Memory Management, with Apologies to Salvador Dali

There is considerable danger of security leaks in multiuser systems. The users do not need to be on concurrently, but this can be the case in some kind of network server.

Many years ago, I was working for a company that was leasing time from a local university (not ISU). I was having trouble with some corrupted disk I/O, so I dumped the contents of some of my buffers before I used them. The buffers complete records of many students, including a lot of confidential information.

Of course you know what happened: older versions of the student files had been deleted, but the disk contents had not been wiped, so the next allocation of those blocks gave the next user the contents. The purpose of memory, after all, is to persist.

This is still an area of potential security leak today. You may work on something confidential (like key generation, or passphrase processing) in RAM, but when multiple users are involved, your job may not even stay in RAM; it might get swapped out and put in virtual memory, which may make it accessible (depending on disk partitions, protection, and the like) to others. Wiping virtual memory would cause a huge reduction in system throughput. Caveat empt(i)or.

PGP Keys

The security of PGP depends on the keys: how they are generated and how they are managed. Begin with key generation.

PGP Key Generation

Recall that PGP uses RSA as part of its process: the session key is encrypted using Alice’s private key, and Bob recovers the session key using Alice’s public key. Where do Alice’s keys come from?

First, Alice chooses a username. The convention has been to use a nice human name followed by a valid email address in angle brackets. My mother would use Alyce Wolper <awolper@comcast.net>

Next, Alice picks a pass phrase, so that only she can use her private key. The pass phrase should be long (to protect against brute force) and include punctuation and the like to avert dictionary attacks. It needs to be easy to remember and hard to guess: “A man’s got to know his limitations.” might be OK unless everyone knows that you are a Dirty Harry freak.
[Following C. H. Lindsey, *Critique of PGP Key Generation.*] Alyce chooses a key length (longer is better; she can have more than one). Two random numbers are generated (more on this in a minute); their length is half of the key length. These are used to pick two large primes: starting at each random number, look at every fourth odd number to see if it is prime. There is room for some choice in the primality testing, but most consider the choices made to be adequately secure.

Once you have two prime numbers $P$ and $Q$, you form their product $N = PQ$. Now you can derive the RSA key in the usual way: choose public $e$ relatively prime to $\phi(n)$, and calculate private $d = e^{-1} \mod \phi(n)$. The ciphertext is $C = M^e$ (encode with the public key), and the message is recovered as $C^d = M^{de} = M^1 \mod \phi(n)$.

The security depends on factoring $\phi(n)$, and for the moment factoring is considered hard.

Any request to use the private key requires the pass phrase. The pass phrase is never stored.

---

**PGP Random Numbers**

Random Number generation is tricky, because the knowledge of one number from a deterministic sequence implies knowledge of all subsequent numbers. PGP generates the random numbers from keyboard latency: it asks you to type for a while.

In older versions (notably 2.6.3, which was the “gold standard” PGP version for many years), PGP kept a 96 word array `randPool` (each word was 32 bits). `randPool` was initialized to zero, and over the course of processing was filled with “random” bits. With each keystroke, a cycle of words was XORed into `randPool`; when `randPool` was filled, it wrapped around, so there was a possibility that a word might be XORed with itself.

The keystroke words included the character typed and a time stamp. Now, time’s arrow moves in one direction only, so subsequent time values are not completely random, since you can rule out the possibility of an earlier time appearing. An attacker who knows when you did key generation might be able to get some information about your random numbers this way.

There can also be issues with system clock resolution. For example, the UNIX `gettimeofday` system call returns the number of microseconds since the last full second, which is more “random” than straight time. The difference between successive calls goes into `randPool`.

When `randPool` wraps around, it us “stirred” by encrypting with the hash (signature generator). This is one-way encryption; you never need to recover the old `randPool`.

These random numbers seed the PRN generator that generates the session key. This makes the sequence of session keys appear random.

The session key actually depends on the message as well as the status of the random number generator.

---

**Keys and Key Authority**
Since PGP uses public key cryptography, everyone involved in the network (in the social sense) needs access to everyone else’s public keys.

First, Alice keeps a “key ring” of public keys of use to her; if she needs to PGP with Bob, she needs to keep Bob’s public key. The danger is that Eve might have substituted her key for Bob’s. In this case, Eve could read messages intended for Bob that had been encrypted with “his” public key. Eve can also forge Bob’s signature.

So how does Alice know that a key is genuine? If Bob has given her the key personally, there is no problem. But this is not always possible, or even desirable.

One method is for Dick to give Bob’s key to Alice, which works if Alice trusts Dick and has his public key. Dick creates a certificate for Bob’s key, including the key itself, time of creation, and validity period. Dick signs a hash of the certificate with his private key, proving authenticity.

Another method is to widely disseminate your public key, by using it as your signature at the end of every email message. A forger’s key would be overwhelmed by the widely circulated legitimate key, since many people would have the real key and note the difference. Any method like this reduces throughput, but in many situations (people sending html mail, for example), a few extra bytes add very little overhead.

PGP includes a key legitimacy field with each entry in a public key ring. Ultimate trust applies to your own key pair; other keys are marked by the user as unknown, untrusted, marginally trusted, or completely trusted. Signatures are also attached to a new key, and the level of trust in the owner of the signature is assigned to the signature trust value.

The actual key legitimacy is calculated from the signature trust fields in the entry. If a key is signed by many trusted owners, then it is taken to be trustworthy (the calculated trust must exceed a threshold determined by user-supplied parameters). This is the “web of trust”; see the diagram on page 456.

Finally, a user can revoke a key by issuing a key revocation certificate.