# EnFilter: A Password Enforcement and Filter Tool Based on Pattern Recognition Techniques

Giancarlo Ruffo and Francesco Bergadano

Dipartimento di Informatica,
Università degli Studi di Torino
{ruffo, bergadano}@di.unito.it

**Abstract.** EnFilter is a Proactive Password Checking System, designed to avoid password guessing attacks. It is made of a set of configurable filters, each one based on a specific pattern recognition measure that can be tuned by the system administrator depending on the adopted password policy. Filters use decision trees, lexical analysers, as well as Levenshtein distance based techniques. EnFilter is implemented for Windows 2000/2003/XP.

## 1 Introduction

Passwords are considered weak when they are (1) guessable (i.e., phone numbers, boyfriend names, car's numberplate, ...), (2) not resistent against dictionary driven attacks (see Crack software [6]), or (3) with low entropy [10], and, as a consequence, not secure in terms of brute force attacks.

Proactive password checking is considered the best approach [3,7,9] for avoiding selection of weak passwords. Quite surprisingly, the literature on this field is limited, if one considers the great practical importance of this problem. This does not imply that acceptable solutions to the given problem do not exist [1,2,3,7,9,10]. In Particular, the EnFilter system, which is presented in this paper, is partially based on ProCheck [1], that provides a solution for Unix systems. Some key features have been changed, as explained in Section 4, and the system has been engineered for Windows 2000/2003/XP.

## 2 Password Filters and Proactive Checking

Spafford suggests that proactive checking can take advantage of Bloom filters [9]; Davies e Ganesan adopt a markovian model in Bapasswd [3], and Nagle proposes a simple, but effective test based on a lexical analyser in [7]. With ProCheck, proactive checking is reduced to a Pattern Recognition problem, where the task is to learn the rules to classify passwords as good or bad. These rules can be represented by means of decision trees, built by classical induction algorithms ID3-like; in fact, ProCheck uses C4.5 of Quinlan [8] to build a decision tree from a "crack" dictionary (a list of examples of bad passwords), and from a randomly generated file of "good" passwords.

F. Roli and S. Vitulano (Eds.): ICIAP 2005, LNCS 3617, pp. 75–82, 2005.

In [1], these approaches are compared on the basis of the compression rate of the given dictionary and the time taken by the classifiers to decide if a password is good or bad. Of course, another important parameter is given by the classification error percentage (the sum of the rate of false negatives and false positives).

as reported in the following sections, Enfilter uses a "Dictionary Filter" based on decision tree classification. The implementation of such a filter is a scalable generalization of ProCheck.

Even though ProCheck is still referenced in [10] as the most efficient solution w.r.t. classification times and space required for storing the compressed dictionary, in [2] learning is phase is refined in order to further improve the compression rate of the derived decision tree. Yan in [10], addresses one of the most important flaws of existing proactive checkers (including ProCheck): they fail to catch weak passwords like `ca12612, hhiijjkk, a1988b53, 12a3b4c5` and `12a34b5`. In fact, such passwords can be considered invulnerable against a dictionary attack, but their *low-entropy* makes them vulnerable to brute force attacks.

As a further consideration, another weakness of ProCheck is given by its poor scalability: even if strong against attacks driven with the same dictionary used during the learning phase, ProCheck fails with passwords that trivially were not given to the inductive learner (as, for example, passwords contained in a dictionary of another language).

ProCheck has been shown to be noisy resistant in [1], i.e., classification must be strong against passwords that are slightly different from those given to the learner. But this is not enough: what if an administrator requires that the system under her responsibility is strong against a particular dictionary? How to improve the proactive checking with additional filters based on some lexical password features (for example, in a given environment we can ask the user to adopt passwords containing at least one upper case letter, two special characters and one digit)? In other words, the proactive checker should combine an efficient and complete defence against dictionary attack and a configurable filter that rejects low-entropy passwords.

