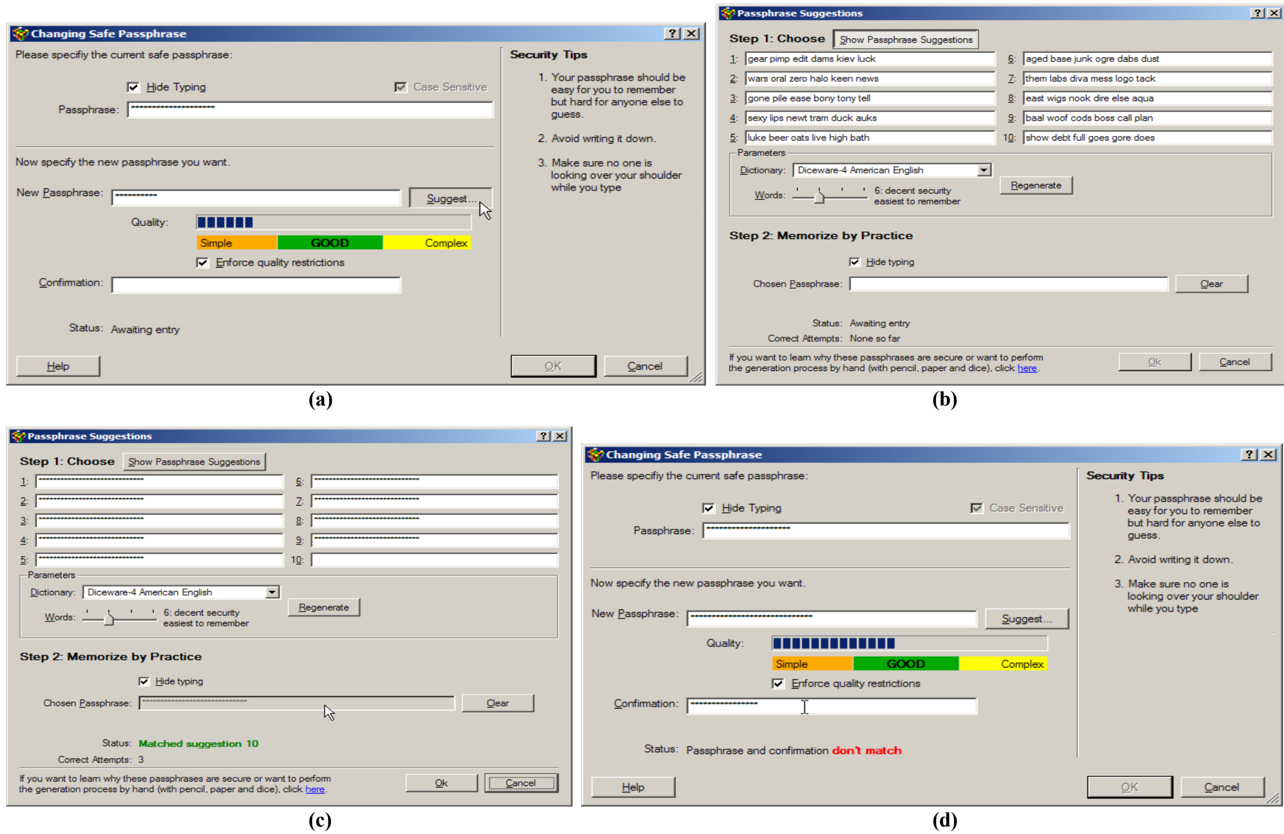


Figure 3: GUI version of the suggestion system when changing the private key protection passphrase in a PKI application. In (a) we see the user after he typed the current passphrase and clicking the button to get passphrase suggestions. The user could opt to not use the suggestion system at all, in which case he would be helped by the passphrase quality meter. Normally the meter will reject the passphrase if it is not within the "GOOD" range, but the user has the chance to override this restriciton as well by disabling in the "Enforce quality restrictions" checkbox. In (b) we see the suggestion dialog box displaying ten randomly drawn Diceware passphrases so that a casual shoulder surfer can't trivially tell which one the user is choosing from. Notice that the "Ok" button starts disabled. Whenever the user types a character in the practice area, the "Show Passphrase Suggestion" pushbutton is automatically released, causing all passphrases to be turned to asterisks and thus requiring the user to proceed from memory. The user can toggle the pushbutton at any time to peruse the suggestions some more, but the next character typed will release it again. In (c) we see the user typing the same passphrase correctly for the third time. The screenshot was captured right when the tenth suggestion text was blinking to indicate a (possibly inexact) match. At this moment the "Ok" button enables, allowing the user to go back to the previous window, shown in (d). The selected passphrase is copied to the "New Passphrase" text field, but the user still has to confirm it yet again. Notice that the Diceware passphrase easily scores good with the quality meter.

confirm. When the user confirms a passphrase by pressing ENTER and it is the same as the previous one, a "correct attempts" counter is incremented. Otherwise, the program assumes the user is confused and starts over, zeroing the counter. When the counter reaches a predetermined number of correct attempts (three, in our example), the "Ok" button is enabled (it starts disabled). At this point the user can continue practicing if desired, but clicking on the "Ok" button brings the program back to the previous window with the "New Passphrase" field already filled with the type passphrase. The usen then has to confirm it one last time in the "Confirmation" text field; a text message provides interactive feedback on whether the two passphrases match and whether the quality meter accepts or rejects the passphrase.

