20 October 2001. From Pawel Krawczyk:

I've successfully retrieved the DRM2 archive from Freenet using the following keys:

> freenet:KSK@msdrm2
> freenet:CHK@xK7PG6G1WplhGrXAsWZsW~o7WYcOAwE,xJ~Iz735FtTNKjgTykoJGw

18 October 2001: Thanks to Anonymous, Zipped file of source code associated with the DRM message below, additional commentary, FreeMe.exe for executing the program, and the author's reason for releasing the material in Readme below:

> http://cryptome.org/FreeMe.zip (93KB)
> http://jya.com/FreeMe.zip
> http://216.167.120.50/FreeMe.zip
> http://www.nunce.org/FreeMe.zip
> http://www.typo.co.il/msdrm2/FreeMe.zip

18 October 2001. Thanks to Sam Simpson (www.samsimpson.com).

Date: Thu, 18 Oct 2001 10:18:05 -0700 (PDT)
From: Sam Simpson <simpson@samsimpson.com>
To: <jya@pipeline.com>
Subject: [Fwd: Microsoft DRM - Technical description]

Found this on sci.crypt today. It's an interesting piece of research which will most likely be stomped soon by MS using DMCA or similar.

-------- Original Message --------

Subject: Microsoft DRM - Technical description
Date: 18 Oct 2001 16:06:50 -0000
From: Anonymous <anonymous@anonymous.poster.com>

http://www.binaries.net

Newsgroups: sci.crypt

-----BEGIN PGP SIGNED MESSAGE-----

Microsoft's Digital Rights Management Scheme - Technical Details

By "Beale Screamer"

This document describes version 2 of the Microsoft Digital Rights Management (MS-DRM), as applied to audio (.wma files). The sources for this material are varied, and some of the information might be slightly incomplete; however, the fundamental ideas are solid and easily verified. There is no attempt at describing the older version 1 of DRM. While version 1 is widely used (probably more widely than version 2!), and the scheme is somewhat simpler, the purpose of this is to describe the latest technology and not necessarily allow all existing systems to be broken. The ideas described here are also implemented in the software originally distributed with this document (but as an independent piece, so the software may or may not be available from where you have obtained this document), so a real implementation can be examined. Not all of the information here is needed in order to write the software that removes the encryption, but some of the more interesting points surrounding the MS-DRM scheme and software are given even if not necessary.

Also note that no code is included in this document, either real code or pseudo-code. All that's in this document is a straight mathematical discussion, which should be fully protected under the 1st Amendment to the U.S. Constitution. I have no doubt that the corporate entities that this document offends will attempt to suppress it, but I don't think any argument they make could hold up to Constitutional scrutiny.

The basic components of MS-DRM involve use of elliptic curve cryptography (ECC) for public key cryptography, DES for a block cipher, RC4 for a stream cipher, and SHA-1 for a hash function. There is also a block cipher which I haven't seen before, used in the MS-DRM system to build a MAC, or keyed hash function. This cipher will be explained completely below, and while the remaining algorithms are well-known, more will be said about Microsoft's use of ECC below.

In the discussion and examples below, all numbers are expressed in hexadecimal in the standard ordering (most-to-least significant) unless otherwise stated. The actual bytes comprising large numbers in any code are stored little endian, so at times it is convenient to look at data in that ordering, and this will be clearly marked when it is done.

One confusing item is that binary data sent back and forth is encoded using Base64, but not using the standard algorithm! For some reason, Microsoft has decided to use the non-alphanumeric character '*' instead of '/', and '!' instead of '+' in some places, and in other places they replace '/' with '@' and '!' with '%'. This means that any software dealing with these strings cannot use a standard Base64 decoder, but must use a custom-build decoder.

FILES INVOLVED

Several key DLLs are kept in \windows\system that relate to the MS-DRM scheme.

drmv2clt.dll: Provides basic DRM version 2 functionality

blackbox.dll: Provides basic, machine-specific crypto for MS-DRM. Functionality replaced by IndivBox.key when the local system has been "individualized."

