Patrick Gruber
A04769799 (p_g149)

Cracking DES

As part of CS 4371, students receive a challenge of decrypting a PDF file, which was encrypted using DES. The file was not given but had to be extracted from a user account in the security lab (named "secret.pdf.enc2"). The key is unknown, but the source code of the program that was used for the encryption is provided.

The Breakdown of my approach

Project 3 was released on November 5th. Since for a while I believed that we had to actually gain brute force access to "User25" in the lab to retrieve the file (part of project 3), I was not able to begin with the challenge directly until November 15th (at which point my group finished task 3 of project 3). I did, however, start doing some research as soon as the challenge was presented in class.

As I usually do, I went to Wikipedia (not advised for research, but usually great to get started) and had a look at the "Data Encryption Standard" (DES) [1]. What I found was that the DES uses a symmetric block cipher. Since we talked about this in class, I knew that the same key is used to encrypt blocks of data, repeatedly. Wikipedia also told me that the key is 8 bytes in size, with parity bits. The actual key is therefore only 56 bits, or 7 bytes, long. This reduces the combination size dramatically.

In class we also talked about something called "Known-Plaintext Attacks". Using this attack, we can compare the decrypted ciphertext against some string that we know has to be present. This allows us to determine if the decryption was successful, or not. Looking at the encrypted file, we can see that the original file was most likely a PDF file. The project description also showed us that PDF headers usually contain something like "%PDF-1.X", where X is a number from 0 to 6. However, this did not tell me anything more about PDF headers, so I tried to confirm it myself using python2.7:

```
$ python2.7
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> fh = open("./test.pdf", "rb")
>>> plaintext = fh.read()
>>> print(plaintext[0:8])
%PDF-1.4
```

I opened a file as a binary, read the entire file, and then printed the first 8 characters. This confirmed that the header is actually the first 8 characters (with no characters being present before this header). For my test PDF file (an arbitrary file that I had on my computer), I got the header and version "%PDF-1.4". As previously mentioned, we do not really know what version the encrypted PDF used. This means that for our plaintext attack, we are only interested in the first 7 bytes, and not 8. Meaning that the known plaintext can be reduced from "%PDF-1.4" to "%PDF-1.".

Next, I started tackling the actual DES encryption. I started off by having a look at the provided "enc2.c" program, as it was used to encrypt the original PDF file. The program is actually pretty straight forward, once we look up what each of the used "DES_*" functions do. "DES_set_odd_parity" [2] modifies the key to be in the correct form (Wikipedia told me earlier that DES uses 8 bit odd parity keys),

and “DES_set_key_checked” checks if the key is in the correct form. The important function call is “DES_ecb_encrypt”, which tells me that DES was used in ECB mode for the encryption. Now I wanted to try this myself, and see how this is done in python (see “DES_encryption.py” for my approach). Encryption seemed to work just as well in python. I was able to read an arbitrary PDF file, encrypt it using a key, decrypt it, and still get the same header as before.

Given that we have a PDF to test the decryption on, I started looking into the key space of DES. As shown in my program, DES can have any given key input. This means that we can actually come up with some pretty crazy keys (for example containing a ‘\r’). The guaranteed way to find the key, would be to brute force all characters in the ASCII table.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure 1: Characters that can be used

There are 128 characters in the ASCII system that could theoretically be used. This means that there are 128^8 combinations, which turns out to be $7.205759404 \times 10^{16}$. And according to the given “enc2.c” program, there is really no restriction on the key that can be used, as it can be executed like this: “./enc2 file_to_encrypt **key_in_hex**”, where the key needs to be a 16 character hex string (8 byte string conversion to hex). However, multiple resources have (including Wikipedia) mentioned that DES is really susceptible to brute force attacks. Again, i tried a simple script to get a feeling for how long this would take me to solve (script: “combinations.py”). When using my “DES_encryption.py” program again, but setting it up for my “combinations.py” (by using the encryption key “00000000”, and changing the code from “`iterator = 20`” to “`iterator = 48`”), I got some results:

```
$ python2.7 combinations.py
HEX-KEY: 3030303030303030 KEY: 00000000
HEADER: %PDF-1.4
HEX-KEY: 3030303030303031 KEY: 00000001
HEADER: %PDF-1.4
HEX-KEY: 3030303030303131 KEY: 00000011
HEADER: %PDF-1.4
HEX-KEY: 3030303030313131 KEY: 00000111
HEADER: %PDF-1.4
HEX-KEY: 3030303031313131 KEY: 00001111
HEADER: %PDF-1.4
HEX-KEY: 3030303131313131 KEY: 00011111
HEADER: %PDF-1.4
HEX-KEY: 3030313131313131 KEY: 00111111
HEADER: %PDF-1.4
HEX-KEY: 3031313131313131 KEY: 01111111
HEADER: %PDF-1.4
HEX-KEY: 3131313131313131 KEY: 11111111
HEADER: %PDF-1.4
HEX-KEY: 3f3f3f3f3f3f3f3f
```

