CFS travails

by Travis Howard

Travis Howard is a independent security researcher currently in search of employment. His home page is *http://travcom.tripod.com.*

auto92089@hushmail.com

This is a story about a mysteriously corrupted encrypted file system and my attempts to recover the data. Forgive me if my narration of it is a bit rough, as I am not exactly sure what happened; all I can do is make observations. This is my first attempt to turn a chronological log of observations into a good narrative. In doing so I am gaining a new appreciation for the work of investigative journalists, who must go through a similar process to turn their notes into a compelling story. This is a work in progress. If you have solutions to any problems I mention herein, please email them to me.

The encrypting file system under observation is CFS, Matt Blaze's "Crypting File System." Parts of it date back to 1987, making it a rather venerable piece of code. Matt Blaze is well known in the computer security community, and the code has been available for scrutiny for some time (*http://www.crypto.com/software/*). The code itself is fairly portable and might be the most popular encrypting file system in UNIX.

First, a little background on how CFS works. CFS stores each encrypted file as a separate entity in the file system (i.e., it is not an encrypted disk device). Storage for each file also includes an initialization vector (IV). Early on, CFS used the inode number to store the IV. Subsequently, it used the GID field. Finally, Matt decided to use a separate symlink (with the prefix ".pvect_") to hold it (by pointing to it). In CFS, the IV is used to make files with identical contents encrypt to different data streams to prevent snoops from noticing correlations between files.

I encountered a bit of file system corruption using CFS.

It's hard to know exactly when the corruption occurred or why. My logs say I had some disk problems around 2 May 2001. I believe some of the encrypted files were moved to lost+found. My notes say that I restored a copy of the encrypted directory hierarchy to make comparisons and moved some files. I was not aware of the significance of the symlink/IVs at that time and probably made some mistakes de-synchronizing them.

I did make some edits to sensitive files on 15 Dec 2001. The files I was working on were not apparently corrupted. Subsequently, a period of hassles ensued, getting CFS to co-exist with my OS, which had a new v3 RPC mechanism. I recompiled various versions of CFS around this time, for better or for worse using an OS release with a buggy compiler (egcs-1.1.1). This compiler is known to generate incorrect code, and could be the source of some of the data corruption.

On 6 Dec 2003, decrypting certain files yielded corruption. That is, ASCII text files suddenly appeared to be binary files. How can I quantify this corruption? I ran a tool called "ent" on the files, and it yielded some interesting results:

```
    $ ent corrupted_text_file
    Entropy = 5.558949 bits per byte.
    Optimum compression would reduce the size of this 6588 byte file by 30
percent.
    Chi square distribution for 6588 samples is 40067.26, and randomly
would exceed this value 0.01 percent of the times.
    Arithmetic mean value of data bytes is 63.7025 (127.5 = random).
    Monte Carlo value for pi is 4.000000000 (error 27.32 percent).
    Serial correlation coefficient is -0.087718 (totally uncorrelated = 0.0).
```

The entropy is one measure of the amount of unpredictable information in the file. For totally random files, it should be 8 bits per byte, and for a file full of nulls, it should be 0 bits per byte. The optimum compression measurement merely describes the percentage of predictable information in the file. Chi square is a statistical text that involves sorting data in “bins” (probably 256 bins in this case, one for each possible byte), squaring the difference between the actual bin contents and the predicted values, then dividing by the probability of that bin being selected (in this case, 1/256 since all bytes are equally probable). Thus, the more a file deviates from random, the higher the chi square score, and the lower the percentage of cases that would exceed it. The arithmetic mean is obvious; the Monte Carlo approximation involves constructing a virtual 1 x 1 dart board with a circle of diameter one inside it, and using the samples as coordinates of darts, then dividing the number that land inside the circle by the total number of darts, and using that to compute a value of pi (it converges rather slowly). Finally, the serial correlation merely describes how much each sample depends on the prior sample.

