**Cloudproxy Nuts and Bolts**

manferdelli@, tmroeder@

**Overview**

Cloudproxy is a software system provide *remotely authenticated* isolation, confidentiality and integrity of code and data for Hosted Systems preventing attacks from co-tenants and (under modest assumptions) insiders in a remote data center. Cloudproxy operates on two components to achieve this, a “Host System” (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to a “Hosted System” (VM, Application, Container).

Cloudproxy provides a mechanism at each level of the software stack to isolate Hosted Systems, measure and remotely verify the exact software and configuration information constituting the Hosted System and provide security services like sealing that ensures that information (like keys) can be securely provisioned and retrieved only by the correct Hosted System, while isolated, on a supported platform.

The key concept, for Cloudproxy, is Code Identity and Measurement coupled with isolation, secret provisioning. A Host System measures a Hosted System incorporating the actual binary code and configuration information resulting in an unforgeable global identity for that code and execution context. Since the Hosted System knows the “identity” of each Hosted System (i.e.- the unforgeable global identity), it can store secrets that only the Hosted System will receive[[1]](#footnote-1). The Host Systems can also “attest” to statements made by Hosted Systems by incorporating the unforgeable global identity in statements it signs (again with keys only an isolated Host System has access to). The upshot of this is a Cloudproxy Hosted System can be isolated, maintain secrets only it knows to encrypt and integrity protect all data it receives or sends and it can securely authenticate itself and thus can employ authenticated public keys tied to its identity that can be relied upon by parties it communicates with over a network.

Readers can consult [1] for a fuller description.

**The Tao**

A Hosted System uses the Cloudproxy API, called the Tao, to achieve the security promises (program isolation, and confidentiality and integrity for programs and data) provided by Cloudproxy. The Programming model is simple and require only a few API calls. The Tao Library is linked into an executable to provide the programming interface in Go or C++.

The basic calls are:

**StartHostedProgram**: StartHostedProgram instructs the Host System to measure and start a new, isolated Hosted System. It names the binary image and other context data to start the program. The Hosted System could be, for example, a VM if the Host System is a VMM or an isolated Linux process if the Host System is Linux.

**Seal**: Seal takes an opaque data blob and appends the measurement of the Hosted System. It encrypts and integrity protects the resulting object (using keys only the Host System knows) and returns the resulting opaque object to the Hosted System. Hosted Systems typically “seal” private signing and encryption keys so they can be later recovered when the Hosted System is restarted using “Unseal” below.

**Unseal**: Unseal takes an opaque blob (produced by a prior “Seal”) from a Hosted System. It decrypts (and checks the integrity of) the blob and compares the measurement of the Hosted System requesting the unseal with the measurement of the Hosted System named in the blob. If the measurements match, it returns the protected data.

**Attest**: Attest takes a blob from a Hosted System and signs a statement naming the blob and the measurement of the Hosted System requesting the Attest. It returns the signed statement (the “Attestation”) along with a certificate (the “Host Certificate”) from an authority certifying that the public key it used to sign the statement belongs to a verified Host System with enumerated security characteristics. See [1] for details for the “Trust Model” and mechanism that allows a recipient of such a certificate to rely on the association between the public key named in the Host Certificate and a trustworthy Cloudproxy Hosted System. The meaning of the signed blob is, informally, “Statement X came from the program with Measurement M while it was isolated. Hosted Systems mainly use attest as follows: they generate a public-private key pair and seal the private key; then they request an attest of the corresponding public key. A party receiving the Attestation and Host Certificate can cryptographically verify the public key came from the named program while isolated and thus subsequent proof of possession of the private key can be used to authenticate statements from the Hosted System.

**GetRandom**: GetRandom provides cryptographically random bits, typically for key generation.

**Principal Names**

Principal names in Cloudproxy are hierarchical and securely name the principal. For example, a principal rooted in a public key will have the public key in its name and a program principal (a measured Cloudproxy Hosted System will have the measurement in the principal name.

