CloudProxy, Go version

This document gives a brief description of the Go implementation of the Tao architecture. It should:

1. Allow the reader to draw the parallel between the current implementation and the one described in "The CloudProxy Tao for Trusted Computing" Tech report in the cloudproxy/Doc directory.
2. Describe the system components to inform the installation procedures set forth in install.md hopefully making debugging installation problems and deployment problems easier.
3. Help explain how the Go implementation discharges the isolation, policy management, key management, channel management and initialization requirements to achieve the security goals of the CloudProxy Tao.

FileProxy in Go

The "FileProxy" application described in The CloudProxy Tao for Trusted Computing Tech report (available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf, or in the Docs directory) rapidly introduces the terminology and architecture. We have reimplemented cloudproxy in Go but the FileProxy paradigm offers a convenient introduction to the Tao so we use it to explain the Go implementation.

The sample code for FileProxy (in Go) is in $(ROOT)/apps/fileproxy, where ROOT is the directory where the cloudproxy git repository (github.com/jlmucb/cloudproxy) was cloned.

As in the original version, fileproxy consists of three programs:

1. the client program (which uploads files to the server and requests files from the server) called fileclient;
2. the server program (which services requests from clients) called fileserver;
3. the key management server (which does not run under the tao) called keynegoserver (which is based on tcca in the Go version).

Operational Overview

To focus attention, we will explain the operation of the Go Tao at the Linux layer. Continuing the description in terms of the fileproxy applications, the Tao host services are provided by the operating system, which is the "host system" and fileclient and fileserver are the "hosted systems" which run as ordinary Linux process.

The operating system provides the Tao host services using a privileged process, running the (Go) program linux_host; thus, linux_host corresponds to "TcService.exe" in the original version

of cloudproxy. The host system, and its Tao services, are started at Linux boot. The kernel and initramfs, which contains all security critical components, including the Tao, as well as all supporting dynamically linked libraries used by the Tao programs and the Tao hosted programs (fileclient and fileserver), are measured as part of a TPM mediated authenticated boot.

Tao services for the OS are provided by a privileged program, linux_host. Conceptually, the boot script in initram runs linux_host at startup (during development, linux_host is actually started at the command line after boot to simplify testing). In the original version, all execution parameters for linux_host were embedded in the program image. The Go version fetches parameters from environment variables at startup from initram, including, importantly, the policy key. As a result, the notion of code identity for linux_host (and other Tao programs) is more complex than in the original version. However, there is no conceptual difference: the code identity of linux_host, attested to by the TPM at first startup, is a cryptographic measurement of everything that affects the execution of linux_host.

The principal services provided by the Tao are:

- Measured, isolated startup: A Tao Host will, upon request (StartHostedProgram), start an isolated program (e.g., fileclient) after computing a cryptographic measurement of the hosted program and its environment. This measurement is unspoofable in the sense that if conveyed by a trusted source, it provides irrefutable proof of the code, policy and configuration of the hosted program and thus completely defines the deterministic behavior of its execution.
- Attestation:
- Seal: Sealed blobs are saved in the files:
- Unseal:
- GetRandom
- PolicyKey:
- (Later) RotateHostKeys, replay attacks

Principal Names

Principal names are hierarchical. The root name for a hosted program, in the development case, might look something like

```
key([0801100118012245080112410 46cdc82f70552eb...]).Program([25fac93bd4cc
868352c78f4d34df6d2747a17f85...])
```

Here, key([`0801100118012245080112…])` represents the signing key of the host and `Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...])` extends the host name with the hash of the hosted program (`25fac93bd4cc868352c78f4d34df6d2747a17f85...`). If the host were a real linux_host rooted in a TPM boot, its name would name the AIK and the PCRs of the booted linux systems which incorporate the hash of the Authenticated Code Module (“ACM”)

that the bios called to start the authenticated boot and the hash of the linux image and it's initramfs.

Guards

Guards make authorization decisions. Current guards include:

- the liberal guard: this guard returns true for every authorization query
- the conservative guard: this guard returns false for every authorization query
- the ACL guard: this guard provides a list of statements that must return true when the guard is queried for these statements.
- the datalog guard: this guard translates statements in the CloudProxy auth language (see tao/auth/doc.go for details) to datalog statements and uses the Go datalog engine from github.com/kevinawalsh/datalog to answer authorization queries. See tao/datalog_guard.go for the translation from the CloudProxy auth language to datalog. And see install.sh for an example policy.

Domains

Domains are really security contexts. Domains are implemented in tao/domain.go. Domains encapsulate configuration information like name, path to key blobs, path to policy key, and the guard employed for authorization decisions.

CreateDomain initializes a new Domain, writing its configuration files to a directory. This creates the directory and, if needed, a policy key pair encrypted with the given password when stored on disk; it also initializes a default guard of the appropriate type if needed via the call:

```
func CreateDomain(cfg DomainConfig, configPath string, password []byte)
(*Domain, error)
```

Any parameters left empty in cfg will be set to reasonable default values.

Domain information is loaded from a text file, typically called tao.config via the call:

```
LoadDomain(configPath string, password []byte)(*Domain, error)
```

which returns a domain object if successful. The password is used to load a key set from disk. If no password is provided, then LoadDomain will attempt to load verification keys only. For example, LoadDomain is called with a configPath and an nil password to load the policy verification key.

A configuration object, type DomainConfig, holds configuration information for the domain between tao activations.

Control flow in the fileproxy application

When reading this description, please refer to figure 1.

After measured boot of the OS, the OS starts linux_host, a privileged program that implements the Tao Services. linux_host reads several several files to configure itself. The most important of these is tao.env whose location is specified by the environment variable TAO_config_path.

Among these is:

- The linux_host (public) policy key which is in the file policy_key which is in the directory specified in tao.conf .
    - Environment variable: PolicyKeysPath = policy_keys.
- The guard:
    - GuardType = AllowAll
- The linux_host sealed blobs whose location is in the file specified by xxx.

Note that the measurement of the OS consists of the hash of the OS image plus its intiram which includes this configuration information and thus completely characterizes the Tao relevant code and policy identity of the linux_host.

For fileproxy, the boot script (or command line) interface calls linux_host over a designated port (XXX) and requests it start its hosted programs: fileclient and fileproxy. While these typically run on different machines, in our prototype, the test scripts start them on the same machine under the same linux_host.