Normally when encryption goes awry (from using the incorrect key, for example), you get something indistinguishable from random bits. As you can see, the files are definitely not random. Random looks like this:

```
$ ent random_file
Entropy = 7.970002 bits per byte.
Optimum compression would reduce the size of this 6144 file by 0 per-
    cent.
Chi square distribution for 6144 samples is 250.58, and randomly would
    exceed this value 50.00 percent of the times.
Arithmetic mean value of data bytes is 126.5417 (127.5 = random).
Monte Carlo value for pi is 3.152343750 (error 0.34 percent).
Serial correlation coefficient is -0.009299 (totally uncorrelated = 0.0).
```

On the other hand, they are more random than ordinary text files:

```
$ ent uncorrupted_text_file
Entropy = 4.436244 bits per byte.
Optimum compression would reduce the size of this 5134 byte file by 44
    percent.
Chi square distribution for 5134 samples is 76869.97, and randomly would
    exceed this value 0.01 percent of the times.
Arithmetic mean value of data bytes is 92.6139 (127.5 = random).
Monte Carlo value for pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is -0.008061 (totally uncorrelated = 0.0).
```

I then wrote two programs, freqcount.pl and freqgraph.pl, to explore the files further. The former counts the frequency of each byte, and the latter represents it graphically (in text mode) as bars of asterisks running horizontally, so you can compare two files in side-by-side windows. What I saw appeared to be disruption of the large frequency spike of the space character; however, it was definitely not uniform. It looked as though it had gone through some polyalphabetic substitution with a periodicity of four or so.

My first reaction was to go to backup tapes. However, my backup tapes for 15 Oct 2003 turned out to be faulty; I had run out of tape but hadn’t noticed. Although I have many backups, this corruption went unnoticed for so long that I didn’t have a backup set old enough.

My next reaction was to see if cfsd, the NFS-like daemon that serves up the files, was at fault. I decided to try ccat (a stand-alone utility) on some of the files to eliminate cfsd as a source of errors. Unfortunately, it warns that it only works on old-format CFS files. CFS went through several stages of evolution; stand-alone tools like ccat have not been kept up to date with the various evolutionary changes.

I should mention here some of the other tools I used. I used bc quite a bit, even though it has the annoying property of not accepting lowercase letters as hex digits and does not have a bitwise XOR operator. So I often ended up using Perl instead. If you try stuff like this you'll need the chr and ord functions. I also referred to an ASCII chart quite a bit.

I read Matt Blaze's notes.ms, and read through some of his code, particularly cmkdir.c. I found a few errors that are security-relevant; I must question how much trust I put in this software without doing even a cursory review of the code. My findings:

1. The type of cipher is included, apparently erroneously, in some key manipulations:

```
struct cfs_admkey {
    ciphers cipher;
    union {
            cfs_adm_deskey deskey;
            cfs_adm_3deskey des3key;
            ...
    } cfs_admkey_u;
};
cfs_admkey k;
...
/* now we xor in some truerand bytes for good measure */
bcopy(&k,ekey,32);  /* assumes key material < 32 bytes */
for (i=0; i<32; i++) {
    ekey[i] ^= randbyte();
}
encrypt_key(&k,ekey);
bcopy(ekey,ek1,32);
decrypt_key(&k,ek1);
/* new &k is our real key */
```

2. A file called "..." is created. Apparently this was to provide a somewhat-known plaintext so that the program can tell whether you have given the correct key or not. CFS attempts to make this 8-byte value half random, but due to shifting in the wrong direction, the attacker knows 7 of the 8 bytes, almost giving him or her a full known-plaintext. By contrast, PGP uses two bytes of known plaintext to determine if you have the correct key (meaning that it accepts the wrong key 1 in 2^{16} times).

```
char str[8];
...
strcpy(str, "qua!");
/* now randomize the end of str.. */
r = trand32();
for (i=0; i<4; i++)
    str[i+4]=(r<<(i*8))&0377;
```

At this point I thought that the IV was stored in the inode numbers. CFS used to do this, and I almost certainly changed the inode numbers on several files during my restoration efforts. There are 2^{32} possible inodes, and I would have to search for the correct one for each file. This could have taken a while. I would have had to be clever and crafty to finish this project in my lifetime. But perhaps it was not as impossible as it seemed; the stream generated from the inode number is XORed with the data at one point; perhaps I wouldn't have had to try all of them in a brute-force method. Also, the file system may not have 2^{32} inodes to try out—perhaps much fewer. Yes, this might be possible.

I thought my problem was that there's a different inode used to seed a pRNG of some kind to create a stream that is being XORed with my program's text. To extract the original text, I would need to know the proper inode number. The size of the inode number tells me its maximum theoretical size, and if the file system it is on has only grown over time, its maximum inode value would tell me an even lower upper bound, while the inodes of nearby files might give me a good place to start searching. I also needed to recognize when I'd deduced the correct inode number.

