File encryption for untrusted remote file systems

and some other practical crypto problems

Vasil Kolev <vasil@ludost.net>

Cryptography

The function of cryptographic protocols is to minimize the amount of trust required.

Ferguson, Schneier, Kohno, “Cryptography Engineering”

- This talk will be as applied as possible
- i.e. almost no maths involved
- but a lot of worse stuff

What you need to know to understand this

- What are AES, RSA, SHA1
- What is a block-cipher mode
    - ECB, Counter mode, GCM, CBC
    - Initialization vector (IV)

Who are we

- pCloud
- cloud storage done right
- yadda, yadda, yadda

Trust?

- Eastern-European company with servers in the USA
- We don't like the idea to be trusted
- We shouldn't be
  - But we want to be used :)
- Requiring too much trust is detrimental in the long run

Contents of this talk

- Encrypted web-based file transfer service
- Peer-to-peer protocol
- Filesystem-level encryption

A note on cyphers/algorithms

- AES256, RSA, SHA1, SHA256
- We use what's best supported and known
- Not much choice if you want to be cross-platform

Encrypted transfer service

Encrypted transfer service

- Sources: https://github.com/pcloudcom/pcltransfer/
  - see root/js/jscommon/1540.pcrypt.js, doc/specs.txt
- Service: https://transfer.pcloud.com/

What does this service do, user's POV

- Give it a password, your email, some other people's mails, files
- Encrypts, uploads the files, sends an email to the recipients with a link
  - with an optional message
- The recipients open the link, give the password and can get the files.
- (probably) even journalists can use it

How is this done - prereq

- All in JavaScript
  - Browsers SUCK
- Using the Stanford JavaScript Crypto Library
- AES256-GCM

AES-GCM

- Wonderful construction
- Requires no padding
- AES in counter mode, with authentication data, e.g.

```
while (!eof)  {
    offset++;
    CT=encrypt_with_key(IV+offset);
    out[offset]=input[offset] ^ CT;
}
out[offset+1]=generate_auth_data();
```

Before everything else..

- Generate a salt
  - We'll be seeing a lot more of the salts later
- Unique, public value
- Stored in plaintext
- What we do in the transfer:

```
salt = sjcl.hash.sha1.hash(
    'pcloud' + new Date().getTime() +
    sjcl.random.randomWords(4) + this.opts.user_email
)
```

Key derivation

- PBKDF2 with HMAC-SHA1, 16384 times, the generated salt, for 256bit key
  - Password-based key derivation function
- Results in a key we can use
- The same password with a different salt results in a different key
- Takes ~100ms to generate, helps against brute-force attacks
  - These are user-selected passwords, which aren't very secure

Encrypting a message

- Very simple - AES256-GCM with IV=SALT

Encrypting a filename

- IV is uint32_t[4];
- memcpy(IV, SALT, 96 bits);
  - The last 32 bits are used as block counter for the GCM
- IV[0] ^= (fileno*2);
  - all the files are numbered, from 1 to N
- AES-GCM with IV

Encypting a file

- IV is uint32_t[4];
- memcpy(IV, SALT, 96 bits);
  - The last 32 bits are used as block counter for the GCM
- IV[0] ^= (fileno*2 + 1);
  - all the files are numbered, from 1 to N
- For each 1MB block B in (0..N) , do
  - IV[1] ^= B
  - AES-GCM with IV

Why is it done like this

- Salting:
  - We shouldn't leak if two files look the same
  - We shouldn't leak if two filenames look the same
    - or are the same
- 1MB file split
  - Trade-off because of the ways browsers work
- AES-GCM
  - Requires no padding
  - Gives an authentication if the file is corrupt
- Weird XORs
  - a way to guarantee difference in IVs

Peer-to-peer protocol

Peer-to-peer protocol

- Sources at https://github.com/pcloudcom/pclsync
- pp2p.c

Why a peer-to-peer protocol?