The root name for a hosted program, in the development case, might look something like

key([080110011801224508011241046cdc82f70552eb...]).Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...])

Here, key([080110011801224508011…])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[[2]](#footnote-2).

**The Tao Paradigm**

The Tao as used by a Hosted System is almost always used in a stereotypical way. Programs always have policy public keys embedded in their image either explicitly or implicitly. Statements signed by the corresponding private (and only those statements) are accepted and acted on by these programs. The policy key(s) plus the Hosted System code reflected in its measure fully describe the security do

When a program starts on a Hosted System, it makes up a public/private key-pair and several keys it uses to “seal” information for itself; it then “seals” all the private information. Next it requests and Attestation naming the newly generated public key and sends the Attestation to a service for the domain which confirms the security properties in the Attestation and Host Certificate and, if these meet security domain requirements, the security domain service signs (with the private portion of the “policy key” whose public portion is in the Hosted System image) an x509 certificate specifying the public key and the Tao Principal Name of the Hosted System

(as mentioned above, this name, specifies, among other things, the Hosted System measurement). This resulting certificate, called the Program Certificate, can be used by any Hosted System to prove its identity to another Hosted System in the same security domain. Program Certificates are used to negotiate mutually SSL channels between Hosted Systems and such Hosted Systems can share information over these channels with full assurance that the it knows the code identity and security properties of its channel partner. In particular, Hosted Systems in the same security domain can full trust other Hosted Systems in the same security domain with data or processing. All communications on these channels is encrypted and integrity protected. Typically, Hosted Systems use the symmetric keys it generates at initialization to encrypt and integrity protect information it stores on disks or elsewhere over the network. Employing a centralized security domain service eliminates the need for all Cloudproxy Hosted Systems in a security domain to maintain lists of trusted hardware or trusted programs and speed distribution, maintenance and upgrade.

Typically, Hosted Systems in the same security domain will share keys used to protect data that may be used on many Host environments. As discussed below, when software is upgraded or a new Hosted System in a security domain is added, these keys can be shared based on policy-key signed directives with upgraded or new systems in a controlled but flexible way eliminating the danger that data might become inaccessible if a particular Cloudproxy Host System is damaged or unavailable.

**The Extended Tao**

Given the Cloudproxy paradigm that is almost universally employed. The Tao contains some additional support functions.

DomainLoad is used to store and retrieve Program Certificates and sealed data.

Extend allows a Hosted System to extend its Principal Name with arbitrary data. For example, rather than having a policy embedded in a program image, a Hosted System can extend its name with a policy key it reads and the new Principal Name will reflect this value thus rooting the “inter security domain” policy.

There are helper functions to build and verify Program Certificates, perform common crypto tasks like key generation.

There is also support for authorization in the Tao. 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.

**Hardware roots of Trust**

Cloudproxy requires that the lowest level system software must be measured by a hardware component which must also be able to provide attest services and seal/unseal services (and optionally some hardware assist for isolation of Hosted Systems). Absent hardware protection, remote users have no principled way to trust the security promises (isolation, confidentiality, integrity, verified code identity) since “insiders” might silently change security critical software or steal keys.

Cloudproxy supports TPM 1.2 and TPM 2.0 as hardware roots of Trust for Host Systems booted on raw hardware. We have implemented support for other mechanisms and adding a new hardware mechanism is relatively straightforward provided the new hardware supports accurate measurement of booted images, isolation for Hosted Systems and attestation which is used to initialize the key hierarchy.

Once the base Host System is safely booted on a supported hardware, Cloudproxy implements software support for recursive Host systems at almost every layer of software including:

1. A Host System running under a VMM which isolates each Hosted Virtual Machines.
2. A Host System running in an operating system which isolates Hosted Processes (or applications).
3. A Host System running in an operating system which isolates Hosted Operating Systems or Containers.
4. A Host System running in an application (like a browser) which isolates Hosted sub-applications, like plug-ins.