The other interesting place for files is in \windows\All Users\DRM (this location is not necessarily fixed but comes from the registry entry HKEY_LOCAL_MACHINE\Software\Microsoft\DRM\DataPath). Here's a sampling of some of the files in this directory (these are hidden system files, so be sure to turn on "view all files" in order to see them!):

IndivBox.key: Despite the extension, this is really a DLL that is an "individualized" version of blackbox.dll

drmv2.lic: The file of licenses (a structured IStorage file)

drmv2.sst: "Secure state" for each of the licenses. Also an IStorage file, but each stream is RC4 encrypted.

v2ks.bla: The version 2 "key store" - this is where all the public/private keys are kept (encrypted, of course!).

v2ksndv.bla: The individualized version 2 "key store."

A SIMPLE BLOCK CIPHER (MULTISWAP)

Microsoft is using a very simple block cipher to create a message authentication code (MAC). As this is not a standard algorithm, I will describe it fully. The main operations in this cipher are 32-bit multiplications and swaps of the two halves of 32-bit words, so I have called this cipher the "MultiSwap" cipher.

The MultiSwap cipher works on 64-bit blocks, using a key that consists of 12 32-bit words, and a current state (or initialization vector) that is 64-bits long. In the Microsoft implementation, the least significant bits of all 12 words are set to 1, although once the cipher is understood it is clear that really only 10 of the words require this bit to be set. The basic operation of the cipher is a transformation that is done on the first 32-bit word of the plaintext block using the first 6 key words, and then repeated for the other plaintext word and remaining key words.

Let k[0], k[1], k[2], k[3], k[4], and k[5] denote the first 6 key

words, and let s[0] and s[1] denote the two words of the state. To transform a 32-bit input word x, first define the following function

f(a)=swap(swap(swap(swap(swap(a*k[0])*k[1])*k[2])*k[3])*k[4]) + k[5]

where * is multiplication modulo 2^(32), and "swap" is an operation which exchanges the two 16-bit halves of a 32-bit word. The complete transformation of a 32-bit word x then consists of

s[1]=s[1]+f(x+s[0])

and

s[0]=f(x+s[0]).

This is first done with the value x set to the first 32 bits of the input, and then repeated with x set to the second 32 bits and the using keys k[6] through k[11]. The output of the block cipher is the new state s[0] and s[1].

The reason this block cipher can be inverted is because all the key words are odd, which means they have multiplicative inverses modulo 2^(32). To invert f(), just do the operations in the reverse order: first subtract off k[5], then do the multiply/swap operations with the inverses of k[4] through k[0]. Notice that only the multiplicative key words really need to be odd, so there is no reason for the least significant bit of k[5] or k[11] to be set; however, Microsoft sets these bits anyway.

This block cipher is never used for encryption, but is used to create a message authentication code (MAC) in the standard way. Assuming the length of the message to be hashed is a multiple of 8 bytes (64 bits), the cipher is initialized with a state of all zeros, and then used to encrypt the entire data. The output of the last block (the final state) is the MAC for that message. This is used in computing packet keys to encrypt protected content by MS-DRM, as will be explained later.

ELLIPTIC CURVE CRYPTOGRAPHY

For ECC, Microsoft is using an elliptic curve over Zp, where p is a 160 bit prime number (given below). The curve consists of the points that lie on the curve y^2=x^3+ax+b, where the operations are done over the field Zp and a and b are coefficients that are given below.

All values are represented as packed binary values: in other words, a single value over Zp is encoded simply as 20 bytes, stored in little endian order. A point on the elliptic curve is therefore a 40 byte block, which consists of two 20 byte little endian values (the x coordinate followed by the y coordinate). Here are the parameters for the elliptic curve used in MS-DRM:

p (modulus): 89abcdef012345672718281831415926141424f7

coefficient a: 37a5abccd277bce87632ff3d4780c009ebe41497

coefficient b: 0dd8dabf725e2f3228e85f1ad78fdedf9328239e

generator x: 8723947fd6a3a1e53510c07dba38daf0109fa120

generator y: 445744911075522d8c3c5856d4ed7acda379936f

Order of curve: 89abcdef012345672716b26eec14904428c2a675

These constants are fixed, and used by all parties in the MS-DRM system. The "nerd appeal" of the modulus is high when you see this number in hexadecimal: it includes counting in the hexadecimal, as well as the digits of fundamental constants e, pi, and sqrt(2).

