File Security and the UNIX Crypt Command

J.A. Reeds

P.J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sufficiently large files encrypted with the UNIX® *crypt* command can be deciphered in a few hours by algebraic techniques and human interaction. We outline such a decryption method, and show it to be applicable to a proposed strengthened algorithm as well. We also discuss the role of encryption in file security.

May 30, 1998

File Security and the UNIX Crypt Command

J.A. Reeds

P.J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

1. File Security

Sometimes one wants to protect a file from being read by unauthorized users or programs, while still keeping the file available to its proper users. Only in isolation is the problem easy: put the file on a machine only you have access to, and keep all copies of the file locked up. *crypt* is useful in the more complicated environment of a multi-user system. *crypt* is a file-encryption program which is also part of one of the text editors. The algorithm is described in the next section. The advantage of having the algorithm embedded in an editor is that the clear text never need be present in the file system.

No technique can be secure against wire-tapping or its equivalent in the computer. Therefore no technique can be secure against the system administrator or other sufficiently privileged users. For these folk it is a simple matter to replace the encryption programs with programs that look the same to their users, but which reveal the key to the sufficiently privileged. Sophisticates may be able to detect this kind of substitution if it is not done carefully, but the naive user has no chance.

To protect files from being read by a casual browser there are two independent techniques, permissions and encryption. The authorization mechanisms supported by the system may make the file inaccessible to any but its owner. Encryption may make the contents incomprehensible. The former does not protect copies of the file on dump tapes. The latter is difficult to implement. The difficulty is not in finding a secure encryption algorithm, but in finding one that is not prohibitively expensive to use, not subject to fast search of key space, fits in with an editor, and is also suffently secure.

File encryption then is roughly equivalent in protection to putting the contents of the file in a safe or a locked desk or an unlocked desk. The technical contribution of this paper is that *crypt* is rather more like the last than the first.

2. UNIX® Crypt

The UNIX crypt command operates on consecutive blocks of 256 characters, which we term *cryptoblocks* to avoid confusion with the file system blocks. If the *i*-th plaintext and ciphertext characters in the *j*-th cryptoblock are denoted p_{ij} and c_{ij}, respectively, they are related by the following formula.

$$c_{ij} = R^{-1}[S[R[i+p_{ij}]+j]-j]-i \qquad (1)$$

In (1) addition and subtraction are done modulo 256. *R* is a permutation of the set {0, ..., 255}, *S* is a self-inverse permutation of the same set, having no fixed points. Therefore *S* is the product of 128 disjoint 2-cycles, and for all *i* and *j* it is true that p_{ij} ‵ c_{ij}. *R* and *S* constitute the key of the cipher, and thus are not known to the cryptanalyst at the beginning of his labors. (See section 5 for a discussion of how they are determined from the key the user types, and how part of the key the user types can be determined from *R* and *S*.)

An operator notation is more useful, in which equation (1) can be rewritten

$$c_{ij} = C^{-i}R^{-1}C^{-j}SC^{j}RC^{i}p_{ij} \qquad (2)$$

where *C* mapping x to $x+1$ is the cyclic shift transformation (Caesar shift is the usual jargon).

One weak point in the cipher is that the index *i* hardly enters into formula (2). If we let

$$A_j = R^{-1}C^{-j}SC^{j}R \qquad (3)$$

then

$$c_{ij} = C^{-i} A_j C^i p_{ij},$$

where A_j is self-inverse, and without fixed points.

This decomposes the cryptanalysis into two parts, the first being the recovery of A_j in each of several successive cryptoblocks, and the second being processing information about the A_j's to get *R* and *S*.

3. Recovering A_j

3.1. Known Plaintext Solution

Suppose the cryptanalyst has parallel plaintext and ciphertext. This should be enough to recover most of the A_j. Concentrate on one cryptoblock, and drop the subscript *j*. For each value of *i* for which the cryptanalyst has c_i and p_i

$$C^i c_i = A C^i p_i$$

from the definition of *A*. Thus $A[i+p_i]=i+c_i$, and because *A* is self-inverse, $A[i+c_i]=i+p_i$. If all 256 plaintext characters are known for the cryptoblock, there will be a lot of these equations, and most of *A* will be known.