In all cases, Hosted Systems have the same interface to the Host System and can use any Host Service (for example, any system call on Linux) so the programming model at each Hosted layer is essentially unchanged from the non-Cloudproxy case.

**Workflow**

A fairly complete example is in go/apps/fileproxy.

Here we describe a very simple Hosted System running on a Linux Host System with Cloudproxy support called Simpleexample. Complete instructions for the Linux installation is included below. The complete code for simpleapplication is in go/apps/simpleexample and an briefer annotated description of the code is included below.

SimpleExample follows the Tao Paradigm. There are two workflows, one for when you run your program for the first time on the target hardware/OS (“Initialization Phase”) and a second for “post initialization” activation of the Cloudproxy application (“Normal Processing Phase”). During the first phase, your application will produce several files (sealed data, Program Certificate); if these files are present and your application can successfully unseal the sealed data, you do not need to do any further initialization.

*Initialization Phase*

For simplicity, let’s assume all your Cloudproxy applications operate in a single security domain; for example, they may all implement a single coordinated transaction service for an entity which has a public key Ppolicy that is used to authenticate all credentials and service directives.

In this phase, your application generates a public private key pair that will authenticate your program to other Cloudproxy applications. It then asks the the Host System to attest to your newly generated public key. This attestation is transmitted to a trusted security domain service which examines the evidence in the attestation naming your application code measurement and Host System evidence, if this complies with the security domain policy, the security domain service uses the private portion of Ppolicy which we denote by ppolicy to sign a “Program Certificate” for the public key you provided. The security domain service must run in a secure location since the safe use of ppolicy is critical to the overall Cloudproxy security. When you receive the Program Certificate you use the Host System Tao to seal the private key producing an encrypted, integrity protected blob that can be used by your Host System, in subsequent invocations, to give you back this private key. The Host System ensures that your program, and only your program, when running under an isolation guarantee provided by the Host System, can obtain this unsealed information. Often, you will also generate symmetric keys that your program will use to encrypt and integrity protect information (including other keys) and potentially negotiate shared keys with other Cloudproxy services or (to protect data that will be used by your application and other Cloudproxy applications in this security domain on this and other machines can use to encrypt and integrity protect data they should all have access to). Finally, you will seal this additional “private data” along with a monotonic counter available locally or remotely accessible from other Cloudproxy services enabling you to prevent “rollback” attacks on this critical state information. You should store all the resulting “sealed” files and certificates so you can retrieve them on subsequent invocations.

*Normal Processing Phase*

In the Normal Processing Phase, when you first enter main, you will typically retrieve all the sealed blobs produced in the Initialization Phase, unseal them and proceed with your normal processing. You can use your sealed keys to encrypt and integrity protect all data you store or transmit. In the course of normal processing, it is likely you will want to cooperate with other Cloudproxy applications securely. Cloudproxy provides support to open an encrypted, integrity protected channel between Cloudproxy applications (locally or remotely) using your Program Key. These channels *authenticate* the identity of peer Cloudproxy applications you can ensure you share important data and rely on interactions over this channel with security domain authorized, trusted programs that are isolated and protected by a Host System meeting the security requirement of the security domain.

**Simple example**

We provide annotated sample go code for a simple example containing all the critical Cloudproxy elements. A full working version of the example is in go/apps/simpleexample.

Three application elements:

1. Simple Client
2. Simple Server
3. Simple Security Domain Signing Service

Common code used by the client and server is in simplecommon.go.