In order to use this public key system, any user must have a private/public key pair. Since the security of the system relies pretty heavily on the private keys remaining secret (even from the user of the system on which they reside), they are carefully hidden.

In fact, there are keys hidden in various files that are used, including blackbox.dll, v2ks.bla, and IndivBox.key. For example, once the player has been individualized, IndivBox.key is created, and there are at least two keys embedded into this file: a 64-bit key used for RC4, and a 160-bit private key for use in ECC. The ECC private key is used as the basic client key (the corresponding public key is stored unencrypted in the key store file, and used as the initial part of the "client id" sent when requesting a license), and additional key pairs are stored in part of the keystore file (v2ks.bla or v2ksndv.bla), encrypted with the RC4 key.

These secret keys are stored in linked lists that contain 32 bits per node (so the key as a whole is not in contiguous memory), interspersed with the code in the library (IndivBox.key for example). The idea is that they can be read by that library, used internally by that library, and never communicated outside the library. Since the IndivBox.key file is shuffled in a random way for each client, these keys would be extremely difficult to extract from the file itself. Fortunately, we don't have to: these keys are part of the object state that is maintained by this library, and since the offset within this object of these secret keys is known, we can let the library itself extract the secret keys! The code for this simply loads up the "black box" library, has it initialize an instance of the object, and then reads the keys right out of that object. This is clearly a weakness in the code which can be corrected by the DRM software fairly easily, but for now it is the basis of our exploit.

GETTING A LICENSE

Each protected media file is encrypted with a "content key" that will unlock the packets of the media stream. We describe briefly how a license (containing a content key) is obtained for information purposes, but the license acquisition protocol is not really important for unlocking that content. Simply use the MS Media player, have it request and decrypt the licenses, store them in drmv2.lic, and then we can extract them directly from that file.

A protected media file is apparently recognized by the presence of a DRMV2 object in the .wma file header. This object has GUID 298ae614-2622-4c17-b935-dae07ee9289c, and contains an XML object 6 bytes into the data part of the object. Among other things, this header contains a "KID" element identifying the key used to unlock the content. The drmv2.lic file is then checked to see if a license with this KID exists locally. If the license doesn't exist, a license request is formed, which sends an encrypted "client id" to the license server. This is sent as a "challenge," which consists of 168 bytes in the MS-Base64 encoding. The first 80 bytes are two ECC points, which make up an ECC encrypted random session key, and the remaining 88 bytes are the "client id" encrypted using RC4 and the session key. The ECC encryption is done using a public key that seems to be fixed for all clients, so it is safe to assume that this corresponds to a private key that is common to all license servers and built in to that side of the system (without access to the server side code, it was impossible to find the corresponding private key).

After some interaction, the license comes back as mime type application/x-drm-v2, as an escaped XML-encoded license in the following format

<LICENSERESPONSE><LICENSE version="x.x.x.x">

...base64 encoded license... </LICENSE></LICENSERESPONSE>

where x.x.x.x is most likely "2.0.0.0". To make things tricky for a sniffer, the license is actually RC4 encrypted using the same session key that was established by the client when sending the challenge. The client then decrypts the license and stores it in the drmv2.lic file.

GETTING THE CONTENT KEY

Getting the content key from a license is pretty easy once the client knows what its public/private key pairs are, and has a copy of the license obtained from drmv2.lic. The license entry is an XML object with an element for "ENABLINGBITS", which has sub-elements ALGORITHM (which should have type "MSDRM"), PUBKEY, VALUE, and SIGNATURE. The PUBKEY element should match one of the client's public keys, or else there a problem! The VALUE element is the ECC-encrypted content key, which can be decrypted by the private key that corresponds to the given PUBKEY.

The content key has a specific format: the y coordinate is ignored, and when the x coordinate is written in storage order (little endian), the first byte is the length of the content key (which may always be 7), which is followed by that many bytes of the content key. While the content key is tied to the encoded media file (which may be common to many

users), the enabling bits value will be different for each user, and tied to that user's public/private keys. Because of this, licenses are not transferable from one user to another, even though the media files themselves are (the new user must obtain his own license from the license server).