Finally, another weakness of ProCheck has been highlighted during its usage at our department, even if it does not concern with a security flaw. ProCheck has been running at the student's laboratory (about 500 users) since four year, but users sometimes complain because too many *difficult* passwords were not accepted. When the checker is too severe, productivity of the user is reduced, and money is lost. The reason behind this excessive severity is explained in Section 5.1, where EnFilter's solution is proposed.

In the Microsoft Windows framework, some commercial tools allow the insertion of additional filters based on lexical rules, but to the best of our knowledge, no current access control package is able to proactively check against both a set of dictionaries and configurable filters.

EnFilter is a tool that allows for the extension of the proactive checking features by means of filters with different characteristics. This control is com-

pletely integrated in the Windows framework, with a proper management of the Security Account Manager (SAM) database and by means of the Notification Packages [5].

## 3 Dictionary Filter Based on Decision Tree Classification

The Dictionary Filter described in Section 4, is based on a decision tree classifier. This proactive checker filters out passwords that are classified as "not resistent" against a dictionary attack.

We view the training phase of a password checker as a particular Pattern Recognition problem. More precisely, we would like to use the dictionaries as sources of *positive examples* (i.e., belonging to the dictionary), and learn more concise descriptions that classify passwords as either positive or negative. We also choose to create an explicit set of *negative examples* (i.e., not belonging to the dictionary) by generating random passwords that do not belong to the dictionaries. Examples are given all at once and no new dictionaries can be added without repeating the training phase. This fits into the standard framework of one-step learning from positive and negative examples.

We chose a *decision tree* representation due to three reasons: (1) word membership is a simple problem and does not need more expressive formalisms, (2) excellent systems have been developed and implementations are available and (3) decision trees are likely to achieve greater compression on word membership problems, because prefixes common to many words need be stored only once.

Words will be described by means of so-called *attributes*, i.e. functions that take a finite number of *values*. A node of a decision tree will correspond to some attribute, and the arcs from this node to its children correspond to particular values of the attribute. Leaves will be associated to a classification, positive or negative. A decision tree can then be used for classifying a word $w$ as follows: we start from the root of the tree, and evaluate the corresponding attribute for $w$ obtaining value $v$; then we follow the arc labelled by $v$ and reach the corresponding node; then, we repeat the procedure until a leaf is reached, and output the associated classification.

As an example, suppose we have a dictionary containing just two bad passwords: "ab" and "cow". Suppose also that we generate two random passwords: "w31" and "exw". Then, we label "ab" and "cow" as positive examples, and "w31" and "exw" as negative examples. In practice, dictionaries with million of words, each with 6 characters or more, are largely used to train the system (e.g, as described in [1] ProCheck was trained on a dictionary of 3,215,846 *bad* words plus a list with the same number of examples containing random *good* words). Now, suppose we describe these examples with 3 attributes:

a1 - equals 0 if the first character is a vowel, 1 otherwise;
a2 - equals 0 if the second character is a vowel, 1 otherwise;
a3 - the length of the word.

Ley $D1$ be a decision tree that will first consider the length of the word (i.e., the value of $a3$ is checked). If the length is less than 3, the example is classified

positive, otherwise $D1$ will examine the second character, to see whether it is a vowel. If $a2 = 0$, the example is classified as positive, otherwise it is negative, e.g, $D1$ classifies examples"exw" and "w31" as negative. $D1$ is an acceptable solution of this simple classification problem.

A reference system for learning decision trees is the very well known ID3 [8]. Its basic top-down learning strategy is followed in most other methods. Initially, the tree is empty and an attribute need be selected for the root node. All positive and negative examples are associated to the root node of the tree. Among the possible attributes, ID3 chooses one that maximizes an information-theoretic quantity called the *gain*.