The Simple Server makes up a secret and stores it. The Simple Client uses a Tao channel to contact the Simple Server to learn the secret and store it securely. We don’t implement rollback protection or distributed key management for intermediate secrets just to keep the example as simple as possible. We give sample code in Go and C++ for the Simple Client and Simple Server. The sample application also uses a Simple Security Domain Signing Service which checks the measurements in the Attestations for the Simple Client and Simple Server and, if the measurements are correct, it signs the Program Certificate. We did not provide sample code for this since the Simple Security Domain Signing Service is, well, simple and need not run on Cloudproxy (although there are good reasons for doing so --- see the key management scenarios below for some reasons). The Fileproxy application (in ~/cloudproxy/go/apps/fileproxy) implements a more complicated service and includes a skeleton rollback server and signing server.

**Installing Cloudproxy**

Complete instructions for Linux installation which allows you to run Simple application is here.

When your ready for installation instructions for a VMM, look at the installations instructions for KVM here.

**The API – Go**

Domains represent security contexts; they encapsulate configuration information like names, 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. The call is:

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.

[The](http://github.com/jlmucb/cloudproxy/tao) [API used by](http://github.com/jlmucb/cloudproxy/tao/auth) a Hosted System:

* [GetRandomBytes(chil](http://github.com/jlmucb/cloudproxy/tao/auth)[dSubp](http://github.com/jlmucb/cloudproxy/tao)[rin auth.SubPrin, n int) (bytes []by](http://github.com/jlmucb/cloudproxy/tao/net)[te, err error): returns a slice of n random bytes.](http://github.com/jlmucb/cloudproxy/tao)
* 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.
* 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 network interface for the Tao channel:

* 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)

**The API -- C++**

***Simple Client in Go (annotation)***

var simplecfg = flag.String("../simpledomain/tao.config",

"../simpledomain/tao.config", "path to tao configuration")

var serverHost = flag.String("host", "localhost",

"address for client/server")

var serverPort = flag.String("port", "8123",

"port for client/server")

var serverAddr string

func main() {

// This holds the cloudproxy specific data for this program

// like Program Cert and Program Private key.

var clientProgramObject simpleexample.ProgramPolicy

// Parse flags, etc (omitted)

// Load domain info for this domain

simpleDomain, err := tao.LoadDomain(\*hostcfg, nil)

if err != nil {

log.Fatalln("simpleclient: Can't load domain")

// Subsequent error checking omitted.

}

var derPolicyCert []byte

if simpleDomain.Keys.Cert != nil {

derPolicyCert = simpleDomain.Keys.Cert.Raw

}

// Extend my name.

err := simpleDomain.ExtendTaoName(tao.Parent())

e := auth.PrinExt{Name: "simpleclient\_version\_1"}

err = tao.Parent().ExtendTaoName(auth.SubPrin{e})

// Retrieve my name.

taoName, err := tao.Parent().GetTaoName()

// Get my keys

sealedSymmetricKey, sealedSigningKey, programCert, delegation, err :=

simplecommon.LoadProgramKeys(\*simpleClientPath)

…

// Unseal my symmetric keys, or initialize them.

var symKeys []byte

if sealedSymmetricKey != nil {

symKeys, policy, err := tao.Parent().Unseal(sealedSymmetricKey)

…

} else {

symKeys, err :=simplecommon.InitializeSealedSymmetricKeys(

\*simpleClientPath, tao.Parent(),

simpleclient.SizeofSymmetricKeys)

…

}

// Remember to zero my keys.

defer simplecommon.ZeroBytes(symKeys)

// Get my private key if present or initialize them.

var signingKey \*tao.Keys

if sealedSigningKey != nil {

signingKey, err = simplecommon.SigningKeyFromBlob(tao.Parent(),

sealedSigningKey, programCert, delegation)

…

} else {

signingKey, err = simplecommon.InitializeSealedSigningKey(

\*simpleclientPath, tao.Parent(), \*simpleDomain)

…

}

// Get the program cert.