More precisely, *A* is the product of 128 disjoint 2-cycles. Each *i* for which the plaintext is known determines one of the 2-cycles. If one assumes that the 2-cycles have equal probability of being chosen, the chance of a given 2-cycle not being chosen is $(127/128)^{256}=(1-2/256)^{256}$, the expected number of 2-cycles not chosen is $128(1-2/256)^{256}$, and the expected number of known values is approximately $256(1-e^{-2})$, which is 221.35. Thus, each block of known plaintext should give all but about 35 of the values of A_j.

3.2. Unknown Plaintext Solution

This, of course, is harder. We assume that the plaintext is all ascii, and that the cryptanalyst has a stock of probable words or phrases that the plaintext plausibly contains.

We proceed by trying to place a probable word in all possible positions in the current cryptoblock. Most of these trial placements will result in contradictions. Either they imply that some plaintext characters cannot be ascii, or they are self-contradictory, or they contradict the implications of a previous placement of a probable word. We consider these cases one by one.

Suppose that one plaintext character, say p_i, is known. Then one of the 2-cycles of *A* is known, the one which interchanges p_i+i and c_i+i. There are 255 other values of *i* for which c_i+i might fall in this 2-cycle, and the chance that none does is $(127/128)^{255}$, which is about .135. (Since the success of the attack doesn't depend on these calculations, the hidden randomness assumptions can remain hidden.) So with probability about 86.5% we find some other value of *j* for which c_j+j is in the known 2-cycle, and so the corresponding value of p_j is known too. If the initial guess at p_i were wrong, then this guess at p_j has a 50% chance of not being ascii (assuming that all 128 ascii characters are legal). Thus each individual guess at a plaintext character has better than a 40% chance of being shown wrong because it would imply some plaintext character is not ascii. A longer probable word, incorrect in all its letters, is even less likely to be acceptable.

There is another kind of constraint probable text imposes on the ciphertext. If there are two places, say *i* and *j*, in the same cryptoblock of plaintext satisfying $p_i+i=p_j+j$, then the definition of *A* shows that $c_j-c_i=i-j$. For instance, the word 'include', common near the beginning of C programs, contains two of these constraints, 'n.l' and 'i....d'. One expects only about one place in each cryptoblock where even one of these constraints is satisfied (other than at the place where 'include' belongs), so the chance of the two being satisfied erroneously is quite small (but not negligible).

Finally, a trial placement may be incompatible with earlier, accepted, placements of probable words.

This is all easy to package into programs. One could start with a special-purpose editor which gets probable text from the user and presents all contradiction-free placements and resulting decipherment. The user then accepts those placements that produce the best looking decipherment, and suggests new probable words. Such an editor can be used to decrypt a completely unknown C program in a few hours, or less. Getting one block generally takes a while, but then the cryptanalyst has a good idea of the style and subject of the program, and other blocks take less time.

Sometimes it is useful to first look for all contradiction-free placements of a single long probable word in all blocks of a file rather than look for several probable words in a single block.

3.3. A statistical attack

The following idea is due to Robert Morris. Before attacking an unknown plaintext, one can automatically generate a lot of plausible plaintext by a statistical analysis of each of the cryptoblocks.

In essence one applies the unknown plaintext attack outlined above to the 20 one-letter probable words formed by the 20 most common ascii letters. Each of the possible 5120 trial placements of these 'words' in a given cryptoblock is scored according to the resulting plaintext it generates, using a formula involving logarithms of the probabilities of the ascii letters. Any decipherment resulting in non ascii letters is immediately ruled out. Otherwise disputes between contradictory trial placements are resolved in favor of the trial placement with the greater score.

This process ends with a partially deciphered cryptoblock with lots of 'noisy' plaintext visible to an indulgent eye. It is easy to use guesses based on this noisy plaintext as a starting point for a session with an interactive *crypt*-breaking editor, as described above.