We can see that either of these keys decrypts the test PDF successfully to where a header containing the "PDF" keyword was found. This was due to the fact that I started the script at the character that I knew would hit. If I, however, would let this program run over the entire ASCII character space, it would not be as easy. If we try all the possibilities from 0x14 to 0x7f (113 different characters), it took about a minute for the character on the fifth position to go from 0x14 to the beginning of the 0x16 space. This means that it would take almost an hour to try the rest of the 113 characters in the fifth position only:

```
$ time python2.7 combinations.py
HEX-KEY: 1414141416172646
Traceback (most recent call last):
  File "combinations.py", line 26, in <module>
    msg = cipher.decrypt(text)
  File "/usr/lib/python2.7/dist-packages/Crypto/Cipher/blockalgo.py", line 295, in
decrypt
    return self._cipher.decrypt(ciphertext)
KeyboardInterrupt

real   0m43.728s
user   0m43.668s
sys    0m0.056s
```

However, we haven't even started changing the first four characters yet. This means that we have to take an hour per character in the fourth position. If we take it a step further, we multiply: 1h×113×113×113×113 = 163047361 hours. That would take about 18 thousand years using a single thread, so this approach is clearly not viable. The logical next step was to Google for another solution. I went back to the Wikipedia page. This is how I found a service that was linked in the references. This pointed me to "https://crack.sh". After a bit of digging, I found out that the service is operated by Toorcon, the company behind a "hacker conference". They had apparently started this project as to enable research, and show that DES is insecure. After reading about them trying all "2^56 =

72,057,594,037,927,936" key combinations (without parity), and that they had a 100% money back guarantee (in case they couldn't find the key), I decided to give it a shot.

It turned out that they offered a known plaintext attack option for DES. All I had to do was clone one of their GitHub repositories and install some required dependencies [4]. The "impacket" module was quite hard to install. In the end, I decided to create a new Conda environment with python2.7, and then install it within that environment, by forking it from GitHub (using "pip install ."). The "./des_kpt.py" still would not work, but after reading the error, I went into "./des_ktp/__init__.py" and removed the version inclusion: "from _version import __version__", and installed "passlib" (again using "pip"). Now the command worked as expected, and I was able to run the examples that are given on the website:

```
$ source activate myenv
(myenv) $ ./des_kpt.py
Error: Missing command

des_kpt.py

    Commands (use "des_kpt.py help <command>" to see more):
      parse   -p <plaintext> -m <mask> -c <ciphertext> [-e]
      encrypt -p <plaintext> -k <key> [-i <iv>]
      decrypt -c <ciphertext> -k <key> [-i <iv>]
      kerb    -i <input>
      help    <command>

(myenv) $ ./des_kpt.py encrypt -p 0000000000000000 -k 1044ca254cddc4 -i
0123456789abcdef
                 PT = 0000000000000000
                 IV = 0123456789abcdef
              PT+IV = 0123456789abcdef
                 CT = 825f48ccfd6829f0
                  K = 1044ca254cddc4
                 KP = 1022324454667688
                  E = 1
(myenv) $ ./des_kpt.py parse -p 0123456789abcdef -m ffffffffffff0000 -c
825f48ccfd6829f0
                 PT = 0123456789ab0000
                  M = ffffffffffff0000
                 CT = 825f48ccfd6829f0
                  E = 0
crack.sh Submission = $98$ASNFZ4mrAAD///////8AAIJfSMz9aCnw
```

According to the website, the parse command can be used to create a submission. What it requires is just the plaintext, a mask (which after trying a few inputs looked like it just tells the services which bytes need to be ignored from the plaintext - in the above example this turns out to be "PT = 0123456789ab0000", using the mask "ffffffffffff0000"), as well as a ciphertext that we want to crack. I wanted to see if I could just insert the entire plaintext, as well as the entire ciphertext of the PDF, and mask the plaintext to just look for the first 8 bytes. However, the "crack.sh" documentation states that the mask can "have at most 24 zero bits." So I couldn't submit the problem this way. Next, I modified my "DES_encryption.py" script to try to only decrypt the part that we are interested in (the first 8 bytes; changing the code from "`plaintext = cipher.decrypt(msg)`" to "`plaintext = cipher.decrypt(msg[0:8])`"). I reasoned that it could work, since we are working with a block cipher. Additionally, I tried to restrict the encryption string, to validate that the output would stay the same (by

changing "`msg = cipher.encrypt(plaintext)`" to "`msg = cipher.encrypt(plaintext[0:16])`"). Again, the decryption and encryption worked just fine, and I should be able to submit the first 8 bytes from the ciphertext and plaintext to get the correct key.