\_ = clientProgramObject.InitProgramPolicy(derPolicyCert, taoName.String(),

\*signingKey, symKeys, programCert)

// Parse policy cert and make it the root of our heierarchy.

policyCert, err := x509.ParseCertificate(derPolicyCert)

…

pool := x509.NewCertPool()

pool.AddCert(policyCert)

// Open the Cloudproxy channel.

tlsc, err := taonet.EncodeTLSCert(signingKey)

…

conn, err := tls.Dial("tcp", serverAddr, &tls.Config{

RootCAs: pool,

Certificates: []tls.Certificate{\*tlsc},

InsecureSkipVerify: false,

})

…

ms := util.NewMessageStream(conn)

// Get Tao name of Server.

// Send a simple request and get response.

/\*

rule := "Delegate(\"jlm\", \"tom\", \"getfile\",\"myfile\")"

log.Printf("simpleclient, sending rule: %s\n", rule)

err = fileproxy.SendRule(ms, rule, userCert)

…

status, message, size, err := fileproxy.GetResponse(ms)

…

\*/

log.Printf("simpleclient: Done\n")

}

***Simple Server in Go***

var simplecfg = flag.String("../simpledomain/tao.config", "../simpledomain/tao.config", "path to simple tao configuration")

func serviceThead(ms \*util.MessageStream, clientProgramName string,

serverProgramPolicy \*simplecommon.ProgramPolicy) {

// How do I know if the connection terminates?

for {

log.Printf("clientServiceThead: ReadString\n")

strbytes, err := ms.ReadString()

terminate, err := resourceMaster.HandleServiceRequest(ms, fileServerProgramPolicy, clientProgramName, []byte(strbytes))

if terminate {

break

}

}

}

func server(serverAddr string, prin string, derPolicyCert []byte,

signingKey \*tao.Keys, serverProgramPolicy \*simplecommon.ProgramPolicy) {

var sock net.Listener

log.Printf("simpleserver: server\n")

// Setup Policy root for verify.

policyCert, err := x509.ParseCertificate(derPolicyCert)

pool := x509.NewCertPool()

pool.AddCert(policyCert)

tlsc, err := taonet.EncodeTLSCert(signingKey)

conf := &tls.Config{

RootCAs: pool,

Certificates: []tls.Certificate{\*tlsc},

InsecureSkipVerify: false,

ClientAuth: tls.RequireAnyClientCert,

}

sock, err = tls.Listen("tcp", serverAddr, conf)

// Service client connections.

for {

conn, err := sock.Accept()

var clientName string

clientName = "XYZZY"

err = conn.(\*tls.Conn).Handshake()

if err != nil {

log.Printf("simpleserver: TLS handshake failed\n")

}

peerCerts := conn.(\*tls.Conn).ConnectionState().PeerCertificates

if peerCerts == nil {

log.Printf("simpleserver: can't get peer list\n")

} else {

peerCert := conn.(\*tls.Conn).ConnectionState().PeerCertificates[0]

if peerCert.Raw == nil {

log.Printf("simpleserver: can't get peer name\n")

} else {

if peerCert.Subject.OrganizationalUnit != nil {

clientName = peerCert.Subject.OrganizationalUnit[0]

}

}

}

log.Printf("simpleserver, peer name: %s\n", clientName)

ms := util.NewMessageStream(conn)

go serviceThead(ms, clientName, serverProgramPolicy)

}

}

func main() {

var serverProgramPolicy simplecommon.ProgramPolicy

// Some initialization skipped.