When the passphrase is finally validated, it is passed through the standard OpenSSL key derivation routine to transform it into an appropriately sized key to the symmetric encryption algorithm that will encrypt the private key. In this application we chose the Blowfish algorithm [10] because its deliberately expensive key schedule makes it one order of magnitude harder to brute force than 3DES (the OpenSSL default).

After that, a list of *degraded passphrases* is generated from the main passphrase, generated as follows:

- **Sorted passphrases with one word discarded:** The passphrase is split into w words (a word is considered any span of characters separated by spaces or puntcuation; accented characters are converted to their non-accented counterparts). The resulting words are sorted in alphabetical order. Then w passphrases with w-1 words are appended to the list by joining the words again with a space separator but each time dropping one of the words. Thus, if the main passphrase is "show debt full goes gore does", the following degraded passphrases would be inserted to the list:

```
does full goes gore show
debt full goes gore show
```

```
debt does goes gore show
debt does full gore show
debt does full goes show
debt does full goes gore
```

- **Passphrases with one character discarded:** Spaces and punctuation are removed from the passphrase, resulting in a string with length *l*. From that string, *l* degraded passphrases with length *l*-1 are inserted to the list by dropping one character at each iteration. For the same example passphrase above, our degraded passphrases would become:

```
oesdebtfullgoesgoreshow
desdebtfullgoesgoreshow
dosdebtfullgoesgoreshow
doedebtfullgoesgoreshow
         . . .
doesdebtfullgoesgoresow
doesdebtfullgoesgoreshw
doesdebtfullgoesgoresho
```

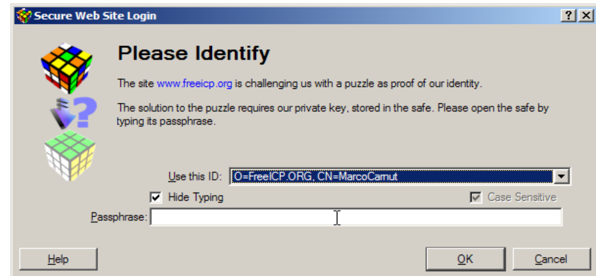
The list of degraded passphrases is not generated if the number of words in the main passphrase is less than six or if the number of characters is less than 20.

Each degraded passphrase in the list is appended with a 12-bit private salt (a random number, independently drawn for each degraded passphrase) and then sent to the OpenSSL key derivation routine to serve as encryption key to the Blowfish algorithm. The message being encrypted is the main non-degraded passphrase. The resulting block of encrypted messages are saved in a file along with means to associate it to that specific private key.