We go through an example now of finding a content key. In this example, we have identified our public and private keys as the following values:

Public key x: 1957f96f3327a25bba52166ad7fcc74087b9734b

Public key y: 8939e1b1ed988182d34d17ebbcb0e03a82d062e7

Private key: 757ff01b853496452eea0b0646c3a357a6f33509

We're looking at a file RIAALuvsMe.wma, and find in the header the following bit of XML:

<KID>nA67jM7dNGIUQIkP5v7hSQ==</KID>

The actual KID seems to be a Base64 encoding of a GUID, but it is treated as a string (uninterpreted) by the software, so the origin doesn't seem to make much difference.

The license is inside the drmv2.lic file, which is a structured "DOC file", meaning it can be accessed through the IStorage and IStream interfaces (and it can be browsed by the Microsoft Visual Studio "DOC File Viewer" tool if you're curious). The top level drmv2.lic file has a lower-level IStorage object for each KID, which can contain a set of licenses for each KID. In order to guarantee valid IStorage names, the KID is first processed to change all '/' characters to '@', and all '!' to '%'. The names of the IStream objects containing the licenses again look like Base64 encoded GUIDs, which turn out to be the LID (license ID?) element stored in the license. This can be verified once the license is obtained, but we're not sure how to generate LID's from the content header information, and so can't directly open the appropriate LID stream. Instead, we simply enumerate through all available streams for this KID, testing each one for a PUBKEY element (see below) that we know. This is taken to be the license for this content. While this is really just a guess as to the proper workings, it seems to work fine in all our tests.

Inside the license we find the following XML (this has been formatted so that it's easier to look at - in the actual file this would all be on one line).

<ENABLINGBITS>

<ALGORITHM type="MSDRM"></ALGORITHM>

<PUBKEY type="machine">

S3O5h0DH*NdqFlK6W6InM2*5VxnnYtCCOuCwvOsXTdOCgZjtseE5iQ==

</PUBKEY>

<VALUE>

VEsbPedfwrybrpkg0fhoOfe5eB9ef0R7QTxgX7NbtMIFK!h*4Pk7ek

PUqlDIRqYwQkgCGE0r0qtQdCUYszT!b7XedCIpsApQjstaFmafahM=

</VALUE>

<SIGNATURE>

KpxCm6lSXH8dTPI359jToftSEuLiP9v*zpHAy!kDEhlYkw6mkfQzlg==

</SIGNATURE>

</ENABLINGBITS>

The SIGNATURE element above is just random garbage. We didn't make a real signature for this example (among other things, we don't have a certified public key, which would have to follow this in a real license. Requiring such a signature keeps people from creating their own licenses, since only those that have been issued valid certificates can do so).

First look at the PUBKEY part. If this is run through a Base64 decoder (modified for the MS character set as described earlier) you get the following binary values, shown below as a memory dump:

0000: 4B 73 B9 87 40 C7 FC D7 6A 16 52 BA 5B A2 27 33

0010: 6F F9 57 19 E7 62 D0 82 3A E0 B0 BC EB 17 4D D3

0020: 82 81 98 ED B1 E1 39 89

Notice how this is exactly our public key from above, stored in little endian order! So this license is for our machine.

Next, take the VALUE element above, run it through a Base64 decoder, and interpret the 80 byte result as 4 20-byte values stored in little endian order. These four numbers are as follows:

Encrypted

u.x: & 1f78b9f73968f8d12099ae9bbcc25fe73d1b4b54

u.y: & 7a3bf9e07fe82b05c2b45bb35f603c417b447f5e

v.x: & 18257450abd22b4d1802484230a646c850aad443

v.y: & 136a9f66165acb8e500ab0292274deb56ffe34b3

To decrypt this value, first we multiply the point u by our private key, resulting in the point

x: 399c72d525a9b65b7543a3e3adc88ce0f6a38db5

y: 66cfa6bdbfbb93b906b22deb36792363d8e8adc2

and then subtract this point from v to get

x: c91590616b4b3707

y: 753e24e50d437e147b4998376f163dc27b639a7a

Since x is so short, we have almost certainly gotten our content key. Writing in storage order (little endian), x is

0000: 07 37 4B 6B 61 90 15 C9

which means that the content key has length 7 (from the first byte), and the actual key is the string of bytes

374B6B619015C9.