When fileclient (fileserver) starts, it looks for its sealed blobs, if it finds them, it requests that linux_host unseal to its keys and then retrieves and certificates it needs. If there are no sealed blobs, fileclient (fileserver) realizes it needs to get and certify new “Program Keys.” It makes up a new private/public key pair (using entropy provided by GetRandom) and requests that linux_host Attest to a statement it constructs which bears its newly created public key and its measurement (as computed by linux_host when it was started). It sends this, over a standard tcp channel, to keynegoserver which compares the program measurement in the attestation to its list of “approved programs” and if it is in that list[1], keynegoserver uses the private portion of the policy key to sign a Program Certificate naming the program measurement and the public key. In either case, fileclient (fileserver) has its private key and Program Certificate at the end of this phase.

[1] This is true if keynegoserver is using the ACL guard. If it uses a datalog guard, it's actually checking a more complex policy than is stated here which includes the endorsement to the attestation: this endorsement provides another auth statement that the receiver can verify against the policy key and add to its datalog engine.

fileserver starts and waits for requests on a well known port. A fileclient, contacts a fileserver on this well known port. When it does, the fileclient/fileserver pair, use their Program Certificates to establish an encrypted, integrity protected channel. This is called Channel Establishment. At the end of this, each program has a cryptographically protected channel to its authenticated partner; in addition, fileclient may demonstrate to fileserver that it possesses private keys corresponding to proffered user identities and hence the fileclient channel speaks for fileclient and any such authenticated users.

fileclient requests services (save/retrieve/create/delete files) that are maintained by the server. Briefly, for each such request, fileclient offers cryptographic evidence that it has a particular right to exercise these services for identified files and fileserver validates the evidence using an authorization guard and fulfills any such authorized request. This is described in greater detail in the CloudProxy Tech report.

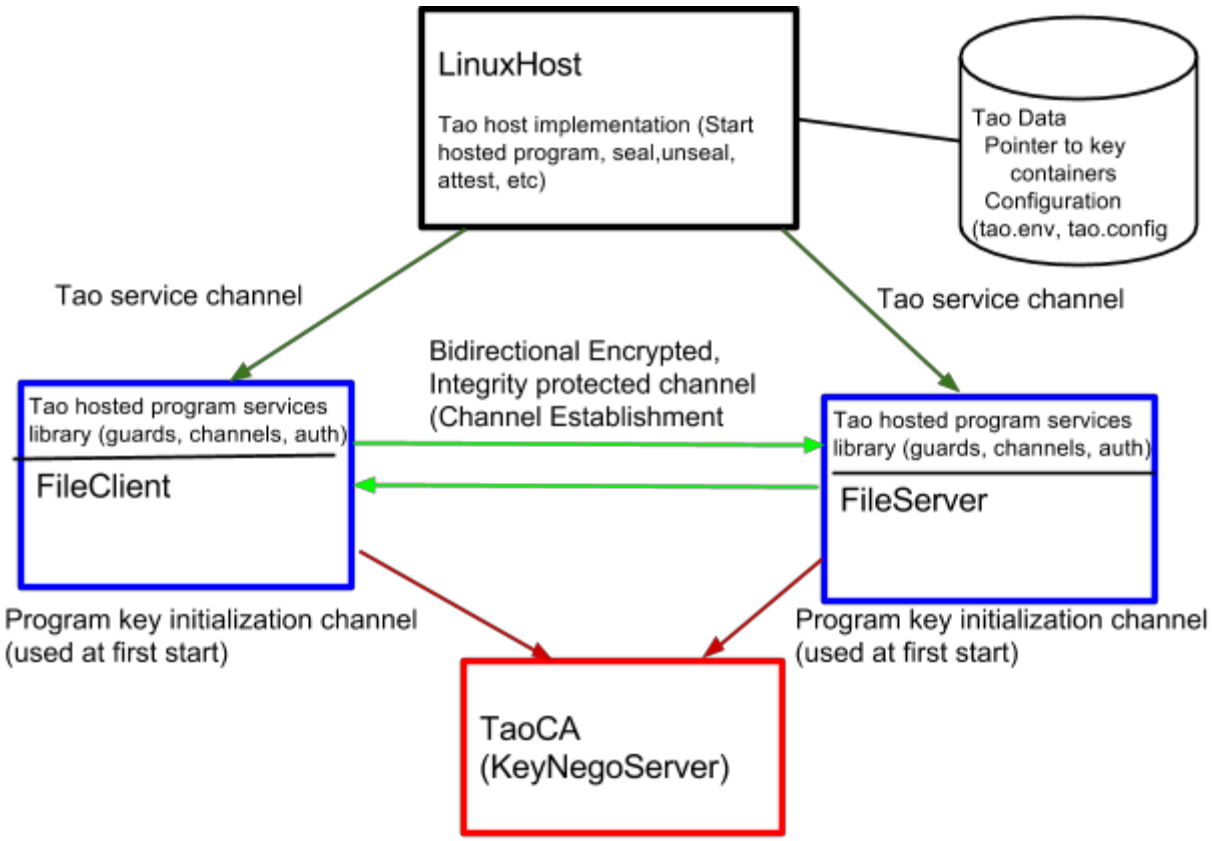

Figure 1- FileProxy Application Architecture

Writing Go Tao programs

The fileclient and fileserver go programs use the Tao library interfaces (github.com/jlmucb/cloudproxy/tao, github.com/jlmucb/cloudproxy/tao/auth, andgithub.com/jlmucb/cloudproxy/tao/net). Specifically, these include:

The Host interface:

- `GetRandomBytes(childSubprin auth.SubPrin, n int) (bytes []byte, err error)`: returns a slice of n random bytes.
- `GetSharedSecret(tag string, n int) (bytes []byte, err error)`: returns a slice of n secret bytes. (This is not currently used in any test programs).
- `Attest(childSubprin auth.SubPrin, issuer *auth.Prin, time, expiration *int64, message auth.Form) (*Attestation, error)`: requests the Tao host sign a statement on behalf of the caller
- `Encrypt(data []byte) (encrypted []byte, err error)`: seals data. (Question: can we specify which principal to seal for? Can there be more than one ---two would be nice for upgrade. Tom's comment for later inclusion: That can be accomplished with Seal policies. The encrypt method is an underlying method that is used to implement Seal, so it's independent of policy). Note that in the current implementation, the only policy is SealPolicyDefault, but this will be generalized in the future to support the kind of operations mentioned in this question.
- `Decrypt(encrypted []byte) (data []byte, err error)`: unseal.
- `AddedHostedProgram(childSubprin auth.SubPrin) error`: create new program.
- `RemovedHostedProgram(childSubprin auth.SubPrin) error`: kill hosted program.
- `TaoHostName() auth.Prin`: Get the Tao principal name assigned to this hosted Tao host. (Unix pathname with hashes, right? ---)for a hosted program under the linux tao, the TaoHostName might be something like tpm([...]).PCRs(...))