The gain of an attribute $a$ is computed as follows. Suppose the father node is associated to $p$ positive and $n$ negative examples, with $p + n = t$. Define the *information* represented by this partitioning of positive and negative examples as

$$I(p, n) = -\frac{p}{t} log_2 \frac{p}{t} - \frac{n}{t} log_2 \frac{n}{t} \tag{1}$$

The information after attribute $a$ has been selected is then computed as the weighted sum of the information corresponding to the children nodes:

$$I(a) = \sum_{i=1}^{s} \frac{t_i}{t} I(p_i, n_i) \tag{2}$$

where there are $s$ possible values of $a$, and $p_i$ $(n_i)$ out of the $p$ positive examples have the $i$-th value for attribute $a$. Again, $t_i = p_i + n_i$. The *gain* of attribute $a$ is then defined as $I(p, n) - I(a)$.

An important topic in decision tree learning, that is very relevant to the present study, goes under the name of *pruning*. Decision trees that are learned with the above technique will correctly classify all given examples, if the attributes are sufficient to discriminate one class from the other. In other words, the *observed error rate* will be 0. However, if the tree is large, it may happen that some leaves, or even large subtrees, are only useful for a limited set of examples.

In the given domain, predictive power of pruned decision tree has been showed to perform well in case of noisy password in [1], for instance, only 0.99% of the words of a file with 150,000 noisy words were classified as good passwords. Other experiments lead also to very encouraging results. Moreover, if pessimistic pruning is adopted at its best, compression rate can be further reduced [2].

Previous experiments performed on ProCheck showed that this approach behaves well in terms of classification time (i.e., linear w.r.t. the password length), compression and error rate: a decision tree classifier of size 24, 161 bytes was obtained from a dictionary of "bad" passwords of about 28 MB, with an error-rate of 0.5%. Using an exception file of 171 KB containing the words in the dictionary that are incorrectly classified as good passwords by the decision tree classifier, ProCheck reduces the error to one-sided and is equal to 0.32 (i.e., misclassified randomly generated password not belonging to the dictionary, but given to the learner).

## 4 EnFilter: How It Works

EnFilter maintains security and reliability properties of ProCheck, but coming through the deficiencies located above. EnFilter[1] is: (1) Highly scalable; (2) Designed for Microsoft Windows 2K/XP/2003; (3) Manageable by a simple GUI.

Once EnFilter has been installed, the user can select one of the following options:

1. Activating or deactivating EnFilter controls that are executed when a password change request is sent to the Notification Package (i.e., a user is attempting to change his/her own password).
2. Testing the strength of a password accordingly the current activated filters.
3. Setting up the installed filters, including those based on different language dictionaries.

The user interacts with an application that can be run from the Windows control panel. This application is a GUI implemented with Microsoft Visual J++. EnFilter calls a procedure contained in a DLL that implements a decision tree classifier. This is a classification procedure, which reads the stored decision trees, each for a given dictionary. Enfilter.dll is also responsible of checking if the password under test can be considered strong against all the activated filters. It implements (and exports) the PasswordChangeNotify, PasswordFilter e InitializeChangeNotify functions, as requested to the developers [5]. As a consequence, EnFilter is activated also when an user asks for a password change.

At the present time, together with the dictionary filter, that allows the user to check the passwords against a set of different language dictionaries, EnFilter supports a configuration filter checking for at least $n$ (with $n > 0$) alphabetical letters, a filter checking for at least $n$ (with $n > 0$) digits, and a filter checking for at least $n$ (with $n > 0$) special characters. EnFilter can also check for a minimum length of the password. In this way, the system administrator is able to set up different access control policies against attacks based on password guessing. In Table 1[2], we show some examples of passwords that can be accepted or rejected by EnFilter depending on the different filters that can be activated.

## 5 Filters Description

The system administrator is responsible of enabling the given filters and, when needed, of configuring them. For brevity, we do not discuss other simpler (but effective) filters, e.g. ones based on lexicographical rules.

[1] You can find further detail and the instruction to download EnFilter at *http://security.di.unito.it*.

[2] Values in the table must be interpreted as it follows: "NO" - the password is filtered out; "OK" - the password passes the test defined by the given filter. As a consequence, it is considered strong.