As the reader may be guessing by now, the purpose of the degraded passphrase file is to help the program to recover the full passphrase if the user later mistypes it in certain ways, but without at the same time facilitating the attacker's job. This was motivated by the fact that in early field testing, we noticed that many Diceware users often made small mistakes when using their passphrases for the first time (sometimes mere seconds after setting them up). The most common errors were:

- writing the words in the wrong order;
- forgetting one word;
- forgetting the exact spice (usually one character).

The picture below shows one of the dialog boxes that request the passphrase to unlock the private key (that particular example is for SSL client authentication [16] to a HTTPS-capable website).



When the user types the passphrase and confirms the dialog by either pressing ENTER or clicking on the "Ok" button, the program first tries to decrypt the private key using the supplied passphrase. If it decrypts correctly, it means that the user entered the exact password. In this case, a counter in the degraded passphrase file is incremented. If it reaches a specified limit (our implementation uses five), we assume that the user has sucessfully memorized that passphrase, so the program deletes the degraded passphrase file.

If the private key decryption fails and there is an associated encrypted degraded passphrase file, the program creates a list of *candidate* degraded passphrases the same way it created the list of degraded passphrases. Then it tries to decrypt each candidate with each degraded passphrase in the file.

Notice that each decryption attempt has to do a mini-brute force to find the correct salt among the 2^{12}=4,096 possible ones, since it is not stored anywhere. This explains the purpose of the salt: it compensates for the smaller entropy of the degraded passphrases by making the each decryption attempt 2,048 times harder on average. This idea is actually due to Manber [15].

If all the decryption attempts fail or there was no degraded passphrase file, the window waves horizontally to grab the user's attention and a message is shown telling the user that the passphrase is wrong. The user can then either try again or cancel the whole operation.

If, on the other hand, one of the decryption attempts is successful, the correct passphrase is recovered. The dialog box then shows the correct passphrase to the user. The user will have to type it again to get through (although we take care not to increment the success counter in this case). This way, we reinforce the correct passphrase in his memory.

5 CONCLUSIONS AND FUTURE WORK

We have described a set of improvements to the Diceware passphrase generation system to make it even easier to memorize, apply and use. We reasoned how our system with fixed words achieves nearly the same resistance to brute force attacks and argued that it is comparable, if not outright superior, to many other password policies in common use.

Arguably the greatest benefit our system brings is the same provided by any random-password scheme: it

renders the classical dictionary-based attacks with less that a few billion items unsuccessful – precisely the attack that used to be most successful in practice. Our contribution is making them easier to remember and to use in practives and thus opening the path for more widespread popularity.

Space constraints prevented us from discussing preliminary data from field experience with end-users and experiments with variations in the user interfaces to provide countermeasures against hardware and software-based keyloggers. These may make fertile ground for future papers.

6 ACKNOWLEDGEMENTS

Thanks to Victor Hora for the endless hours tweaking the dictionaries to remove obscure words and make passphrases don't look too weird overall.

Scrabble is a registered trademark of Hasbro, Inc. and J.W. Spear & Sons.