A hosted system represented by a tao, `tao`, obtains the pointer to its host interface by calling `tao.Parent()`.

The Guard interface:

- `Subprincipal() auth.SubPrin`: returns a unique subprincipal for this policy.
- `Save(key *Signer) error`: writes all persistent policy data to disk, signed by key
- `Authorize(name auth.Prin, op string, args []string) error`
- `Retract(name auth.Prin, op string, args []string) error`
- `IsAuthorized(name auth.Prin, op string, args []string) bool`
- `AddRule(rule string) error`
- `RetractRule(rule string) error`
- `Clear() error`: removes all rules.
- `Query(query string) (bool, error)`
- `RuleCount() int`
- `GetRule(i int) string`.
- `String() string`: returns a string suitable for showing auth info.

The network interface:

- `func DialWithKeys(network, addr string, guard tao.Guard, v *tao.Verifier, keys *tao.Keys) (net.Conn, error)`
- `func Listen(network, laddr string, config *tls.Config, g tao.Guard, v *tao.Verifier, del *tao.Attestation) (net.Listener, error)`

Running hosted programs in a development environment

In a "production" system, linux_host has already been started and you hosted program has been started. This is not the case in development so here is an example of how to start one. Here's a simple manual procedure to run tao programs (like fileclient) in development on Linux or a Mac. We assume the directory for the cloudproxy binaries are in your path.

In this example, the test specific files are in an initially empty test directory; on my mac, the test directory is `/Users/manferdelli/cloudproxy/apps/testcode/test` and the root of the cloudproxy depository directory is `/Users/manferdelli/cloudproxy`. For convenience, define:

```
export TAO_TEST=/Users/manferdelli/cloudproxy/apps/testcode/test
export TAO_ROOTDIR=/Users/manferdelli/cloudproxy
export TAO_USE_TPM=no
export TAO_config_path=$TAO_TEST/tao.config
export TAO_guard=AllowAll
```

Make a directory for the linux host information and one for the policy keys.

```
cd $TAO_TEST
mkdir linux_tao_host
mkdir policy_keys
```

FIrst, we create a tao.config file, policy key and self signed policy cert. The private key is password protected and we use the password "nopassword" in this example. To create these, use the tao_admin utility:

```
tao_admin -create -name testing -pass nopassword
```

You should see the following output:

```
Initializing new configuration in:
/Users/manferdelli/cloudproxy/apps/testcode/test/tao.config
```

The policy subdirectory you created should have two new files called cert and signer and the test directory should now have a filecalled “tao.config”. Next, create a tao host:

```
linux_host -create -root -pass nopassword
```

It will create a file called “keys” in your linux_tao_host directory and respond with:

```
Loading configuration from:
/Users/manferdelli/cloudproxy/apps/testcode/test/tao.config
Linux Tao Service started and waiting for requests
```

Start the linux_host service as follows:

```
linux_host -service -root -pass nopassword &
```

We added an “&” so as not to block the shell we’re typing to. Note that this also constructs and admin_socket in linux_host_tao which should be removed before you start the service again.

Finally, run your hosted program, using a linux_host binary. In the following command, we assume you’ve copied the binary for the hosted program into $TAO_TEST and it’s called myhostedprogram.exe:

```
linux_host -run -- ./myhostedprogram.exe
```

Your hosted program should now be running and can use the host services.

Running fileproxy

Running the fileproxy demo simply requires running the three component programs. So first, you must compile the three programs. Compile the three programs by typing:

```
cd $TAO_ROOTDIR/apps/fileproxy
go install ..
```

Assuming you have a running linux_host (for example, by following the procedure above), first initialize the keynegoserver policy key and configure the test directory for fileclient and fileserver. Explain file placement, log placement keys,...

In a new shell run keynegoserver:

```
cd $TAO_TEST
linux_host -run -- .keynegoserver
```

In a different shell, initialize fileclient and fileserver.

```
linux_host -run -- fileserver -init
linux_host -run -- fileclient -init
```

After initialization, you can kill keynegoserver (press control-C in the shell you started it in).

Run fileserver in a new shell:

```
cd $TAO_TEST
linux_host -run -- fileserver
```

Then start the fileclient demo; this will construct a file, upload it to fileserver using the tao channel and then retrieve a file while bragging about each and every step in the log file.

```
cd $TAO_TEST
export $GOOGLE_HOST_TAO=
linux_host -run -- fileclient -demo1
```

When the demo concludes, you can kill fileserver if you no longer have any test programs that require it.

Code walk-through

keys.go

A Keys manages a set of signing, verifying, encrypting, and key-deriving keys.

```
type Keys struct:
        dir      string
        policy   string
        keyTypes KeyType
        SigningKey   *Signer
        CryptingKey  *Crypter
        VerifyingKey *Verifier
        DerivingKey  *Deriver
        Delegation   *Attestation
        Cert         *x509.Certificate
}
```