- A way to copy files directly between users
- Both sides are untrusted
- The network is easy to listen to

What we have beforehand

- SHA1 and size of the file we need to get
- an RSA keypair
  - Regenerated periodically

Asking if someone has a file

- We cannot just ask/reply for a specific checksum, it's a leak
- psync_p2p_check_download()
- Check query consists of:
  - first 3 bytes of FILESHA1=sha1() of the file
  - file size
  - some random RND1
  - sha1(FILESHA1||RND1)
- This is broadcast/multicast in the local network
- Only the size of the file can be found here

Checking if you actually have the file

- psync_p2p_check(), psync_p2p_has_file()
- For every file we have with SHA1 that starts with those 3 bytes, check if the size and the otner sha1() match.
- If we find it, we bind() to a socket and reply with
  - port
  - some random RND2
  - sha1(FILESHA1||RND2)
- The second sha1 is just proof that we have the file
  - YES, we do check if RND1!=RND2 :)
- This looks like a good place to try MITM. . .

Requesting access to the file

- The requester asks the central service for an authentication token to be able to access the file, with its RSA key
  - psync_p2p_get_download_token()
  - This is the actual proof that we're allowed to have it
  - The token contains a signature of the RSA key
    - This is how we fight MITM

Passing the file

- Then, on a TCP connection to the port of the responder, the following is sent:
  - RSA public key
  - token
  - first 3 bytes of FILESHA1=sha1() of the file
  - RND2
  - sha1(FILESHA1||RND2)
- The responder verifies the token with the API, and if it's OK for this key sends back (in psync_p2p_tcphandler()):
  - Encrypted with the public RSA key, an AES256 key and IV
  - The file, encrypted with AES256-CTR with the key and IV
- We have and check the SHA1 for the file

Rationale

- We don't give away any information
- We make sure you can't "steal" files
- We consider the drawback of having to ask the central service for tokens and token validations acceptable
  - Trade-off - one RTT against having PKI
- Leak - if you have the SHA1 of the file you can see if anyone has it
  - All clients have an option to disable it

Encrypted file storage

Encrypted file storage

- FUSE fs on top of a remote file/object storage ("cloud")

Encrypted file storage

- Sources again at https://github.com/pcloudcom/pclsync
- mainly pcrypto.c
  - not as scary as it looks, although:

```
static int memcmp_const(const unsigned char *s1,
const unsigned char *s2, size_t cnt)
{
  size_t i;
  uint32_t r;
  r=0;
  for (i=0; i<cnt; i++)
    r|=s1[i]^s2[i];
  return (((r-1)>>8)&1)^1;
}
```

. . . or, if you like puzzles

spot the error: (pcypto.c:420)

```
revsize=0;
for (i=0; i<3; i++){
  b=!!((revisionid>>(i*8))&0xff);
  revsize=(revsize&(b-1))+b*(i+1);
}
```

Why not just AES-GCM again?

$$P1 \oplus P2 = encrypt(P1) \oplus encrypt(P2)$$

- e.g. it leaks like a sieve
- Susceptible to replay attacks (pieces of old file in the new one)

Why not IEEE P1619 or similar?

> *When using transparent encryption, one must therefore address these vulnerabilities by means outside the scope of this standard.*

IEEE Std 1619-2007 on traffic analysis, replay attacks, and sector randomization

> *This audit finds that EncFS is not up to speed with modern cryptography practices.*

https://defuse.ca/audits/encfs.htm (EncFS audit)

Key storage

- One RSA keypair per user
- The user encrypts the private key with a passphrase and stores it with us
  - This is a trade-off, to make it possible to use it on more devices
  - Not strictly required

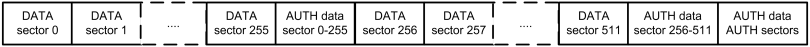
For each file and directory. . .

