## 1 Secret Inverse Keys

The typical set of cryptographic primitives includes hashes, message authentication codes, secret keys, public key pairs. Other higher level constructs such as key negotiation protocols are built on top of them. The notation for reasoning about them can be represented as an algebra that exposes the group structure (commutativity, adjacent inverses cancel, right cancellation, etc) or emphasize the functional structure. The most obvious example is public key pairs. RSA keypairs commute, which creates some hazards to be avoided, and allows for some interesting constructs.

$$\begin{aligned} alice_{Pub} * alice_{priv} &= alice_{priv} * alice_{Pub} && (1)\\ alice_{Pub} * alice_{priv} &= 1 && (2)\\ x_{Pub} &= x_{Verify} = x_{Encrypt} && (3)\\ x_{priv} &= x_{sign} = x_{decrypt} && (4) \end{aligned}$$

Though it is almost always the case that when RSA keypairs are generated, one of them is supposed to be made public. But that is not necessarily true, because the important property is that they keys are asymmetric inverses. Some new kinds of operations become possible when it is possible for two parties to share an asymmetric keypair, while it is also impossible that either party knows both of them.

## 2 Cryptographic Authorization Enforcement

When a person is granted a keypair, an assumption is made that the person makes efforts to not leak his private key, to have some idea what he is signing, and to not leak information once decrypted. If a program is run that is known to perform authorization checks, a keypair can be generated for the program to certify that the program did the check. Notice that this is not the same thing as having a signature, because that admits that we check that the signature is invalid and proceed to use it anyway. It should behave like encryption, where it is cryptographically impossible to override the requirement. A signature will take a stream of data and perform a private RSA operation on an MD5 sum of the data. It generally will not perform an RSA operation on the data itself (a key in this case).

An example of where this is useful is when a program wants to write an encrypted file, the program will generate a random key for the cipher it will begin writing it with. Since the program made up the random key, it didn't need to decrypt anything to get the key. So it can begin writing it out now. But in order for it to be retrieved later on, we will need to write the key to a public key of somebody that is authorized to open the file later on.

But if that user has a normal RSA encryption keypair, that means that the user can write valid grants to any of the public keys he knows about. The user is effectively making his own decisions about who can access which files, without any interference from a program designed to ensure that such users are authorized by other rules that must be enforced.

## 3 Asymmetric Inverse Secret Keys

A way to deal with the problem would be to only give the user his private decryption key, and have the authorization program keep the encryption key. But the problem is that the keypair needs to be generated somewhere. If the user generates it, he knows both keys and can grant to everyone. If the program generates it, then the program can write grants for anybody it pleases. Otherwise, a trusted entity would need to distribute the keys to the program and the uploader. If we are certain that only the program has the encrypt key and the user has the decrypt key, we can cryptographically ensure that the user does the grant without the assistance of the program. This would mean that if the user is unavailable, a grant of the file cannot be made. If the authorization agent is unavailable, the user cannot go around it.

So, we can perform a key exchange that generates an RSA session key between the two entities such that each end only has one of the keys, yet they are inverses. Alice and Bob both own keypairs that are exclusively for generating these session keys. They exchange the public keys of these keypairs. Abstractly, they each compute a new asymmetric secret key like:

$$symmetricKey = (me_{priv} * you_{pub})$$

Because RSA operations not only commute for a single keypair, but also for heterogenous keypairs, we effectively blind our encryption key so that we only know one of them, yet both combined will cancel.

$$secret * (alice_{priv} * bob_{pub}) * (bob_{priv} * alice_{pub})$$

But because RSA keys are not quite just a computed number, the RSA values of interest (once already computed) are the modulus $N$, and the two inverse keys $E$ and $D$:

$$(((msg^{E})^{D})\%N) = msg$$

The application of $E$ and $D$ could be in either order, so in that sense, the distinction between public and private key isn't entirely real. Though, as an implementation detail $E$ is often a predictable value, so both $E$ and $N$ constitute the actual public part of the key. But there is a problem caused by the two sides having a different modulus $N$. So we apply the abstract asymmetric key by carefully keeping track of all the key parts we know about. It turns out that they know everything except the decrypt key of the other side: $you_D$. They both have a function to encipher data:

$$cipher(msg, me, you) = ((((msg^{me_D})\%me_N)^{you_E})\%you_N)$$

So this can be thought of as a cryptographic write pipe that guarantees that the first party wrote the data, while the second party read it. This chain can be used to ensure that users are not granted privilege to do certain things on their own, and that programs require users to be present.

## 4 Simplifying Public Key Use

If every agent has an X509 certificate (a DN, and a Verify key, and a sign operation) plus a keypair for encryption purposes (an Encrypt key, and a decrypt operation), then a lot of things become fairly simple. When one agent connects to another, they can begin with a 2 way SSL exchange to mutually identify identities. They then use their encryption keys to generate session keys for requests involving granting access.

If a user wants to give permission to another, he must ask for permission by unwrapping the grant given to him and wrapping up the grant to the authorization service, so that the authorization service can unwrap it and wrap up a grant of that key to the user. If the authorization service refuses to upload grants that it cannot unwrap, then it will not be possible for the user to grant directly to other users. And if the authorization service cannot open the grants that it wrote, then it will need to get them from users, proving that the program is not acting on its own.

Ensuring that the user and the program are both present is a common problem. For database applications, a user may need access to drop the entire database in the context of running a migration program. But the migration program or the user alone do not have such access.