Formats for keys are defined in protobuf. The important ones are:

```
message CryptoKey {
  enum CryptoPurpose {
    VERIFYING = 1;  // public
    SIGNING = 2;    // private
    CRYPTING = 3;   // private
    DERIVING = 4;   // private
  }
  enum CryptoAlgorithm {  // algorithm, mode, etc., all rolled into one
    ECDSA_SHA = 1;
    AES_CTR_HMAC_SHA = 2;
```

```
    HMAC_SHA = 3;
  }
  required CryptoVersion version = 1;
  required CryptoPurpose purpose = 2;
  required CryptoAlgorithm algorithm = 3;
  required bytes key = 4;  // serialized
<algorithm><purpose>Key<version>
}

message CryptoKeyset {
  repeated CryptoKey keys = 1;
}

// PBEData is used by root Tao hosts to seal a serialized CryptoKeyset
// using a user-chosen password.
message PBEData {
  required CryptoVersion version = 1;
  required string cipher = 2;  // "aes128-cbc"
  required string hmac = 3;  // "sha256"
  required int32 iterations = 4;  // 4096
  required bytes iv = 5;
  required bytes ciphertext = 6;
  required bytes salt = 7;
}

enum NamedEllipticCurve {
  PRIME256_V1 = 1;  // aka secp256r1
}
message ECDSA_SHA_VerifyingKey_v1 {
  required NamedEllipticCurve curve = 1;
  required bytes ec_public = 2;  // =
OpenSSL::EC_POINT_point2oct(pub_key)
}

message ECDSA_SHA_SigningKey_v1 {
  required NamedEllipticCurve curve = 1;
  required bytes ec_private = 2;  // = OpenSSL::BN_bn2bin(priv_key)
  required bytes ec_public = 3;  // =
OpenSSL::EC_POINT_point2oct(pub_key)
}

enum CryptoCipherMode {
  CIPHER_MODE_CTR = 1;
}

message AES_CTR_HMAC_SHA_CryptingKey_v1 {
  required CryptoCipherMode mode = 1;
  required bytes aes_private = 2;
```

```
  required bytes hmac_private = 3;
}

enum CryptoDerivingMode {
 DERIVING_MODE_HKDF = 1;
}

message HMAC_SHA_DerivingKey_v1 {
  required CryptoDerivingMode mode = 1;
  required bytes hmac_private = 2;
}

// signing and encryption use a short header that contains a
// version number and a four-byte key-hint to distinguish among keys
message CryptoHeader {
  required CryptoVersion version = 1;
  required bytes key_hint = 2;
}
// The result of signing.
message SignedData {
  required CryptoHeader header = 1;
  required bytes signature = 2;
}

// The result of encrypting.
message EncryptedData {
  required CryptoHeader header = 1;
  required bytes iv = 2;
  required bytes ciphertext = 3;
  optional bytes mac = 4;  // optional for modes that don't require mac
}

// A PDU to be serialized and fed to HKDF for derivation.
message KeyDerivationPDU {
  required bytes previous_hash = 1;
  required fixed32 size = 2;
  required string context = 3;
  required fixed32 index = 4;
}
```

Functions include:

- `func GenerateSigner() (*Signer, error)`：creates a new Signer with a fresh key.
- `func (s *Signer) ToPrincipal() auth.Pri`: produces a "key" type Prin for this signer. This contains a serialized CryptoKey for the public half of this signing key.
- `func MarshalSignerDER(s *Signer) ([]byte, error)`： serializes the signer to DER.

- `func UnmarshalSignerDER(signer []byte) (*Signer, error)`: deserializes a signer from DER.
- `func NewX509Name(p X509Details) *pkix.Name`: returns a new pkix.Name
- `func MarshalSignerProto(s *Signer) (*CryptoKey, error)`: encodes a signing key as a CryptoKey protobuf message.
- `func (s *Signer) CreateSignedX509(caCert *x509.Certificate, certSerial int, subjectKey *Verifier, subjectName *pkix.Name) (*x509.Certificate, error)`: creates a signed X.509 certificate for some other subject's key.
- `func marshalECDSA_SHA_SigningKeyV1(k *ecdsa.PrivateKey) *ECDSA_SHA_SigningKeyV1`: encodes a private key as a protobuf message.
- `func MarshalSignerProto(s *Signer) (*CryptoKey, error)`: encodes a signing key as a CryptoKey protobuf message
- `func (s *Signer) CreateSelfSignedX509(name *pkix.Name) (*x509.Certificate, error)`: creates a self-signed X.509 certificate for the public key of this Signer.
- `func marshalECDSA_SHA_VerifyingKeyV1(k *ecdsa.PublicKey) *ECDSA_SHA_VerifyingKeyV1`: encodes a public key as a protobuf message.
- `func (v *Verifier) Verify(data []byte, context string, sig []byte) (bool, error)`: checks an ECDSA signature over the contextualized data, using the public key of the verifier.
- `func (v *Verifier) ToPrincipal() auth.Prin`: produces a "key" type Prin for this verifier. This contains a serialized CrypoKey for this key.
- `func FromPrincipal(prin auth.Prin) (*Verifier, error)`: deserializes a Verifier from a Prin.
- `func FromX509(cert *x509.Certificate) (*Verifier, error)`: creates a Verifier from an X509 certificate.
- `func UnmarshalVerifierProto(ck *CryptoKey) (*Verifier, error)`: decodes a verifying key from a CryptoKey protobuf message.
- `func (v *Verifier) CreateHeader() (*CryptoHeader, error)`: instantiates and fills in a header for this verifying key.
- `func contextualizeData(h *CryptoHeader, data []byte, context string) ([]byte, error)`: produces a single string from a header, data, and a context.
- `func contextualizedSHA256(h *CryptoHeader, data []byte, context string, digestLen int) ([]byte, error)`: performs a SHA-256 sum over contextualized data.
- `func (c *Crypter) Encrypt(data []byte) ([]byte, error)`: encrypts plaintext into ciphertext and protects ciphertext integrity with a MAC.
- `func (c *Crypter) Decrypt(ciphertext []byte) ([]byte, error)`: checks the MAC then decrypts ciphertext into plaintext.
- `func marshalAES_CTR_HMAC_SHA_CryptingKeyV1(c *Crypter) *AES_CTR_HMAC_SHA_CryptingKeyV1`: encodes a private AES/HMAC key pair into a protobuf message.
- `func MarshalCrypterProto(c *Crypter) (*CryptoKey, error)`: encodes a Crypter as a CryptoKey protobuf message.
- `func UnmarshalCrypterProto(ck *CryptoKey) (*Crypter, error)`: decodes a crypting key from a CryptoKey protobuf message.
- `func (c *Crypter) CreateHeader() (*CryptoHeader, error)`: instantiates and fills in a header for this crypting key.