**Table 1.** En example of different password tests of the user "scrooge"

| Password | Windows Standard Check | Applied Filters | | | |
|---|---|---|---|---|---|
| | | Italian dictionary | English dictionary | At least 1 special character | At least 2 digits |
| Scrooge | NO | NO | NO | NO | NO |
| 2Skulls | OK | OK | NO | NO | NO |
| xltrk9u | OK | OK | OK | NO | NO |
| xltr+k9u | OK | OK | OK | OK | NO |
| Xty89'ad76 | OK | OK | OK | OK | OK |
| S(r0oge1 | OK | OK | OK | OK | OK |

### 5.1 Dictionary Filter

This filter is strongly based on the decision tree approach described in Section 3, with the following important differences w.r.t. ProCheck:

1. a new set of attributes was used. Learned decision tree is able to classify words using an attribute $a_i$ (where $i = 1, ..., 12$) for each letter $i$ in the word. Each attribute can have 27 different values, given by the corresponding letter (e.g., $a_i = 1$, if the i-th character is '$a$' or '$A$', $a_i = 2$, if the i-th character is '$b$' or '$B$', ..., else $a_i = 0$). For example, password $Scr0ge1$ is represented with the following attribute-value vector: $< 19, 3, 18, 0, 7, 5, 0, 0, 0, 0, 0, 0 >$.
2. We included a different decision tree for each dictionary. In the current distribution we *compressed* four natural language dictionaries: Italian, English, Spanish and German.

The second feature comes from the scalability and configurability requirements that can be addressed from a typical system administrator, and therefore it is just a direct consequence of making EnFilter practical and usable in the real world. The first difference is more evident if we recall (one of) the attribute-value representation that has been adopted in ProCheck:

$a_n$ = value of letter number $n$ in the word *(for n between 1 and 12)*, where values are as follows: 1 for vowels; 2 for *n, m, r, l, c, g, k, x, j, q,* and *h*; 3 for *t, d, b, p, f, v, w, s,* and *z*; 4 for digits; 5 for all other characters
$a_{13}$ = 0 if the word contains a special character, 1 otherwise;
$a_{14}$ = number of capital letters in the word.

This set of attributes results in a big compression (i.e., characters are grouped together, and therefore the number of arcs in the final decision trees is dramatically reduced), but it has two heavy drawbacks:

1. The learning phase is strongly biased on attributes $a_{13}$ and $a_{14}$: randomly generated words contain much more upper case letters and special characters than strings listed in a crack dictionary. The learned decision tree is too restrictive, because we wish to let words such as *xltrk9u* pass the dictionary filter and leave to the administrator the choice of limiting the minimum number of upper case and special characters using the lexicographical filters introduced early in this section.

2. Too many words collapse in the same description, creating misleading collisions. For example, the word *Teorema* has the same ProCheck-representation (i.e., $< 3, 1, 1, 2, 1, 2, 1, ?, ?, ?, ?, ?, 1, 1 >$[3]) of the string *woIxaku*, that it does not belong to any of the used dictionaries.

With the new set of attributes, EnFilter performed well in terms of compression and error rate. In fact, we observed an average of 0.5% of false positives[4], with a confidence interval of $[0.35, 3.84]$. These results are comparable with ProCheck, with its average of 0.53% of false positives, and with a confidence interval of $[0.25, 0.67]$. Moreover, we have an identical false negatives rate in both systems. Despite to these encouraging results in terms of error percentage, the compression, as expected, decreased from a rate of 1000 to 1, to a ratio of 100 to 1.

It goes without saying that the compression factor is not so important as some years ago, because of the bigger capacity and lower cost of current memorization devices. Nevertheless, the EnFilter distribution is very small in size. It includes the following decision trees: English.tree (394 KB), Italian.tree (241 KB), Spanish.tree (55 KB), German.tree (69 KB), that can be further compressed in a zipped archive. Finally, the entire installation package is sized 621 KB.