DECRYPTING THE CONTENT

The content encryption process is simpler to explain than decryption, so we start with that. The content key is not used directly, but is processed for several different uses. First, the content key is hashed using the SHA-1 hashing algorithm, producing a 20-bit output. The first 12 bytes of this output are used as an RC4 key, and a block of 16 words (or 64 bytes) of zeros is encrypted. The least significant bits of the first 12 words of this output are all set, and are used as the MultiSwap key. The next 2 words are the encryption in-whitening mask and the next 2 words are the encryption out-whitening mask (this will be explained later). The last 8 bytes of the original SHA-1 hash output are used as a DES key.

To encrypt the content so that packets can be accessed randomly (for seeks), the content cannot be encrypted as one single stream. However, to strengthen the cipher we also don't want to re-use the same key for every packet. To satisfy both of these goals, MS-DRM uses the following scheme to encrypt a packet: First, the packet (with size rounded

down to a multiple of 8 bytes) is run through MultiSwapMAC to produce a 64-bit MAC. For some reason, the 32-bit halves of this MAC are swapped before further processing. Next, the entire packet is RC4 encrypted using the swapped MAC as an 8-byte RC4 key. The 8-byte MAC (with swapped halves) is then run through a "whitened DES" by first XORing with the in-whitening mask, then running through DES (using the DES key from the last paragraph), and then XORing the result with the out-whitening mask. The resulting 8 bytes are then placed in the final encrypted packet, overwriting the last full 8-byte block (not the last 8 bytes of the packet, but blocking the packet from the beginning into 8 byte pieces, and overwriting the last full piece).

To decrypt such a packet, first locate the last full 8-byte block, run it through the whitened DES decryption, and the result is used as an RC4 key to decrypt the packet. This will produce the correct packet except for 8 bytes: those in the position of the last full 8-byte block are wrong, since they were overwritten in the last phase of the encryption. However, by swapping the halves of the RC4 key, we have the MAC for the original packet up to and including the original bytes in this position. Since the MAC is actually created out of a block cipher, we can recover the original 8 bytes as follows: run the entire packet through MultiSwapMAC up to the block in question, but not including it. This output is the next-to-last state seen by MultiSwapMAC in the encryption MAC computation, and we just recovered the final output of the MAC, so we can put these two values into the block cipher decryption to obtain the original data of this 8-byte block. The original 8-bytes are placed back into the packet, and now the entire original contents are restored!

This is a pretty clever scheme: by using a MAC constructed from a block cipher, individual packet keys can be computed and encoded into the encrypted packet with absolutely no increase in space, and since the size is maintained nothing in the structure of the content file (describing packet sizes or other parameters) needs to be changed at all. The encrypted content can be completely transparent to applications that deal with .wma files in a non-content-sensitive manner.

We finish this section showing an example of content decryption, using the content key from the previous paragraph. We first process the content key by running it through SHA-1 to obtain

15 CB 92 F9 97 2E C8 75 29 4F 12 65 36 B6 C6 DB AC A2 40 35

The first 12 bytes are used as an RC4 key to encrypt a block of all zeros, giving

0000: 80 0A 2D 48 D1 FD 7E ED 83 69 4A 7D A5 D5 EE C4

0010: 4E E1 64 52 D1 71 98 26 9A F3 14 E3 51 C8 B6 92

0020: D4 93 E4 57 97 6D 63 EF 0E 06 07 54 F7 DD ED 38

0030: E8 CA A0 D0 83 13 F1 DB C1 70 AE 56 61 7D FB 94

The first 48 bytes are interpreted as 12 32-bit words, stored little endian, and are saved as the MultiSwap key after setting all least significant bits. So the key values are k[0]=0x482D0A81, k[1]=0xED7EFDD1, etc. The last 16 bytes of the RC4-encrypted block are the whitening masks for DES, and the key for DES is given as the last 8 bytes of the SHA-1 hash value.