7 REFERENCES

1. R. Morris and K. Thompson, *Password Security: A Case History*, Communications of the ACM, Vol.22, No.11, November, 1979, pp.594-597. http://citeseer.ist.psu.edu/morris79password.html
2. David C. Feldmeier & Philip R. Karn, *UNIX password security – ten years later*, Proceedings of the UNIX Security Workshop (August 1989), http://www.ja.net/CERT/Feldmeier_and_Karn/crypto_89.ps
3. Poul-Henning Kamp, *MD5 based password scrambler*, http://people.freebsd.org/~phk/
4. N. Provos & D. Mazières, *A Future-Adaptable Password Scheme*, http://www.openbsd.org/papers/bcrypt-paper.ps
5. Urity, *Cracking NTLMv2 Authentication* http://www.blackhat.com/presentations/win-usa-02/urity-winsec02.ppt#256,1,Cracking NTLMv2 Authentication
6. Microsoft Inc., *Windows 2000 Kerberos Authentication White Paper*, http://www.microsoft.com/windows2000/techinfo/howitworks/security/kerberos.asp
7. Arnold G. Reinhold, *The Diceware Passphrase Home Page*, http://world.std.com/~reinhold/diceware.html
8. John Walker, *Passphrase Generator*, http://www.fourmilab.ch/javascrypt/pass_phrase.html
9. M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson and M. Wiener, *Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security*, http://www.crypto.com/papers/keylength.pdf
10. B. Schneier, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish),* Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994, pp. 191-204, http://www.schneier.com/paper-blowfish-fse.html
11. Charles Gillman, *LMCrack - Cracked in 60 seconds*, http://www.infosecwriters.com/hhworld/hh9/lmcrack.htm
12. @Stake, *LC5 - The Password Auditing & Recovery Application*, http://www.atstake.com/products/lc
13. Solar Designer, *John the Ripper password cracker*, http://www.openwall.com/john/
14. Alec Muffet, *Crack 5.0*, http://www.crypticide.com/users/alecm/
15. Udi Manber, *A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack*, 1994, Department of Computer Science, University of Arizona, ftp://ftp.cs.arizona.edu/reports/1994/TR94-34.ps
16. Eric Rescorla, *RFC 2818: HTTP Over TLS*, 2000, http://www.faqs.org/rfcs/rfc2818.html

Appendix: The proposed Diceware-4 Dictionary for American English

Each four dice tosses selects one word. Use the two first dice to look up the row and the two last dice to look up word in the intersecting column. Repeat this process six times or more to get a secure passphrase suggestion.