### 5.2 Directory Service Filter

Another filter currently under testing is based on another information theoretic quantity called *Levenshtein distance* [4]. It returns the distance between two strings, which is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a character.

Let $s$ and $t$ be two strings of length $n$ and $m$, respectively. Given two characters $s_i$ and $t_j$, which are respectively the i-th and the j-th character of $s$ and $t$, let us define function $r$ as follows: $r(s_i, t_j) = 0$, if $s_i = t_j$; $r(s_i, t_j) = 1$, otherwise. Now, we build a matrix $D$ of integers, with dimension $(n+1)(m+1)$. Values in $T$ are defined recursively as it follows:

$$
\begin{aligned}
D(i,0) &= i,\ i = 0, 1, ..., n \\
D(0,j) &= j,\ j = 0, 1, ..., m \\
D(i,j) &= min(D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+r(s_i,t_j)),\ i \neq 0 \wedge j \neq 0
\end{aligned}
$$

The element $D(n+1, m+1)$ gives the Levenshtein distance $d(s,t)$ between strings $s$ and $t$. For example:
$d(\text{"}scrooge\text{"}, \text{"}scrooge\text{"}) = 0$, $d(\text{"}scrooge\text{"}, \text{"}S(r0oge1\text{"}) = 5$, and so on.

We used this distance in order to validate the password against the many values retrievable from the user's entry of a given Directory Service (e.g., LDAP,

[3] The value ? is used when the corresponding attribute is undefined. In this example, the word "Teorema" is 7 character long, and therefore attributes $a_8, ..., a_{12}$ are undefined.

[4] false negatives, i.e., words in the dictionary erroneously classified as good passwords, are not considered, because we use an exception file for them.

Microsoft Directory Service). The filter accepts only the passwords having a Levenshtein distance greater than a given threshold $\varepsilon$ from all the string values contained in the user's entry of the local directory server. Such information usually contains many user's personal information, like birthdate, phone numbers, address, and so on. Observe that the degree of variation can be properly configured, i.e., threshold $\varepsilon$ can be set by the system administrator, even if a default value of 3 is given if this filter is activated.

## 6 Conclusion

EnFilter, a proactive password checker designed and implemented for Microsoft Windows platforms, has been introduced. It is a configurable and scalable tool, which leaves the administrator the responsibility of adequating filters to the password policies. In particular, the "Dictionary Filter" improves the previous results obtained with ProCheck, reducing false alarms caused by the absence of special characters or upper cases in the checked password. Moreover, EnFilter does not increase the false negative rate, that is anyhow reduced to zero by adopting small sized exception files.

## Acknowledgements

The authors wish to thank the anonymous referees for helping them to significantly improve the quality of the presentation. This work has been partially financially supported by the Italian FIRB 2001 project number RBNE01WEJT "Web MiNDS".

## References

1. Bergadano, F., Crispo, B., and Ruffo, G. High Dictionary Compression for proactive password checking on ACM TISSEC, 1(1), Nov. 1998.
2. Blundo C., D'Arco P., De Santis A., Galdi C., Hyppocrates: a new proactive password checker, The Journal of Systems and Software, N. 71, 2004.
3. Davies, C. and Ganesan, R. Bapasswd: a new proactive password checker In Proc. of 16th NIST-NCSC National Computer Security Conference (1993).
4. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl., 6:707-710, 1966.
5. Microsoft Knowledge Base HOWTO: Password Change Filtering & Notification in Windows NT - article n. 151082
6. Muffett, A. Crack 4.0, 5.0 11.
7. Nagle, J. B., An obvious password detector. In USENET news - comp.sources.unix 16 (60), 1988.
8. Quinlan, J. R. C4.5: Programs for Machine Learning Morgan Kaufmann, San Mateo, CA.
9. Spafford, E.H. OPUS: Preventing Weak Password Choices Computers and Security, 11, (1992) pp.273-278.
10. Yan, J. A Note on Proactive Password Checking, ACM New Security Paradigms Workshop, New Mexico, USA, September 2001.