Assume we get a packet with size 1450 bytes, which is 181 8-byte blocks followed by 2 additional bytes. Numbering the byte positions as 0 through 1449, we look at the bytes in positions 1440 through 1447 (the last full 8-byte block), and find that they are

A8 49 65 36 A2 33 18 09

XORing with the encryption out-whitening mask (the last 8 bytes from the RC4-encrypted block above) we get

69 39 CB 60 C3 4E E3 9D

and decrypting with DES using key 36 B6 C6 DB AC A2 40 35 gives

FD EF 98 7D 8B 77 72 FD

and finally XORing with the encryption in-whitening mask (the next-to-last 8-byte piece in the RC4-encrypted block above) gives

15 25 38 AD 08 64 83 26

This is then used as an RC4 key to decrypt the packet. To replace bytes at positions 1440 through 1447 with the correct values, take the RC4 value and swap the words around to get

08 64 83 26 15 25 38 AD

This is the MultiSwapMAC of the input packet using bytes 0 through 1447. We run bytes 0 through 1439 through MultiSwapMAC to get

D9 F7 D9 53 A9 6E 14 D9

and then use this as the state input, along with the original MAC output as the data input to the MultiSwapDecode function to obtain

DA 05 D8 EB 97 FE 1E 7B

These 8 bytes are placed in positions 1440 through 1447, and then the entire original packet is restored.

OTHER ISSUES

Communication between different DLL modules is encrypted and checked at multiple points. This works roughly as follows: Objects are initialized with communication parameters by sending certified public keys to the object you want to communicate with. The second object verifies these certificates, generates a random session key (which it uses to generate a MultiSwap key in addition to the use as a session key), and sends the encrypted session key back to the calling object. Future "sensitive communication" is RC4 encrypted with the session key, and run through MultiSwapMAC to verify integrity (after padding with zeros to make the data a multiple of 64 bits). This is done for data sent both to and from the object.

Presumably, this is done so that anyone monitoring parameters passed between DLL modules wouldn't see any "sensitive data," although its use for this purpose is pretty limited. However, it does lead to some interesting and strange situations: when blackbox is sent a packet to decrypt, it decrypts it, and then immediately *re-encrypts* it using the session key to send it back to the media player! So in decrypting a packet, the computer actually goes through a decrypt/encrypt/decrypt sequence of operations!

One very important effect of this scheme is that Microsoft fully controls who gets to write modules that interact with the basic Microsoft media modules. Without a certified public key (and the corresponding private key) it is impossible to write a compatible DLL that interfaces with their code. Since Microsoft controls the issuing of certified public keys, they also have complete control over who is allowed to make compatible and competing products. Microsoft's reputation for being generous to competitors is well-known, so this effectively gives Microsoft a technically guaranteed monopoly power.

Of course, these certificates and private keys must be distributed with any "Microsoft blessed" software as well, and in fact exist in the media player and blackbox DLLs. They're not hard to extract, if you know where to look, but I won't give them here. They would be of limited use anyway, since Microsoft also has a "revocation list" mechanism built in to the Media player software, meaning that they can revoke any of these certificates at their whim, remotely disabling any software that depends on that certificate for communication.

-----BEGIN PGP SIGNATURE-----

Version: 2.6.2

iQCVAwUBO5qt3JCr1f2GXCalAQE8ygP9Gb4Dm0ZQ5GePjAIfMFyqYVtUNSUUfj7A
3ZLwbMwUtnRHeYDGWRJEqvJMPf4SujKHcwQL3LtefrhH7dOn6r4AyUQV6ymezpd/
AMY53ONufawU+T8YgilEe2WCDRc4Y/uDbQFZIhcPQ+H78nzFSvdj+FzQ7pKrxsIr
QWe1ZNP4xfY=

=WL0q

-----END PGP SIGNATURE-----

Readme file in:

http://cryptome.org/FreeMe.zip (93KB)
http://jya.com/FreeMe.zip
http://216.167.120.50/FreeMe.zip
http://www.nunce.org/FreeMe.zip
http://www.typo.co.il/msdrm2/FreeMe.zip

-----BEGIN PGP SIGNED MESSAGE-----

The software distributed with this README file removes content protection from any Windows Media Audio file (.wma file) that uses DRM version 2 (as implemented in Windows Media Player version 7). It has been tested under Windows 98, so may or may not work with other Media Player/OS combinations. Also be aware that many "protected content" files out there are actually DRM version 1, especially if they are older files. This software will not do anything to unprotect version 1 files.