Knowing whether the inode number is right or not would probably require running statistical tests on the contents of the file. On top of all this, these files' contents are sensitive (that's why they're encrypted), so I wouldn't be able to distribute any brute-forcing. I would either have to use Matt Blaze's CFS code or develop my own and test it to make sure it was doing exactly the same thing as his.

At this point I decided to print out all the papers on CFS and dive deep, really deep, into the belly of the beast. I really needed to understand the CFS encryption technology. Perhaps I could be clever and find an algorithmic shortcut, or a known-plaintext situation.

On 6 Jan 2004, I found there were "only" about 280,000 inodes on that file system. That meant 280,000 trials, far more reasonable than the earlier estimate, which ran as high as 4,294,967,296.

Checking each decryption for the bytewise frequency count of the space character could do a 1:256 winnowing. I'd need about another 1:100 reduction of those candidates to perform manual inspection (I figured 10 manual inspections per file is my upper limit).

On 7 Jan 2004, I thought I might have found my answer, due to the equations in Matt's papers:

```
D_p = DES^-1(K_2, E_p xor DES^1(K_1, g(p mod m)))
      xor DES^1(K_1, f(p mod m)) xor i
```

All the data are 8 bytes wide, except perhaps the keys. I think in this case, everything is known except *i*, which "is a bit representation of a unique file identifier derived from the UNIX inode number and creation time of the encrypted file."

Still, in the equations above, *i* is the only variable that is unknown to me. Since it is not used to seed any RNGs or to key any ciphers, I was in luck. Looking at the decryption equation, if *i* were wrong it would give me the correct data but XORed with some constant. That would be consistent with the entropy measurements I had made, which indicated that the file was definitely non-random but did not have the strong frequency spike of the space character in English prose.

Stated differently: If you plugged the wrong value of *i* (let's call it *i'*) into the above equation, you'd get this:

```
D_p' = D_p xor (i' xor i)
```

That's what was happening to me. Each 8-byte chunk of data that I saw was the original plaintext, XORed with some 8-byte constant value, which was the error (XOR) of *i'* and *i*. Statistically, if I know the most common D_p′ and the most common D_p, and I know *i′*, I can solve for *i*:

```
i = D_p’ xor D_p xor i’
```

Now, in that equation, each variable is an 8-byte array. At first, I thought *i* was 4 bytes long, and it simply concatenated it to itself to form an 8-byte value (later I learned that *i* was indeed an 8-byte value). Each byte of *i* can be solved independently:

```
i[0] = D_p’[0] xor D_p[0] xor i’[0]
i[1] = D_p’[1] xor D_p[1] xor i’[1]
…
i[3] = D_p’[3] xor D_p[3] xor i’[3]
i[0] = D_p’[4] xor D_p[4] xor i’[0]
…
i[3] = D_p’[7] xor D_p[7] xor i’[3]
```

In other words, I essentially had a polyalphabetic substitution cipher, with four substitution tables all built on bitwise XOR with a constant. For text files, I might even be able to solve each byte of the value *i* independently, suggesting only 4 x 256 = 1024 guesses! This was workable! In fact I could vary each byte of *i* at the same time, meaning only 256 decryptions!

On closer reading of the release notes (notes.ms), I found that CFS now stores an IV for filename in a symbolic link .pvect_filename. Amazingly, I had traced corruption to those files missing symbolic links! Hooray! Now I knew exactly what was missing, which files were corrupted, and how to fix it! I verified that these .pvect_ files were missing on my oldest backup too.

This new development meant I needed to guess eight hex digits, 32 bits of randomness. Quite an improvement, although I had to read the source to see exactly how it was used to see if I could search each byte independently. I suspected that I would be able to.

I could test my theories by creating a known-plaintext file, noting then removing the .pvect_whatever file, and then conducting a search for the proper (known) value. This might be faster than understanding CFS’s code.

On 9 Jan 2004, I wrote a little program to find files without .pvect_ files. There were 19, and had been exactly 19 for a while. Not too bad. I had about 500 files in CFS.

I also modified ccat to the extent necessary to work with new-style dirs. It wasn’t pretty, but it worked. Then I modified ccat to accept both an IV and a passphrase.

I wrote a program that tries IVs with each byte ranging from 0 to 255. For each of the 256 possibilities, it runs ccat with an IV of that value, repeated (all bytes equal). Then it does a simple frequency count for offsets of 0, 1, 2, and 3 into the file. After doing this it dumps all the information using Perl’s Data::Dumper. My plan was to analyze this output later using statistical tests. Estimates of the time to do this were around 38 minutes, which turned out to be surprisingly accurate.