Next, I had to find the actual values of the "secret.pdf.enc2" file, in order to be able to submit the brute force job. This turned out to be quite easy with python. I just had to convert the right locations within the file into a hexadecimal representation. I used "extract.py" to do this:

```
(myenv) $ python extract.py
('%PDF-1.4', '255044462d312e34')
('v\x14s\xe7\x89!\r\xc2', '761473e789210dc2')
```

From both my "test.pdf" and the "secret.pdf.enc2" I read the first 8 bytes, and converted them to a hex representation. As previously mentioned, this is not enough information for the "parse" command. We need a mask, which we'll use to ignore the exact version of the PDF (since we do not know which version was used for the encrypted file). This means that the mask "ffffffffffffff00" should work. I tried to see if this was correct using the parse command:

```
(myenv) $ ./des_kpt.py parse -p 255044462d312e34 -m ffffffffffffff00 -c
255044462d312e37
                 PT = 255044462d312e00
                  M = ffffffffffffff00
                 CT = 255044462d312e37
                  E = 0
crack.sh Submission = $98$JVBERi0xLgD/////////ACVQREYtMS43
```

The resulting job plaintext turned out to be "255044462d312e00". Using python once again:

```
(myenv) $ python
Python 2.7.15 |Anaconda, Inc.| (default, Oct 10 2018, 21:32:13)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "255044462d312e00".decode("hex")
'%PDF-1.\x00'
```

I was able to confirm that it was looking for the correct plaintext, and I had already generated the submission for "crack.sh". I went to the page [5], copied my submission token, submitted, and waited. About five minutes later, I went back to my parse command and saw that I had copied some of my variables to the incorrect spot. Previously, I had identified the plaintext, as well as the ciphertext correctly, but when inserting the values into the "parse" command, I used PDF header plaintexts for both the "-p" and the "-c" command. I'm not exactly sure where I copied the ciphertext from, but it appears that it is just another PDF header ("%PDF-1.7"). I basically asked "crack.sh" to find a key that would turn my plaintext back into a plaintext. Two days later they sent me a result:

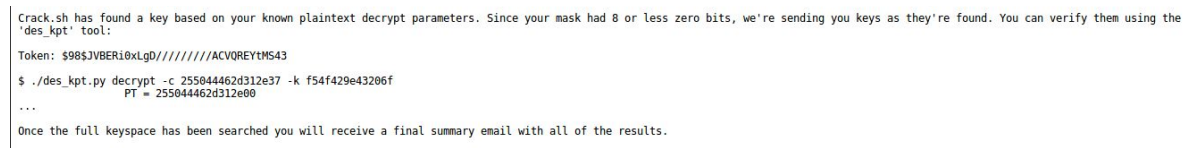

Figure 2: The first brute force result

Note that this was the only key that could be identified. While waiting for the above result, I made some changes to the submission, so it would accurately reflect the values that I am looking for:

```
(myenv) $ ./des_kpt.py parse -p 255044462d312e34 -m ffffffffffffff00 -c
761473e789210dc2
                  PT = 255044462d312e00
                   M = ffffffffffffff00
                  CT = 761473e789210dc2
                   E = 0
crack.sh Submission = $98$JVBERi0xLgD/////////AHYUc+eJIQ3C
```

Then I carefully checked for mistakes this time around, and submitted it again. This new submission had to wait in line for its turn (it took about three days before it was ready to run).