4. Knitting

Once several blocks have been mostly decrypted, the corresponding information about the A_j can be used to recover R and S. Let $Z = R^{-1} C R$. Then (3) can be rewritten as

$$A_j = Z^{-j} A_0 Z^j$$

and hence

$$Z A_{j+1} = A_j Z.$$

We call this the *knitting equation*: Z knits the A_j sequence together. We solve this last equation for Z, from which a value for R can be found. Once R is known, the equation

$$S = R A_j R^{-1}$$

gives a value for S. Even if all this works out, R and S are not completely determined, for if the pair (R, S) works, so will $(C^k R, C^k S C^{-k})$, for any k.

The idea behind solving for Z is simple. Suppose we hypothesize $Zx = y$. Then for each value of j for which $A_j[y] = v$ and $A_{j+1}[x] = u$ are known, it must be true that $Zu = v$. Hence if several successive A's are fairly well known, each hypothesis about Z will generate several more, and so forth, and all these have to be consistent with all that is known about the A's. In practice there is a chain reaction of hypotheses about Z which quickly lead to a contradiction if the initial guess was wrong.

Once Z has been mostly recovered, one can use the knitting equation to fill in missing values in the A's.

5. Recovering some key bytes

Once R and S are known, it is possible to determine the first 2 letters of the key the user typed. At the same time we discover which of the 256 equivalent (R, S) pairs was generated by *crypt*.

5.1. How *R* and *S* are Built

The user's key is transformed into 13 bytes b_0, b_1, ..., b_{12} by the same subroutine used to encrypt UNIX passwords. b_0 and b_1 can be any characters the user can type, so $0 \leq b_0, b_1 < 128$, while the rest of the b_i are restricted to the 64 characters '/', '.', '0', ..., '9', 'a', ..., 'z', 'A', ..., 'Z'.

From these bytes the program builds various pseudo-random numbers from which it constructs R and S. The details are a bit tedious. First mix all the b_i together:

$$x_0 = 123$$
$$x_{i+1} = x_i b_i + i \quad 0 \le i < 12.$$

Here arithmetic is done modulo 2^{32}, and $-2^{31} \le x_i < 2^{31}$. Now compute a sequence of *s*'s:

$$s_{-1} = x_0$$
$$s_i = 5s_{i-1} + b_i \quad 0 \le i < 256.$$

Here s_i is computed modulo 2^{32}, $-2^{31} \le s_i < 2^{31}$, and the subscript on *b* is evaluated modulo 13. Next, compute some *r*'s:

$$r_i \equiv s_i \pmod{65521},$$

where the peculiar notation means that r_i has the same sign as s_i and $-65520 \le r_i \le 65520$. Whew. Now compute

$$u_i \equiv r_i \pmod{256}, \quad 0 \le u_i < 256,$$
$$v_i \equiv r_i/256 \pmod{256}, \quad 0 \le v_i < 256.$$

Alternately, write r_i in 2's complement binary. Then u_i is the number given by the low order 8 bits, and v_i is the next 8 bits.

Initialize an array representing $R(i)$ so that $R(i)=i$ for all *i*. Then compute $R(i)$ from the x_i by doing

$$x_i \equiv u_i \pmod{i+1}, \quad 0 \le x_i < i+1$$
swap $R(255-i)$ and $R(x_i)$,

successively for $i=0$, $i=1$, ..., $i=255$. If the r_i were uniformly distributed over a suitable set of integers, then all 256! possible *R* would be equally likely.

Initialize an array representing $S(i)$ to $S(i)=0$ for all *i*. Then for $i=0$, $i=1$, ..., $i=255$, successively,

```
If $S(255-i) \ne 0$, do nothing.
Otherwise, let
    $y_i \equiv v_i \pmod{i}$,
and then
    while $S(y_i)=0$
        $y_i \equiv y_i+1 \pmod{i}$
    then $S(255-i)=y_i$, and $S(y_i)=255-i$.
```

Then *S* is the product of 128 2-cycles.

5.2. Finding *k*

Decrypting a file produces 256 cryptographically equivalent possibilities for (R,S). It is possible to determine which *crypt* used and to recover the b_i all at once.