There is another piece of software, called "unfuck", which similarly removes protections, but there is a fundamental difference in how these two pieces of software work. Unfuck works by allowing the player to unprotect and uncompress the audio, and then captures the audio samples on the way to the sound card. This software attacks the problem directly: it simply removes the encryption from the protected file, leaving everything else exactly the same. Because of this, there is no loss of quality due to uncompressing and re-compressing the content -- what you're left with is exactly the original content, just not protected. It's also much faster.

Please be aware that this software is "proof-of-concept" or "demonstration" level, not production level. While every effort has been made to make sure that it works properly, it may fail in unforeseen situations -- it has NOT been thoroughly tested! There isn't much chance that this program will screw up your system, but use at your own risk!

WARNING!!!!! I have just learned that the new Microsoft Media Player EULA includes a clause that says they can *automatically* modify the software on your system, without any confirmation from you required! In other words, they can disable your software, or force an upgrade so that FreeMe won't work, just because they feel like it. Be careful out there!

CONTENTS:

README - This file

LICENSE - Yes, a license! Read it!

Technical - Full technical description of how MS DRMv2 works

Philosophy - My philosophy on why I released this, and what's wrong with the DMCA

FreeMe.exe - The actual program

src/ - The source code

The first 4 files can possibly be widely re-distributed and mirrored without much fear of real legal worries -- however, you will almost certainly be harassed by several big and powerful companies, so keep that in mind. The last two files (the program) cause more problems: distributing these in the U.S. is almost certainly a violation of the DMCA. However, outside the U.S. should be mostly ok -- so mirror these on as many foreign sites as possible! Again, you may be harassed by big and powerful companies, and might get threatening letters from lawyers, so be prepared for that.

INSTALLATION:

There's just a single executable file "FreeMe.exe" to install. You can copy it so that it's in your executable PATH (for example, copy to directory C:\WINDOWS\COMMAND), or you can put a shortcut to it on your desktop -- see below.

USAGE:

This is a command-line program, and the best way to run it is from the command line. If it is installed, and the executable is in your PATH, all you have to do is type "freeme x.wma" at a DOS prompt in order to unprotect the file "x.wma". There is a verbose flag that you can invoke to have it print out all sorts of information as it discovers it (your public/private key pairs, KID of the file you're unprotecting, content key, etc.). For example, typing "freeme -v x.wma" unprotects the file as above, but in verbose mode. The output file will be called "Freed-x.wma", where "x.wma" is the original filename.

One problem with this being a command line utility is that many audio files have very long file names, so you'll have to put the filename in quotes in order for this to work, like so: Prompt> freeme "Billy and the Boingers - The RIAA Stole My Soul.wma"

As an alternative, you can put a shortcut to the FreeMe.exe executable on your desktop, and then can simply drag files from the file explorer to FreeMe. However, there is one big problem with this: the filename given to FreeMe is actually the short filename, so if you did this with the file above, you'd end up with an output file named something like "Freed-BILLYA~1.WMA" Unfortunately, I don't know how to fix this -- maybe someone else out there does.

SOURCE CODE:

The full source code is included in the src/ directory, but you will need a Win32 version of the OpenSSL package (and crypto library) in order to compile it.

There are some definite problems with this code, which I would suggest to people interested in improving what I've distributed: First off, a lot of things in Windows seem to be designed to be easier in C++ than in C; unfortunately, I don't know a lot about Windows programming, and never have learned or used C++ at all, so some of my code may be a little strange in its approach. Secondly, my .asf/.wma file format processing is hopelessly naive. Surely there are better ways to do this, or existing libraries to do it. Also, I don't really do XML parsing, but just a very simplistic scanning. This seems to work for every license I've seen, but using a real XML parser would make this much more robust. Finally, a nice pretty GUI would be good, but wasn't necessary for my "proof-of-concept" code, so I didn't include it.