Thinking back, I do think I recall having seen symlinks pointing to garbage in the lost+found; I must have decided they were deletable and didn’t restore them. All this work due to that miscalculation! Oh well, I had learned quite a bit about CFS.

I then noticed that the IV was 8 bytes in ASCII-encoded hexadecimal, which is a convenient way to store a 32-bit value. However, it was not converted into binary before use; the ASCII values were used instead! So basically you had a 64-bit IV, but it could only hold 2^{32} different values. Thus, my program should have really calculated frequency counts for offsets of 0–7 into the file (not 0–3):

```
i[0] = D_p'[0] xor D_p[0] xor i'[0]
i[1] = D_p'[1] xor D_p[1] xor i'[1]
…
i[7] = D_p'[7] xor D_p[7] xor i'[7]
```

It's worth noting here, as a sidebar, that the more I knew about the file in question, the better I could narrow down the possibilities for the IV. If I had known nothing about the file, if it had been truly random data, I wouldn't have been able to tell whether one IV was better than another. The more I knew about the file's lack of randomness, though, the better I could weed out all the irrelevant IVs.

This knowledge comes from knowing the names of the files, remembering their contents, that kind of stuff. One thing about the IVs is that they don't affect the high bits of the file. That is, since the IV is all in the ASCII range of 0–127, being XORed with the file will not change the high bits. That is unfortunate, as the files typically don't have high bit set anywhere. However, this makes sense from a cryptographic perspective of trying to hide similarities between files; you are modifying bits that people care about instead of ones that are typically not used.

Taking all of this into consideration, I wrote a program (smart_iv) that very quickly deduced the missing IV. The basic idea behind this was run ccat once and break its output down by which byte of the IV it is being XORed against (forming 8 bins). Then count the frequencies of the characters in that bin. The whole data structure is essentially an 8 x 256 array. It then tests for the IV that gives me the greatest number of space characters. One could map the frequency counts based on the IV being tested, but I found it easier to go the other way: that is, to XOR the space character with the IV in question and look at the occurrences (frequency) of that entry in the frequency chart. This program was capable of recovering the longer files of English text, probably about five of the 19. I then renamed this program (find_blank) because I had some better ideas.

I changed ccat to accept an IV, but what IV do I specify? Ideally, I would use a null vector. That is, if I let i'=0, then it drops out of the equations mentioned above because XOR with 0 is an identity operation. In fact, cfsd probably assumes i'=0 when there are no .pvect_ files. However, ccat uses C string functions to parse the IV from the command line, and eight nulls would be read as a zero-length string. So, instead, I specified "00000000" and dealt with the XOR deltas between that and the actual IV instead of with absolute values (that is, they are relative to 0x3030303030303030 since ASCII for "0" is 0x30).

Around this time I noticed that there were other files which had .pvect_ files but that they were also corrupted. Is there no end to this pain? I would have to examine every file on the encrypted file system, by hand, to determine whether it looked reasonable. Argh! I put this off until I had completed recovering the 19 files without .pvect_ symlinks.

11 Jan 2004

I noticed that there was a class of files that were not English prose, and therefore did not have the prevalence of the space character in their frequency profiles. So I wrote another program, called maximize_printables, which found the IVs that maximized

1. Theoretically, minimize_nonprintables and maximize_printables should give the same results since they are complementary. However, my implementation of one or the other must have been in error since I think I noticed different results.

the number of printable characters. This program did not work as well as I expected, since some IVs that maximized printables also contained junk like vertical tabs (!), which virtually never appear in ASCII files.

The next logical step was another program, called minimize_nonprintables, which did what you'd expect. However, unlike the maximizing spaces heuristic, which tended to pick only a handful of IVs, the minimize_nonprintables generally printed out several (on the order of 16 or so). I found that since it usually got half of the bytes in the IV correct, I could try one possibility, with ccat, see halves of words that I recognized, then exploit these inter-byte dependencies to infer the correct values of the currently incorrect IV bytes. I was able to recover around five more of the 19 files without .pvect_ files.[1]

Finding those halves of words in the plaintext gave me an idea. Perhaps I should write another program, for files containing a known string. For example, an RCS file has the words "head", "access", "symbols"—surely this knowledge could be helpful in determining an IV!