First suppose we knew the values of all the r_i. Then

$$s_i = 65521c_i + r_i, \qquad -65521 \le c_i \le 65521$$
$$s_{i+1} = 5s_i + b_i + M_i 2^{32}, \qquad -2 \le M_i \le 2.$$

The bounds on *c* and *M* follow from the bounds on *s* and *b*. Substituting and rearranging gives

$$b_i = r_{i+1} - 5r_i - 225M_i + 65521(c_{i+1} - 5c_i - 65551M_i).$$

Consider this equation modulo 65521. *b* must be ascii, at least, there are only 5 possible values for M_i, and the *r*'s are known. Bogus values are unlikely to give acceptable *b*'s. Also, each value of b_i is constrained by values of *i* 13 apart. So knowing the r_i will determine the b_i.

For the first part, we try each of the 256 possibilities in turn, assuming the current ones are the correct *R* and *S*, and attempting to reconstruct all the *b*'s. In practice, for the 255 incorrect values of *k* the process below fails to construct a consistent set of *b*'s, and so excludes all but the correct *k*.

From the trial *R* it is easy to read off the x_i which generated it. First, $x_{255}=R(255)$. Then modify *R* by making $R(x_{255})=R(255)$, and proceed by induction. Here's an example, with a permutation on 8 things:

k	0	1	2	3	4	5	6	7
$R(k)$	2	6	5	7	0	1	3	4

$R(7)$ was constructed, by the algorithm above, by switching the previous value of $R(7)$ with some $R(i)$ with i less than 7. Hence x_7 is 4, and at the next step, we consider a permutation on 7 things:

k	0	1	2	3	4	5	6
$R(k)$	2	6	5	4	0	1	3

From this x_6 is 3, and so forth. The process is just running the construction of R backwards. Note that although R could plausibly be argued to be a random permutation, it is one which in no way conceals the data from which it was constructed. Randomness in the sense of uniform distribution is by no means synonymous with the intuitive meaning of not containing information. It is the latter property which is important to cryptography.

A similar process allows us to get some of the y_i. We get y_{255} the same way we got x_{255}, but we can only deduce other y_i when we are sure that neither the *while* step nor the do-nothing step in the algorithm above were not executed.

Now how close do x_i and y_i come to determining r_i? First suppose we knew u_i and v_i. Then we would have 16 bits in the binary representation of r_i. Unfortunately the possible values of r_i require nearly 17 bits, so each pair $(u_{i,}v_i)$ probably is consistent with two values of r_i, and so in the expression for b_i above there are likely to be 4 choices for (r_i, r_{i+1}). Clearly there is still not much chance of getting even a single bad guess of a b_i.

So how do we get u_i and v_i? Since

$$x_i \text{ a } u_i \pmod{256}$$

for each i g 128 there are at most two choices of u_i (namely x_i and $x_i + i + 1$) for each value of x_i. Likewise, if we know y_i, there are at most two choices for v_i. Thus there are 4 more choices to be made for each guess at an r_i.

In practice this is all nearly enough to determine all of the b_i uniquely for exactly one value of k. That is, there is only one of the 256 equivalent (R,S) pairs for which there are any b's left, and then there are never more than a few hundred possible sets. Only one of them, and therefore the correct one, regenerates R and S. There was no trouble doing this in 190 trials. Each trial takes a minute or two of computer time. Thus decrypting files enough to determine (R,S) also enables the cryptanalyst to find b_0, ..., b_{12}.

This would not be more than a curiosity, except for the fact that the first two bytes of the user's key pass through unchanged and become b_0 and b_1. This knowledge is clearly of great use in guessing how the user makes up his keys.

6. A proposed enhancement

A recent proposal for strengthening the *crypt* command is as follows. Instead of relating the i-th plaintext and ciphertext letters in the j-th cryptoblock by

$$c_{ij} = C^{-i} R^{-1} C^{-j} S C^{j} R C^{i} p_{ij}$$

it is proposed to use

$$c_{ij} = C^{-f_i} R^{-1} C^{-j} S C^{j} R C^{f_i} p_{ij}.$$

R and S are as before. The new item is the function f which may be interpreted as an irregular rotor motion. The key now is the triple (R,S,f). If f were known, then the new cipher would be breakable by the same methods as the old.