Finally, you know that Microsoft is going to make some changes that will render my software useless. You've got the source code, so use that as a starting point to change with them.

HOW TO CONTACT ME:

Being anonymous, it's hard to give a way to contact me. However, if you have something very important to tell me, post to the sci.crypt Usenet newsgroup with a subject that includes the phrase "To Beale Screamer". My PGP key is given in this distribution, in case you need it (and I will always sign anything I distribute).

Please don't inundate the poor people in sci.crypt with a bunch of pointless comments. But I did want to give people some avenue for contacting me if absolutely necessary.

MESSAGES:

I have included messages below for specific groups of people.

Users: Please respect the uses I have intended this software for. I want to make a point with this software, and if you use it for purposes of violating copyrights, the message stands a very good chance of getting lost. Also, Microsoft is obviously going to release patches to their media player in order to get around the exploit in my software -- I think you'll be safe it you refuse to upgrade from your current version of the Windows Media Player (but see the warning above about "forced upgrades"). Unless they want to sacrifice backward compatibility, you will at least be able to work with your current (legally obtained) media files for the near future.

Microsoft: You guys have put together a pretty good piece of software. Really. The only real technical flaw is that

licenses can't be examined for their restrictions once they are obtained. My real beef is with the media publishers' use of this software, not the technology itself. However, it's easy to see where software bloat and inefficiency comes from when this code is examined: every main DLL has a separate copy of the elliptic curve and other basic crypto routines, and parameters passed back and forth between modules are encrypted giving unnecessary overhead, not to mention all the checks of the code integrity, checks for a debugger running, code encryption and decryption. Perhaps you felt this was necessary for the "security through obscurity" aspect, but I've got to tell you that this really doesn't make a bit of difference. Make lean and mean code, because the obscurity doesn't work as well as you think it does.

Also read the message below to the Justice Department!

Justice Department: Maybe this should really be addressed to the state officials, since it looks like the current U.S. administration doesn't care too much about monopoly powers being abused. But for whoever is interested, there is a very serious anti-competitive measure in this software. In particular, for various modules of the software to be used, you must supply a certified public key for communication. Guess who controls the certification of public keys? Microsoft. So if someone wants to make a competing product, which integrates well with the Windows OS, you will need to get Microsoft's permission and obtain a certificate from them. I don't know what their policy is on this, so don't know if this power will be abused or not. However, it has the potential for being a weapon Microsoft can use to knock out any competition to their products.

Artists: Don't fear new distribution methods -- embrace them. Technology is providing you the means to get your art directly to consumers, avoiding the big record companies. They want a piece of the action for YOUR creativity, and you don't need to let them in on it any more. Your fans will treat you nicely, unless you treat your fans poorly (take note of that Lars). Bo Diddley didn't have anything to fear from his fans, but a lot to fear from Leonard Chess. Think about that.

Publishers: Give us more options, not fewer. If you try to take away our current rights, and dictate to us what we may or may not do, you're going to get a lot of resistance. You better find a way to play nicely soon, because technology is making it possible for artists to make do without you at all. Try getting some progressive thinkers into management -- current people don't seem to be able to cope with the new environment that is emerging.

- - --

Original Distribution Date: October 18, 2001

by "Beale Screamer"


```
-----BEGIN PGP SIGNATURE-----
Version: 2.6.2
iQCVAwUBO84IDZCr1f2GXCalAQGRcAP8CHkU6B42NZNiuhS/roxKJTljm36Doq+R
zrqFeO2JY9xMCMhBlYP6RRkDATdlMWNj/U3DLXJ/lBJYUeSwMT3vsUTUHOA/lGMQ
9VqYHmAEwImnKWBNDG694abVeCFa9H/FziLLjeJQ73ADcfjr4rJ/FpHMxrtb2YfF
K5QaP3QRXl0=
=RK34
-----END PGP SIGNATURE-----
```