	1-1	1-2	1-3	1-4	1-5	1-6	2-1	2-2	2-3	2-4	2-5	2-6	3-1	3-2	3-3	3-4	3-5	3-6	4-1	4-2	4-3	4-4	4-5	4-6	5-1	5-2	5-3	5-4	5-5	5-6	6-1	6-2	6-3	6-4	6-5	6-6
1-1	abel	able	abut	aces	acid	acme	acne	acts	adam	adds	aden	aeon	afar	afro	aged	ages	ahoy	aide	aids	ails	aims	aint	aire	airs	airy	akin	ales	alga	ally	alms	aloe	alps	also	alum	amen	amid
1-2	amir	ammo	amok	amps	anal	anew	anil	anna	anon	ante	anti	ants	aped	apes	apex	apse	aqua	arab	arak	arch	arcs	area	aria	arid	arks	arms	army	arts	arty	arum	asap	ashy	asia	asks	asps	atom
1-3	atop	auks	aunt	aura	auto	aver	avid	away	awry	axes	axis	baal	babe	baby	back	bags	bail	bait	bake	bald	bale	ball	band	bane	bang	bank	bans	bard	bare	bark	barn	bars	bart	base	bash	bask
1-4	bass	bate	bath	bats	baud	bawl	bays	bead	beak	beam	bean	bear	beat	beau	beck	beds	beef	been	beep	beer	bees	beet	begs	bell	belt	bend	bent	berk	best	beta	bets	bevy	bias	bibs	bide	bids
1-5	bier	bike	bile	bill	bind	bing	bins	bird	bite	bits	blab	blah	blat	bled	blew	blip	blob	bloc	blot	blow	blue	blur	boar	boat	bobs	body	boer	bogs	bogy	boil	bola	bold	bole	bolt	bomb	bond
1-6	bone	bong	bonn	bony	book	boom	boon	boor	boos	boot	bops	bore	born	boss	both	bout	bowl	bows	boys	brag	bran	bras	brat	braw	bray	bred	brew	brig	brim	brio	brit	brow	buck	buds	buff	bugs
2-1	bulb	bulk	bull	bump	burn	byte	cake	call	came	cane	cars	cats	cave	cent	chin	city	clap	clay	clef	cleg	clip	clod	clog	clot	club	clue	coal	coat	coax	cobs	coca	cock	coda	code	cods	cogs
2-2	coil	coin	coke	cola	cold	cole	cols	colt	coma	comb	come	cone	conk	cons	cony	cook	cool	coon	coop	coos	cope	cops	copy	cord	core	cork	corn	cost	cosy	cots	coup	cove	cowl	cows	crab	crag
2-3	cram	cran	crap	crew	crib	crim	crop	crow	crux	cuba	cube	cubs	cuds	cued	cues	cuff	cull	cult	cups	curb	curd	cure	curl	curs	curt	cusp	cuss	cute	cuts	cyan	cyst	czar	dabs	dace	dado	dads
2-4	daft	dais	dale	dame	damn	damp	dams	dane	dank	dare	dark	darn	dart	dash	data	date	daub	dawn	days	daze	dead	deaf	deal	dean	dear	debt	deck	deco	deed	deem	deep	deer	deft	defy	deli	dell
2-5	demo	dens	dent	deny	desk	dews	dewy	dhow	dial	dice	died	dies	diet	diff	digs	dill	dime	dims	dine	ding	dint	dips	dire	dirt	dish	disk	diva	dive	dock	docs	dodo	doer	does	doge	dogs	dogy
2-6	dohs	dole	doll	dolt	dome	done	dong	dons	dont	doom	door	dope	dopy	dose	dote	dots	dour	dove	down	doze	dozy	drab	drag	dram	drat	draw	dray	drew	drip	drop	drub	drug	drum	dual	dubs	duce
3-1	duck	duct	dude	duds	duel	dues	duet	duff	duke	dull	dumb	dump	dune	dusk	dust	duty	each	earl	earn	ears	ease	east	easy	eats	echo	eddy	eden	edge	edit	eggs	eire	elan	elms	else	ends	epic
3-2	eras	ergo	ergs	eros	euro	even	eves	evil	exam	exit	eyes	face	fact	fade	fail	fair	fake	fall	fame	fans	fear	feat	feds	feed	feel	feet	fell	felt	figs	file	fill	film	find	fine	fire	firm
3-3	fish	fist	five	flag	flap	flat	flaw	flew	flex	flip	flop	flow	foal	fogs	fold	font	food	fool	foot	ford	fork	form	foul	four	foxy	free	frog	from	fuel	full	fume	fund	funk	fury	fuse	gain
3-4	game	gang	gate	gave	gear	geek	gems	gene	germ	gets	gift	gird	girl	give	glad	glow	glue	glut	goal	goat	goes	gold	golf	gone	gong	good	goon	gore	goth	grab	gray	grew	grey	grid	grip	grow
3-5	grub	gulf	gums	guns	guru	gust	guts	guys	gyms	gyro	hack	hail	hair	half	hall	halo	halt	hams	hand	hang	hard	harm	hate	hats	have	hawk	hear	help	here	hero	high	hint	hold	hole	home	hood
3-6	hook	hope	horn	hose	hour	hull	hurt	hush	icon	idea	idle	inch	into	iron	isle	item	jail	jaws	jerk	jets	jobs	joey	john	join	joke	joys	judo	july	jump	june	junk	jury	just	keen	keep	kelp