Again, the more I know about the plaintext, the easier it is to recover an IV. For simplification, consider the case of an RCS file. In this case, I know it begins with "head\t". In most cases I never increment the major number, so the next characters are usually "1.", which is followed by one or more digits. If I know that a fixed string of length l occurs at a particular offset into the file, I can easily compute l bytes of the IV starting at that offset modulo 8 and wrap around in the obvious fashion.

12 Jan 2004

It's worth noting that this applies to other types of files too; for scripts, I start with "#! /". The extra space is there for portability because Dynix uses a four-byte "magic number" to identify scripts. I never expect to see Dynix, but it's only one byte and portability is not a bad habit to have. In many cases the following letters are "usr/" too.

I like to start from simple and specific cases, and enhance scripts to handle general cases. So the next program I wrote, known_header, cracked an IV based on known plaintext at the beginning of the file. In this case, I simply XORed "0" and the known byte and the byte from the decrypted file, and that gave me the IV. Simplicity itself! This recovered about five of the 19 files. The rest of the 19 were automatically generated files which I simply removed, so I consider these techniques moderately successful.

But let's say I'm looking for a known string of some length l in the plaintext at an unknown location. If the IV were fixed, I would look for one of eight patterns, with the pattern being dependent on my offset into the file (modulo 8 of course). At each step I might have to look ahead as far as *l*–1 bytes, to see if I was at the beginning of a match. That might be acceptable, but the IV isn't fixed. What I'm really doing is trying to find the IV that induces the most occurrences of that string, and I'm trying to do it without iterating over all possible IV values (2^32 iterations over the length of the file sounds like too much computation). Surely there has to be a better way.

14 Jan 2004

To count strings of length *l*, I thought of an (*l*+1) state DFA with multiple "tokens" that move around to keep track of the state. Specifically, there will be 2^32 logical tokens, but in implementation I'll consolidate them into *l*+1 tokens, one for each state.

Every time a logical token hits the final state, it increments a counter associated with its IV. In actuality, there may be multiple logical tokens for a given IV. For example, if the IVs were only two bytes long, and the string we want to find is ABABC and we read in ABAB, then there will be two tokens, one on the first B and one on the second. Note that the string must be longer than the IV to have two logical tokens in the works.

To simplify the matter, I modified ccat to accept a null IV. I merely tested to see if it was a null string and, as a special case if it was, accepted it as the null vector (which would be impossible to pass on a command line anyway). That helped a bit.

17 Jan 2004

It's interesting to note that in some cases I might know two different types of things about a file: for example, that space is the most frequent character and that it contains a given string. Each of these individually induces a distribution on the IV. However, it is more difficult to represent and combine this distribution between these two programs than achieving the same level of confidence using one technique alone. That is, it is easier to print out the most probable IV than it is to communicate the top 10 IVs. And even that is easier than communicating all IVs and their associated probabilities.

19 Jan 2004

I've written known_strings. I started by examining all the relevant finite-state automata (FSA) classes on CPAN; none really fit the bill. I then wrote the main body of the program, proceeding as though I had already written any relevant classes. I knew that I'd need some kind of custom FSA and that I'd need some structure in which to store results.

Representing the results of the searching posed an interesting challenge. Did I want to allocate a Perl structure with 2^32 entries? Even at a byte per entry, that's the entire address space of the machine! I could do it with a large file, but that lacked elegance. If the known string were short, there would be many matching IVs and the program would be quite slow.

One alternative would seem to be storing only non-zero entries, but this is only useful if the resulting array is sparse. The sparseness of this array depends on the length of the string I'm seeking and the content of the file. Another alternative would be to represent the partial matches of IVs, and after parsing the file, to loop through all IVs to see how many of these partial matches a specific IV actually matches. Put another way, my results might be "any IV that starts with A" and "any IV that starts with AB". I would then enumerate all IVs, starting with AAAAAAAA, noting that it matches once and so outputting "AAAAAAAA 1". When I got to ABAAAAAA, which matches twice, I'd output "ABAAAAAA 2". Thus, instead of storing the results array in memory, I could actually store it temporally by walking through IVs and dynamically generating the values it would have. Of course, if I spent 1 ms on every IV, iterating through them all would take 49 days.

In this light, perhaps it would be better to iterate over the file than to deal with the whole IV space. Most of the files were small text files. In fact, most UNIX files are small; for this reason the file system is optimized for dealing with relatively small files.

And that's the current situation. I took a little detour investigating alternatives to CFS and will let you know in the future how it all ends up.