6.1. Known plaintext attack of proposed enhancement

We first recover the f_i, and the proceed as before. We note that in a given cryptoblock, if $p_i + f_i = p_k + f_k$, for some i and k, then $c_i + f_i = c_k + f_k$. Also, because the encryption is an involution, if $p_i + f_i = c_k + f_k$, then $c_i + f_i = p_k + f_k$.

We can exploit these identities as follows. If

$$p_i + f_i = p_k + f_k \tag{4}$$

then

$$c_i + f_i = c_k + f_k$$

and hence

$$p_i - p_k = f_k - f_i$$

and

$$c_i - c_k = f_k - f_i$$

and

$$p_i - p_k = c_i - c_k . \tag{5}$$

Thus (4) for some i and k imply (5) for the same i and k. We take the occurrence of (5) as a sign that the four equations of (4) might have happened, and further take the common value $p_i - p_k = c_i - c_k$ as a *vote* for the value of $f_k - f_i$. Similarly, the occurrence of

$$p_i - c_k = c_i - p_k$$

is a vote that $f_k - f_i$ has this common value.

Experiments show that of all occurrences of (5), about half are caused by (4) and half are accidental. The accidental occurrences scatter their votes higgledy piggledy but the causal occurrences vote *en bloc* for the correct value of $f_k - f_i$.

Thus for each cryptoblock we enumerate all votes of the above type, representing them by triples (i,k,d) meaning that there is a vote that $f_i - f_k = d$. Let **S** be the set of all the votes. We attempt to resolve these votes by discarding about 1/2 of them and building the others into a self-consistent set of values for the f_i. Note that although each instance of a vote comes from one cryptoblock, the f_i are the same from block to block, so that the votes from all the known blocks can be combined.

Each cryptoblock contributes about 500 such votes, so 2500 characters of known plaintext will generate about 5000 triples.

6.2. Voting

We are given a set **S** of 5000 or more triples (i,k,d), each representing an equation

$$f_i - f_k = d .$$

We want to find a maximal consistent subset of these equations. That is, we want values $f_0, f_1, \cdots, f_{255}$ that solve as many of these equations as possible. Here is one method that works in practice.

We solve instead a seemingly more complicated problem: find probability laws P_0, P_1, ..., P_{255}, each on the integers mod 256, such that

$$L = \prod_{(i,j,d)\,\mu\mathbf{S}} \left[\frac{1}{2}\frac{1}{256} + \frac{1}{2} P(X_i - X_j = d) \right]$$

is maximized, where the X's are independent random variables, each X_i with law P_i. If we let $g_{ij} = P(X_i = j) = P_i(\{j\})$, then

$$L = \prod \left[\frac{1}{2}\frac{1}{256} + \frac{1}{2} \sum_t P(X_j = t \text{ and } X_i = t + d) \right]$$

$$= \prod \left[\frac{1}{2} \frac{1}{256} + \frac{1}{2} \sum_t g_{i,t+d} g_{j,t} \right].$$

L is a function of the 65,536 non-negative variables g_{ij} subject to the 256 constraints $\sum_{j=0}^{255} g_{ij} = 1$. Such a function may be readily maximized by the algorithm of Baum and Eagon, also called the EM algorithm.

In practice the maximizing g_{ij} values are all close to 0 or 1, and we take for f_i that value of j for which g_{ij} is biggest.

This takes about 20 minutes of VAX time.

7. Summary

It turns out from this work that the UNIX file encryption command is not as strong as its designers had hoped. While a simple modification like the one discussed above makes encrypting short files safer, finding a much more satisfactory replacement appears hard.

8. Reference

Leonard E Baum and J A Eagon, *An Inequality with Applications to Statistical Estimation for Probabilistic Functions of Markov Processes and to a Model for Ecology,* Bulletin of the AMS, **73** May 1967, pp 360-363.