- `func GenerateDeriver() (*Deriver, error)`: generates a deriver with a fresh secret.
- `func (d *Deriver) Derive(salt, context, material []byte) error`: uses HKDF with HMAC-SHA256 to derive key bytes in its material parameter.
- `func marshalHMAC_SHA_DerivingKeyV1(d *Deriver) *HMAC_SHA_DerivingKeyV1`: encodes a deriving key as a protobuf message.
- `func MarshalDeriverProto(d *Deriver) (*CryptoKey, error)`: encodes a Deriver as a CryptoKey protobuf message.
- `func UnmarshalDeriverProto(ck *CryptoKey) (*Deriver, error)`: decodes a deriving key from a CryptoKey protobuf message.
- `func (k *Keys) X509Path() string`: returns the path to the verifier key, stored as an X.509 certificate.
- `func (k *Keys) PBEKeysetPath() string`: returns the path for stored keys.
- `func (k *Keys) PBESignerPath() string`: returns the path for a stored signing key.
- `func (k *Keys) SealedKeysetPath() string`: returns the path for a stored signing key.
- `func (k *Keys) DelegationPath() string`: returns the path for a stored signing key.
- `func NewTemporaryKeys(keyTypes KeyType) (*Keys, error)`: creates a new Keys structure with the specified keys. (what's temporary about the keys)
- `func NewOnDiskPBEKeys(keyTypes KeyType, password []byte, path string, name *pkix.Name) (*Keys, error)`: creates a new Keys structure with the specified key types store under PBE on disk. If keys are generated and name is not nil, then a self-signed x509 certificate will be generated and saved as well.
- `func (k *Keys) newCert(name *pkix.Name) (err error)`:
- `func (k *Keys) loadCert() error`:
- `func NewTemporaryTaoDelegatedKeys(keyTypes KeyType, t Tao) (*Keys, error)`: initializes a set of temporary keys under a host Tao, using the Tao to generate a delegation for the signing key. Since these keys are never stored on disk, they are not sealed to the Tao.
- `func PBEEncrypt(plaintext, password []byte) ([]byte, error)`: encrypts plaintext using a password to generate a key. Note that since this is for private program data, we don't try for compatibility with the C++ Tao version of the code.
- `func PBEDecrypt(ciphertext, password []byte) ([]byte, error`: decrypts ciphertext using a password to generate a key. Note that since this is for private program data, we don't try for compatibility with the C++ Tao version of the code.
- `func MarshalKeyset(k *Keys) (*CryptoKeyset, error)`: encodes the keys into a protobuf message.
- `func UnmarshalKeyset(cks *CryptoKeyset) (*Keys, error)`: decodes a CryptoKeyset into a temporary Keys structure. Note that this Keys structure doesn't have any of its variables set.
- `func NewOnDiskTaoSealedKeys(keyTypes KeyType, t Tao, path, policy string) (*Keys, error)`: sets up the keys sealed under a host Tao or reads sealed keys.

auth/ast.go

AuthLogicElement is any element of the authorization logic, i.e. a formula, a term, or a principal extension.

```
type AuthLogicElement interface {
```

```
        // Marshal writes a binary encoding of the element into b.
        Marshal(b *Buffer)
        // String returns verbose pretty-printing text for the element.
        String() string
        // ShortString returns short debug-printing text for the
element.
        ShortString() string
        // fmt.Formatter is satisfied by all elements.
        isAuthLogicElement() // marker
}
```

Prin uniquely identifies a principal by a public key, used to verify signatures on credentials issued by the principal, and a sequence of zero or more extensions to identify the subprincipal of that key.

```
type Prin struct {
        Type string  // either "key" or "tpm".
        Key  Term    CryptoKey protobuf structure
        Ext  SubPrin // zero or more extensions for descendents
}

PrinExt is an extension of a principal.
type PrinExt struct {
        Name string // [A-Z][a-zA-Z0-9_]*
        Arg  []Term
}
```

SubPrin is a series of extensions of a principal.

```
type SubPrin []PrinExt
```

- `func (p Prin) MakeSubprincipal(e SubPrin) Prin`: MakeSubprincipal creates principal given principal p and extensions e.
- `func NewKeyPrin(material []byte) Prin`: NewKeyPrin returns a new Prin of type "key" with the given key material.

root_host.go

RootHost is a standalone implementation of Host, it uses a global of type:

```
type RootHost struct {
        keys        *Keys
        taoHostName auth.Prin
}
```

- `func NewTaoRootHostFromKeys(k *Keys) (Host, error)` f: returns a RootHost that uses these keys.

- `func NewTaoRootHost() (Host, error)`: generates a new RootHost with a fresh set of temporary keys.
- `func (t *RootHost) GetSharedSecret(tag string, n int) (bytes []byte, err error)`: returns a slice of n secret bytes.

Attest requests the Tao host sign a statement on behalf of the caller. The skeletal implementation is:

```
func (t *RootHost) Attest(childSubprin auth.SubPrin, issuer *auth.Prin,
        time, expiration *int64, message auth.Form) (*Attestation,
error) {
        child := t.taoHostName.MakeSubprincipal(childSubprin)
        if issuer != nil {
                if !auth.SubprinOrIdentical(*issuer, child) {
                        return nil, newError("invalid issuer in
statement")
                }
        } else {
                issuer = &child
        }
        stmt := auth.Says{Speaker: *issuer, Time: time, Expiration:
expiration, Message: message}
        return GenerateAttestation(t.keys.SigningKey, nil /* delegation
*/, stmt)
}
```

- `func (t *RootHost) AddedHostedProgram(childSubprin auth.SubPrin) error`: notifies this Host that a new hosted program has been created.
- `func (t *RootHost) TaoHostName() auth.Prin`: TaoHostName gets the Tao principal name assigned to this hosted Tao host. The name encodes the full path from the root Tao, through all intermediary Tao hosts, to this hosted Tao host.

stacked_host.go

A StackedHost implements Host over an existing host Tao.

```
type StackedHost struct {
        taoHostName auth.Prin
        hostTao     Tao
        keys        *Keys
}
```

- `func NewTaoStackedHostFromKeys(k *Keys, t Tao) (Host, error)` : takes ownership of an existing set of keys and returns a StackedHost that uses these keys over an existing host Tao.
- `func NewTaoStackedHost(t Tao) (Host, error)` : generates a new StackedHost with a fresh set of temporary keys.