- we keep
  - 256bit key
  - 128bit IV
- all encrypted with the public RSA key of the user

File encryption

- The file is split in 4096-byte "sectors"
- Each sector is encrypted with AES256-CBC, with IV=authentication block for that sector
- Should be in psync_crypto_aes256_encode_sector() and psync_crypto_aes256_decode_sector(), still unfinished

Encrypted file



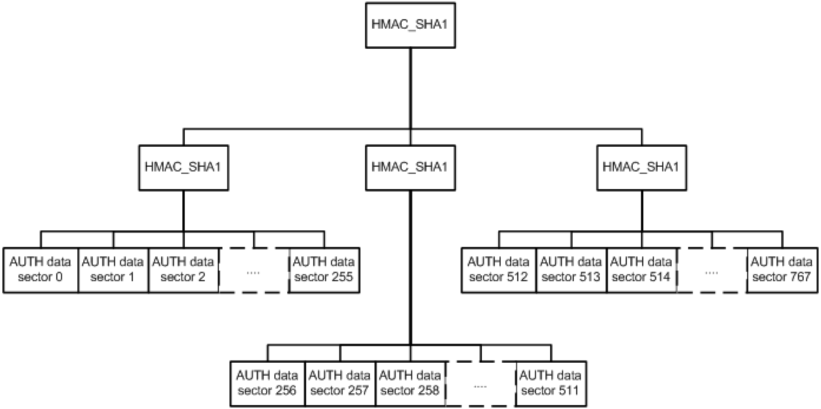
File authentication

- "authentication block"
- 128 bits (AES256 block size)
- Consists of HMAC_SHA1 and a revision number
  - written in a weird way
- HMAC_SHA1(DATA||sector_id||revision), with secret=per-file IV
- The "revision" is needed so if you encrypt A, B and then A, it doesn't leak.
- For every 256 sectors, one sector (4096 bytes) with auth data is stored, encrypted with AES256-ECB
  - This doesn't leak, as there can't be two such sectors which are the same

File authentication - hash tree

- How to ensure the integrity of the whole file
  - efficiently?
- Hash tree!
- For every sector of auth data, there's one auth block = HMAC_SHA1(DATA) with secret=per-file IV
- See picture

Hash tree


HMAC_SHA1
HMAC_SHA1
HMAC_SHA1
HMAC_SHA1
AUTH data sector 0
AUTH data sector 1
AUTH data sector 2
....
AUTH data sector 255
AUTH data sector 512
AUTH data sector 513
AUTH data sector 514
....
AUTH data sector 767
AUTH data sector 256
AUTH data sector 257
AUTH data sector 258
....
AUTH data sector 511

Efficiency of the hash tree

- One read() needs $\log_{256}$(file_size/4096)+1 reads
  - and those blocks will be cached already
- One write() requires $\log_{256}$(file_size/4096)+1 reads and writes
- Better that rewriting the whole file
- We have full file integrity

Challenges

- No leakage
- . . . but the same file in the same directory must have the same name
  - to prevent collisions

Keys, functions

- again, we have a per-directory key and IV
- see psync_crypto_aes256_encode_text() and psync_crypto_aes256_decode_text()

Filename encryption explained, 1/2

- If the filename is < AES256_BLOCK_SIZE (128bit), AES256-ECB

Filename encryption explained, 2/2

- calculate HMAC_SHA1(data_after_first_block, dir_IV)
- XOR the first block with the HMAC
- do AES256-CBC on the whole thing with the dir_IV)

Filename decryption

- do AES256-CBC on the filename with the key and IV
- calculate HMAC_SHA1(data_after_first_block, dir_IV)
- XOR the first block with the HMAC

Why do all this complicated crap?

- Two files in the same directory have the same encrypted name
- Two blocks in a filename that are the same are not the same in the encrypted name
- A repeating first block of the filename will not be the same in the encrypted name

Stuff we haven't done yet

- Integrity of the whole tree and files
- Providing other people's public keys (while sharing files)

Conclusion

- Tons of caveats and problems
- It's not easy to design something that you can't break
- It's probably impossible to design something others can't break
  - (please, please break this one)

Questions?

- Any questions?

Thank you!

- Thank you for listening
  - or not snoring too loud :)