4-1	kelt	kent	kept	kerb	keys	kick	kids	kiev	kill	kiln	kilo	kilt	kina	kind	king	kirk	kiss	kite	kith	kits	kiwi	knee	knew	knit	knob	knot	know	kudu	labs	lace	lack	lacy	lade	lads	lady	lags
4-2	laid	lain	lair	lake	lama	lamb	lame	lamp	land	lane	last	late	lead	leaf	left	less	liar	life	lift	like	lime	limo	line	link	lion	lips	list	live	load	loan	lock	loft	logo	logs	long	look
4-3	loom	loop	lord	lore	lose	loss	lost	lots	loud	love	luck	luke	lurk	lust	made	maid	mail	main	make	male	mall	mama	many	mark	mars	mary	mask	mass	mate	math	matt	maze	meal	mean	meat	meek
4-4	meet	mega	melt	memo	menu	meow	mess	mice	mike	mile	milk	mind	mini	mint	miss	mist	mode	monk	mood	moon	more	most	move	much	mugs	mule	muse	mush	musk	must	mute	mutt	myna	myth	nabs	nags
4-5	nail	name	nape	naps	nato	nave	navy	nazi	near	neat	neck	need	neon	nerd	nest	nets	news	newt	next	nice	nick	nigh	nile	nils	nine	nips	nits	noah	node	nods	none	nook	noon	norm	nose	nosy
4-6	note	noun	nude	nuke	null	numb	nuns	nuts	oafs	oaks	oars	oast	oath	oats	obey	oboe	odds	odes	odor	ogre	ohio	okay	once	ones	only	onto	open	oral	orca	ouch	ours	oval	oven	over	pack	pact
5-1	pads	page	paid	pain	pair	palm	pant	park	part	pass	past	pate	path	pats	paul	pave	pawn	paws	pays	peak	peal	pear	peas	peat	peck	peek	peel	peep	peer	pegs	pelt	pens	pent	perk	perl	perm
5-2	pert	peru	pest	pets	pews	phew	pick	pied	pier	pies	pigs	pike	pile	pill	pimp	pine	ping	pink	pins	pint	piny	pion	pipe	pips	pisa	pith	pits	pity	plan	play	plea	plod	plop	plot	ploy	plug
5-3	plum	plus	pock	pods	poem	poet	pogo	poke	poky	pole	poll	polo	pomp	pond	pong	pony	poof	pooh	pool	poor	pope	port	pull	pump	push	raid	rail	rain	ramp	rats	rave	read	real	rear	reef	reel
5-4	rent	rest	rice	rich	rick	ride	rime	rims	ring	riot	rise	risk	road	roar	rock	rode	roll	roof	room	root	rope	rose	rows	ruby	rude	rues	ruff	rugs	ruin	rule	rump	rune	rung	runs	runt	ruse
5-5	rush	rusk	rust	ruth	ruts	sack	sacs	safe	saga	sage	sago	sags	said	sail	sake	saki	sale	salt	same	sand	sane	sang	sank	saps	sash	save	sawn	saws	says	scab	scam	scan	scar	scat	scud	scum
5-6	seal	seam	sear	seas	seat	sect	seed	seek	seem	seen	seep	seer	sees	self	sell	semi	send	sent	serf	seth	sets	sewn	sews	sexy	sham	ship	shoe	shot	show	sick	side	sign	silk	silo	sing	sink
6-1	sins	site	sits	size	skin	skip	slam	slap	slim	slip	slow	smog	snap	snow	soap	sock	soda	sofa	soft	soil	sold	solo	some	song	soon	sort	soul	soup	spit	spot	star	stay	stop	such	suit	sure
6-2	tabs	tack	tags	tail	take	tale	talk	tall	tamp	tang	tank	tape	taps	task	taxi	team	tear	teas	teen	tell	tend	tens	tent	term	test	text	than	that	them	then	they	thin	this	thus	tick	tide
6-3	tied	ties	tilt	time	tint	tiny	tips	tire	toad	toes	tofu	toga	told	tomb	tone	tons	tony	took	tool	tour	town	toys	trad	tram	trap	tray	tree	trim	trio	trip	true	tuba	tube	tubs	tuna	tune
6-4	turk	turn	twin	type	ugly	undo	unit	upon	urge	used	user	uses	utah	vale	vamp	vans	vary	vase	vast	veil	vent	verb	very	vest	vice	view	vile	visa	void	vote	wage	wait	wake	walk	wall	want
6-5	ward	ware	warm	warn	warp	wars	wash	watt	wave	ways	weak	wear	webs	week	weep	weld	well	wend	went	were	west	wham	whap	what	whee	when	whoa	wick	wide	wife	wigs	wild	will	wimp	wind	wine
6-6	wing	wink	wins	wipe	wire	wise	wish	wisp	with	woke	wolf	wont	wood	woof	word	work	worm	wrap	xray	yale	yank	yard	yeah	year	yens	yoga	yoke	york	your	yoyo	zero	zest	zeus	zone	zoom	zulu