- `func (t *StackedHost) GetSharedSecret(tag string, n int) (bytes []byte, err error)`: returns a slice of n secret bytes.

Attest requests the Tao host sign a statement on behalf of the caller.

```
func (t *StackedHost) Attest(childSubprin auth.SubPrin, issuer
*auth.Prin,
        child := t.taoHostName.MakeSubprincipal(childSubprin)
        return GenerateAttestation(t.keys.SigningKey, d, stmt)
}
```

- `func (t *StackedHost) Encrypt(data []byte) (encrypted []byte, err error)`: Encrypt data so that only this host can access it.
- `func (t *StackedHost) Decrypt(encrypted []byte) (data []byte, err error)` : Decrypt data that only this host can access.
- `func (t *StackedHost) AddedHostedProgram(childSubprin auth.SubPrin) error` : AddedHostedProgram notifies this Host that a new hosted program has been created.

TaoHostName gets the Tao principal name assigned to this hosted Tao host. The name encodes the full path from the root Tao, through all intermediary Tao hosts, to this hosted Tao host.

```
func (t *StackedHost) TaoHostName() auth.Prin {
        return t.taoHostName
}
```

linux_process_factory.go

A LinuxProcessFactory supports methods for creating Linux processes as hosted programs. It is implemented in linux_process_factory.go. The key functions are:

- `func FormatHostedProgramSubprin(id uint, hash []byte) auth.SubPrin`: produces a string that represents a subprincipal with the given ID and hash.
- `func (LinuxProcessFactory) MakeHostedProgramSubprin(id uint, prog string) (subprin auth.SubPrin, temppath string, err error)`: computes the hash of a program to get its hosted-program subprincipal. In the process, it copies the program to a temporary file controlled by this code and returns the path to that new binary.

ForkHostedProgram uses a path and arguments to fork a new process. The skeleton implementation is:

```
func (LinuxProcessFactory) ForkHostedProgram(prog string, args []string)
(io.ReadWriteCloser, *exec.Cmd, error) {
        // Get a pipe pair for communication with the child.
       defer clientWrite.Close()
       serverRead, clientWrite, err := os.Pipe();
```

```
        defer clientWrite.Close();
        clientRead, serverWrite, err := os.Pipe()
         env := os.Environ()
         evar := HostTaoEnvVar+"=tao::TaoRPC+tao::FDMessageChannel(3, 4)"
         if err := cmd.Start(); err != nil {
                 channel.Close()
                 return nil, nil, err
         }
        return channel, cmd, nil
}
```

linux_host

A LinuxHost is a Tao host environment in which hosted programs are Linux processes. A Unix domain socket accepts administrative commands for controlling the host, e.g., for starting hosted processes, stopping hosted processes, or shutting down the host. A LinuxTao can be run in stacked mode (on top of a host Tao) or in root mode (without an underlying host Tao).

```
type LinuxHost struct {
        path           string
        guard          Guard
        taoHost        Host
        childFactory   LinuxProcessFactory
        hostedPrograms []*LinuxHostChild
        hpm            sync.RWMutex
        nextChildID    uint
        idm            sync.Mutex
}
```

NewStackedLinuxHost creates a new LinuxHost as a hosted program of an existing host Tao. The core implementation is:

```
func NewStackedLinuxHost(path string, guard Guard, hostTao Tao)
(*LinuxHost, error) {
        lh := &LinuxHost{
                path:  path,
                guard: guard,
        }
        if _, ok := hostTao.(*TPMTao); !ok {
                subprin := guard.Subprincipal()
                if err := hostTao.ExtendTaoName(subprin); err != nil {
                        return nil, err
                }
        }
        k, err := NewOnDiskTaoSealedKeys(Signing|Crypting|Deriving,
hostTao, path, SealPolicyDefault)
```

```
        if err != nil {
                return nil, err
        }
        lh.taoHost, err = NewTaoStackedHostFromKeys(k, hostTao)
        return lh, nil
}
```

- `func NewRootLinuxHost(path string, guard Guard, password []byte) (*LinuxHost, error)`: creates a new LinuxHost as a standalone Host that can provide the Tao to hosted Linux processes.

LinuxHostChild holds state associated with a running child program.

```
type LinuxHostChild struct {
        channel      io.ReadWriteCloser
        ChildSubprin auth.SubPrin
        Cmd          *exec.Cmd
}
```

- `func (lh *LinuxHost) GetTaoName(child *LinuxHostChild) auth.Prin`: returns the Tao name for the child.
- `func (lh *LinuxHost) ExtendTaoName(child *LinuxHostChild, ext auth.SubPrin) error`: irreversibly extends the Tao principal name of the child.
- `func (lh *LinuxHost) GetTaoName(child *LinuxHostChild) auth.Prin`: returns the Tao name for the child.
- `func (lh *LinuxHost) ExtendTaoName(child *LinuxHostChild, ext auth.SubPrin) error`: irreversibly extends the Tao principal name of the child.

Seal encrypts data for the child. This call also zeroes the data parameter.

```
func (lh *LinuxHost) Seal(child *LinuxHostChild, data []byte, policy
string) ([]byte, error) {
        defer zeroBytes(data)
        lhsb := &LinuxHostSealedBundle{
                Policy: proto.String(policy),
                Data:   data,
        }
        switch policy {
        case SharedSecretPolicyDefault:
        case SharedSecretPolicyConservative:
        // We are using a master key-deriving key shared among all
        // similar LinuxHost instances. For LinuxHost, the default
        // and conservative policies means any process running the same
        // program binary as the caller hosted on a similar
        // LinuxHost.
                lhsb.PolicyInfo =
proto.String(child.ChildSubprin.String())
```

```
        case SharedSecretPolicyLiberal:
        // The most liberal we can do is allow any hosted process
        // running on a similar LinuxHost instance. So, we don't set
        // any policy info.
        default:
                return nil, newError("policy not supported for Seal: " +
policy)
        }
        m, err := proto.Marshal(lhsb)
        if err != nil {
                return nil, err
        }
        defer zeroBytes(m)
        return lh.taoHost.Encrypt(m)
}
```