Meanwhile I was curious to see what result I had gotten from my faulty submission. So I created another file called “test_encrypt.py”, where I wanted to see what happened when I used that key during a decryption (because in theory, using this key should return the same file). I opened “test.pdf”, read it, took the key, hex decoded it, ran the encryption and wrote it to a file. The first time I ran it, I got an error “ValueError: Key must be 8 bytes long, not 7”. I decided to wait for my second submission, and see what results I get:

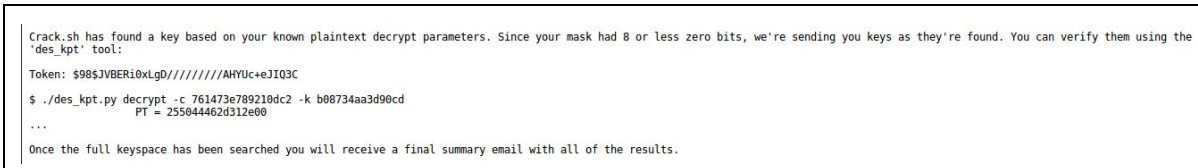

Figure 2: The second brute force result

Once again, the key was only 7 bytes long. I believed that the 7 byte key from the result must have been without parity bits, given that “crack.sh” checks all the 2^56 combinations. I went back to the “crack.sh” website and couldn’t find anything that confirmed my suspicions. Then, I went to the GitHub page that provided the code for “des_kpt.py” [6], where I found: “NOTE: Results from crack.sh are 7-byte (56-bit) keys without parity. You can use des_kpt.py encrypt or decrypt to add parity to the key if needed.” And sure enough, the command that was included in the result email worked with a 7 byte key (instead of 8). The next logical step was to see how “des_kpt” transformed the 7 byte key to 8 (adding the parity bits). I opened the source code of “des_kpt”, and traced the decrypt command back to a file called “./des_kpt/commands/DecryptCommand.py”. This file contained a function called “execute”, which took the key, passed it through “`key_parity = des.expand_des_key(key)`” and then used this key as: “`des_obj = DES.new(key_parity, DES.MODE_ECB)`”. This function could be found in the previously

required “passlib” module and is included as “`from passlib.utils import des`”. So I want back and ran my key through this function:

```
(myenv) $ python
Python 2.7.15 |Anaconda, Inc.| (default, Oct 10 2018, 21:32:13)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from passlib.utils import des
>>> des.expand_des_key("b08734aa3d90cd".decode("hex"))
'\xb0B\xcc\x94\xa2\xecB\x9a'
>>> des.expand_des_key("b08734aa3d90cd".decode("hex")).encode("hex")
'b042cc94a2ec429a'
```

Now I finally had an 8 byte key. I created the “decrypt.py”, which generated “secret.pdf”, and informed me that, this time, it all worked as expected.

The final processing time for the key was a little over a day, once “crack.sh” started processing my submission. All the other time that I had to spend on this was due to me having to figure out how to use DES in python, and how the “crack.sh” system worked. A few mistakes on my end added some time as well. If I had to decrypt another DES ciphertext, my guess would be that I would be a lot quicker. I would say that this confirms the fact that DES is not safe to use. I was able to figure this out in about three weeks, with little knowledge about DES, simply by using Google and Wikipedia.

Addendum

I went back and expanded the first key as well, since I was curious to see if it could actually work. With parity bits, “f54f429e43206f” turned out to be “f4a6d052e41880de”. I changed my “test_encrypt.py” from “`key = "f54f429e43206f00"`” to “`key = "f4a6d052e41880de"`”. As a result, I got:

```
(myenv) p@letterbomb:~/proj/sandbox/des_kpt/finals$ python test_encrypt.py
%PDF-1.4
('f4a6d052e41880de', '\xf4\xa6\xd0R\xe4\x18\x80\xde')
('\r\x13\xf5\xf2\x1d71u', '0d13f5f21d373175')
```

Where “`'\r\x13\xf5\xf2\x1d71u'`” should appear as a valid PDF header, if this operation was successful. To see what actually happened, I also changed “`enc = cipher.decrypt(plaintext)`” to “`enc = cipher.decrypt("255044462d312e37".decode("hex"))`”. Here, “255044462d312e37” is the original ciphertext that I used for the submission. Now, I got the output:

```
(myenv) $ python test_encrypt.py
%PDF-1.4
('f4a6d052e41880de', '\xf4\xa6\xd0R\xe4\x18\x80\xde')
('%PDF-1.5', '255044462d312e35')
```

The version position turned from 7 into a 5, but this only seems to work for a “%PDF-1.7” input, and not for “`%PDF-1.4`” (the previous try). This makes me think that “crack.sh” found a key that generates a valid header, by pure luck.

Sources

[1] https://en.wikipedia.org/wiki/Data_Encryption_Standard
[2] https://www.openssl.org/docs/man1.1.0/crypto/DES_set_odd_parity.html
[3] https://crypto.stackexchange.com/questions/58656/can-i-find-the-encryption-key-if-i-know-the-plain-text-and-the-encrypted-text-d
[4] https://crack.sh/des_kpt/
[5] https://crack.sh/get-cracking/
[6] https://github.com/h1kari/des_kpt