- `func (lh *LinuxHost) Unseal(child *LinuxHostChild, sealed []byte) ([]byte, string, error)`: decrypts data for the child, but only if the policy is satisfied.
- `func (lh *LinuxHost) Attest(child *LinuxHostChild, issuer *auth.Prin, time, expiration *int64, stmt auth.Form) (*Attestation, error)`: signs a statement on behalf of the child.
- `func (lh *LinuxHost) StartHostedProgram(path string, args []string) (auth.SubPrin, int, error)`: starts a new hosted program.
- `func (lh *LinuxHost) StopHostedProgram(subprin auth.SubPrin) error`: stops a running hosted program.
- `func (lh *LinuxHost) ListHostedPrograms() ([]auth.SubPrin, []int, error)`:returns a list of running hosted programs.
- `func (lh *LinuxHost) KillHostedProgram(subprin auth.SubPrin) error`:kills a running hosted program.
- `func (lh *LinuxHost) TaoHostName() auth.Prin`: returns the name of the Host used by the LinuxHost.

linux_host_tao_rpc

LinuxHostTaoServer is a server stub for LinuxHost's Tao RPC interface.

```
type LinuxHostTaoServer struct {
        lh    *LinuxHost
        child *LinuxHostChild
}
type linuxHostTaoServerStub LinuxHostTaoServer
```

NewLinuxHostTaoServer returns a new server stub for LinuxHost's Tao RPC interface.

```
func NewLinuxHostTaoServer(host *LinuxHost, child *LinuxHostChild)
LinuxHostTaoServer {
        return LinuxHostTaoServer{host, child}
```

```
}
```

Serve listens on sock for new connections and services them.

```
func (server LinuxHostTaoServer) Serve(conn io.ReadWriteCloser) error {
        s := rpc.NewServer()
        err := s.RegisterName("Tao", linuxHostTaoServerStub(server))
        s.ServeCodec(protorpc.NewServerCodec(conn))
        return nil
}
```

GetTaoName is the server stub for Tao.GetTaoName.

```
func (server linuxHostTaoServerStub) GetTaoName(r *TaoRPCRequest, s
*TaoRPCResponse) error {
        s.Data = auth.Marshal(server.lh.GetTaoName(server.child))
        return nil
}
```

ExtendTaoName is the server stub for Tao.ExtendTaoName.

```
func (server linuxHostTaoServerStub) ExtendTaoName(r *TaoRPCRequest, s
*TaoRPCResponse) error {
        ext, err := auth.UnmarshalSubPrin(r.Data)
        return server.lh.ExtendTaoName(server.child, ext)
}
```

- `func (server linuxHostTaoServerStub) GetRandomBytes(r *TaoRPCRequest, s *TaoRPCResponse) error`:is the server stub for Tao.GetRandomBytes.
- `func (server linuxHostTaoServerStub) GetSharedSecret(r *TaoRPCRequest, s *TaoRPCResponse) error`:is the server stub for Tao.GetSharedSecret.

Seal is the server stub for Tao.Seal.

```
func (server linuxHostTaoServerStub) Seal(r *TaoRPCRequest, s
*TaoRPCResponse) error {
        if r.Policy == nil {
                return newError("missing policy")
        }
        data, err := server.lh.Seal(server.child, r.Data, *r.Policy)
        s.Data = data
        return err
}
```

- `func (server linuxHostTaoServerStub) Unseal(r *TaoRPCRequest, s *TaoRPCResponse) error`: is the server stub for Tao.Unseal.

- `func (server linuxHostTaoServerStub) Attest(r *TaoRPCRequest, s *TaoRPCResponse) error`:is the server stub for Tao.Attest.

net/listener.go

A Listener implements net.Listener for Tao connections. Each time it accepts a connection, it exchanges Tao attestation chains and checks the attestation for the certificate of the client against its tao.Guard. The guard in this case should be the guard of the Tao domain. This listener allows connections from any program that is authorized under the Tao to execute.

```
type listener struct {
        gl         net.Listener
        guard      tao.Guard
        verifier   *tao.Verifier
        delegation *tao.Attestation
}
```

NewTaoListener returns a new Tao-based net.Listener that uses the underlying crypto/tls net.Listener and a tao.Guard to check whether or not connections are authorized.

```
func Listen(network, laddr string, config *tls.Config, g tao.Guard, v
*tao.Verifier, del *tao.Attestation) (net.Listener, error) {
        config.ClientAuth = tls.RequireAnyClientCert
        inner, err := tls.Listen(network, laddr, config)
        return &listener{inner, g, v, del}, nil
}
```

- `func ValidatePeerAttestation(a *tao.Attestation, cert *x509.Certificate, guard tao.Guard) error`: checks a tao.Attestation for a given Listener against an X.509 certificate from a TLS channel.

Accept waits for a connect, accepts it using the underlying Conn and checks the attestations and the statement.

```
func (l *listener) Accept() (net.Conn, error) {
        c, err := l.gl.Accept()
        // Protocol:
        // 0. TLS handshake (executed automatically on first message)
        // 1. Client -> Server: Tao delegation for X.509 certificate.
        // 2. Server: checks for a Tao-authorized program.
        // 3. Server -> Client: Tao delegation for X.509 certificate.
        // 4. Client: checks for a Tao-authorized program.
        ms := util.NewMessageStream(c)
        var a tao.Attestation
        if err := ms.ReadMessage(&a); err != nil {
                c.Close()
```

```
                return nil, err
        }
        if err := AddEndorsements(l.guard, &a, l.verifier); err != nil {
                return nil, err
        }
        peerCert := c.(*tls.Conn).ConnectionState().PeerCertificates[0]
        if err := ValidatePeerAttestation(&a, peerCert, l.guard); err !=
nil {
                c.Close()
                return nil, err
        }
        if _, err := ms.WriteMessage(l.delegation); err != nil {
                c.Close()
                return nil, err
        }
        return c, nil
}
```

- `func (l *listener) Close() error` : closes the listener.
- `func (l *listener) Addr() net.Addr`: returns the listener's network address.

net/client.go

- `func EncodeTLSCert(keys *tao.Keys) (*tls.Certificate, error)`:EncodeTLSCert combines a signing key and a certificate in a single tls certificate suitable for a TLS config.
- `func generateX509() (*tao.Keys, *tls.Certificate, error)`: creates a fresh set of Tao-delegated keys and gets a certificate from these keys.
- `func ListenTLS(network, addr string) (net.Listener, error)`: creates a fresh certificate and listens for TLS connections using it.
- `func DialTLS(network, addr string) (net.Conn, error)`: creates a new X.509 certs from fresh keys and dials a given TLS address.
- `func DialTLSWithKeys(network, addr string, keys *tao.Keys) (net.Conn, error)`: connects to a TLS server using an existing set of keys.

Dial connects to a Tao TLS server, performs a TLS handshake, and exchanges tao.Attestation values with the server, checking that this is a Tao server that is authorized to Execute. It uses a Tao Guard to perform this check.

```
func Dial(network, addr string, guard tao.Guard, v *tao.Verifier)
(net.Conn, error) {
        keys, _, err := generateX509()
        return DialWithKeys(network, addr, guard, v, keys)
}
```

DialWithKeys connects to a Tao TLS server using an existing set of keys.

```
func DialWithKeys(network, addr string, guard tao.Guard, v
*tao.Verifier, keys *tao.Keys) (net.Conn, error) {
        if keys.Cert == nil {
                return nil, fmt.Errorf("client: can't dial with an empty
client certificate\n")
        }
        tlsCert, err := EncodeTLSCert(keys)
        conn, err := tls.Dial(network, addr, &tls.Config{
                RootCAs:            x509.NewCertPool(),
                Certificates:       []tls.Certificate{*tlsCert},
                InsecureSkipVerify: true,
        })
        // Tao handshake: send client delegation.
        ms := util.NewMessageStream(conn)
        if _, err = ms.WriteMessage(keys.Delegation); err != nil {
                conn.Close()
                return nil, err
        }
        // Tao handshake: read server delegation.
        var a tao.Attestation
        if err := ms.ReadMessage(&a); err != nil {
                conn.Close()
                return nil, err
        }
        if err := AddEndorsements(guard, &a, v); err != nil {
                conn.Close()
                return nil, err
        }
        // Validate the peer certificate according to the guard.
        peerCert := conn.ConnectionState().PeerCertificates[0]
        if err := ValidatePeerAttestation(&a, peerCert, guard); err !=
nil {
                conn.Close()
                return nil, err
        }
        return conn, nil
}
```

- `func AddEndorsements(guard tao.Guard, a *tao.Attestation, v *tao.Verifier) error`:reads the SerializedEndorsements in an attestation and adds the ones that are predicates signed by the policy key.
- `func TruncateAttestation(kprin auth.Prin, a *tao.Attestation) (auth.Says, auth.PrinExt, error)`: cuts off a delegation chain at its "Program" subprincipal extension and replaces its prefix with the given key principal. It also returns the PrinExt that represents exactly the program hash.

- `func IdenticalDelegations(s, t auth.Form) bool`: IdenticalDelegations checks to see if two Form values are Says and are identical delegations (i.e., the Message must be an auth.Speaksfor). This function is not in the auth package, since it's specific to a particular .

net/ca.go

HandleCARequest checks a request from a program and responds with a truncated delegation signed by the policy key.

```
func HandleCARequest(conn net.Conn, s *tao.Signer, guard tao.Guard) {
        defer conn.Close()
        // Expect an attestation from the client.
        ms := util.NewMessageStream(conn)
        ra, err := tao.GenerateAttestation(s, nil, truncSays)
        endorsement := auth.Says{
                Speaker: s.ToPrincipal(),
                Message: auth.Pred{
                        Name: "TrustedProgramHash",
                        Arg:  []auth.Term{auth.PrinTail{Ext:
[]auth.PrinExt{pe}}},
                },
        }
        if truncSays.Time != nil {
                i := *truncSays.Time
                endorsement.Time = &i
        }
        if truncSays.Expiration != nil {
                i := *truncSays.Expiration
                endorsement.Expiration = &i
        }
        ea, err := tao.GenerateAttestation(s, nil, endorsement)
        eab, err := proto.Marshal(ea)
        var a tao.Attestation
        if err := ms.ReadMessage(&a); err != nil {
                return
        }
        peerCert :=
conn.(*tls.Conn).ConnectionState().PeerCertificates[0]
        if err := ValidatePeerAttestation(&a, peerCert, guard); err !=
nil {
                return
        }
        truncSays, pe, err := TruncateAttestation(s.ToPrincipal(), &a)
        if err != nil {
                fmt.Fprintln(os.Stderr, "Couldn't truncate the
attestation:", err)
```

```
                return
        }
        ra, err := tao.GenerateAttestation(s, nil, truncSays)
        endorsement := auth.Says{
                Speaker: s.ToPrincipal(),
                Message: auth.Pred{
                        Name: "TrustedProgramHash",
                        Arg:  []auth.Term{auth.PrinTail{Ext:
[]auth.PrinExt{pe}}},
                },
        }
        if truncSays.Time != nil {
                i := *truncSays.Time
                endorsement.Time = &i
        }
        if truncSays.Expiration != nil {
                i := *truncSays.Expiration
                endorsement.Expiration = &i
        }
        ea, err := tao.GenerateAttestation(s, nil, endorsement)
        eab, err := proto.Marshal(ea)
        ra.SerializedEndorsements = [][]byte{eab}
        if _, err := ms.WriteMessage(ra); err != nil {
                fmt.Fprintln(os.Stderr, "Couldn't return the attestation
on the channel:", err)
                return
        }
        return
}
```

- `func RequestTruncatedAttestation(network, addr string, keys *tai.Keys, v *tao.Verifier) (*tao.Attestation, error)`: connects to a CA instance, sends the attestation for an X.509 certificate, and gets back a truncated attestation with a new principal name based on the policy key.

Libraries and support code

Standard Go libraries for crypto, SSL and other support functions.

Developer's view of fileproxy

todo

Get Domain from env vars (code path)
Init (make keys, seal, attest, store blobs, cert, etc)

KeyNegoServer

Private Key initialization (tao_admin)
Initializing approved principal (code) list
Approved machine signer root key
open port
process requests
    syntax checking
    check attestor chain
    store and issue

FileProxy.FC startup
    (Init) Same as Linux host
    policy loading

FC processing
    write a file
        construct file
        collect evidence
        upload request
        upload file
    read a file

FS processing

fileproxy in C++