// Load CloudProxy domain configuration.

simpleDomain, err := tao.LoadDomain(\*simplecfg, nil)

// Get policy cert for this domain.

var derPolicyCert []byte

if simpleDomain.Keys.Cert != nil {

derPolicyCert = simpleDomain.Keys.Cert.Raw

}

err = simpleDomain.ExtendTaoName(tao.Parent())

// Extend my name.

e := auth.PrinExt{Name: "simpleserver\_version\_1"}

err = tao.Parent().ExtendTaoName(auth.SubPrin{e})

taoName, err := tao.Parent().GetTaoName()

// Get my keys and certs (or initialize them).

var programCert []byte

sealedSymmetricKey, sealedSigningKey, programCert, delegation, err :=

simplecommon.LoadProgramKeys(\*simpleserverPath)

// Get my symmetric keys.

var symKeys []byte

// Make sure my keys are zeroed.

defer simplecommon.ZeroBytes(symKeys)

if sealedSymmetricKey != nil {

symKeys, policy, err := tao.Parent().Unseal(sealedSymmetricKey)

} else {

symKeys, err = simplecommon.InitializeSealedSymmetricKeys(

\*simpleserverPath, tao.Parent(),

simplecommon.SizeofSymmetricKeys)

}

// Get my Program Key.

var signingKey \*tao.Keys

if sealedSigningKey != nil {

log.Printf("retrieving signing key\n")

signingKey, err = simplecommon.SigningKeyFromBlob(tao.Parent(),

sealedSigningKey, programCert, delegation)

} else {

signingKey, err = simplecommon.InitializeSealedSigningKey(\*simpleserverPath,

tao.Parent(), \*simpleDomain)

programCert = signingKey.Cert.Raw

}

taoNameStr := taoName.String()

\_ = serverProgramPolicy.InitProgramPolicy(derPolicyCert, taoNameStr,

\*signingKey, symKeys, programCert)

err = server(serverAddr, taoNameStr, derPolicyCert, signingKey,

&serverProgramPolicy)

log.Printf("simpleserver: done\n")

}

***SimpleDomainService in Go***

***Some Common code in Go***

Error code and some helper functions omitted.

const SizeofSymmetricKeys = 64

type ProgramPolicy struct {

Initialized bool

TaoName string

ThePolicyCert []byte

ProgramSigningKey tao.Keys

ProgramSymKeys []byte

ProgramCert []byte

}

func (pp \*ProgramPolicy) InitProgramPolicy(policyCert []byte, taoName string,

signingKey tao.Keys, symKeys []byte, programCert []byte) bool {

log.Printf("InitProgramPolicy\n")

pp.ThePolicyCert = policyCert

pp.TaoName = taoName

pp.ProgramSigningKey = signingKey

pp.ProgramSymKeys = symKeys

pp.ProgramCert = programCert

pp.Initialized = true

log.Printf("InitProgramPolicy done\n")

return true

}

// RequestTruncatedAttestation 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.

func RequestKeyNegoAttestation(network, addr string, keys \*tao.Keys,

v \*tao.Verifier) (\*tao.Attestation, error) {

…

}

tlsCert, err := taonet.EncodeTLSCert(keys)

…

conn, err := tls.Dial(network, addr, &tls.Config{

RootCAs: x509.NewCertPool(),

Certificates: []tls.Certificate{\*tlsCert},

InsecureSkipVerify: true,

})

…

defer conn.Close()

// Tao handshake: send client delegation.

ms := util.NewMessageStream(conn)

if \_, err = ms.WriteMessage(keys.Delegation); err != nil {

return nil, err

}

// Read the truncated attestation and check it.

var a tao.Attestation

…

ok, err := v.Verify(a.SerializedStatement,

tao.AttestationSigningContext, a.Signature)

…

return &a, nil

}

// returns sealed symmetric key, sealed signing key, DER encoded cert, delegation, error

func LoadProgramKeys(path string) ([]byte, []byte, []byte, []byte, error) {

\_, err := os.Stat(path + "sealedsymmetrickey")

…

\_, err = os.Stat(path + "sealedsigningKey")

…

\_, err = os.Stat(path + "signerCert")

…

sealedSymmetricKey, err := ioutil.ReadFile(path + "sealedsymmetricKey")

…

sealedSigningKey, err := ioutil.ReadFile(path + "sealedsigningKey")

…

derCert, err := ioutil.ReadFile(path + "signerCert")

…

ds, err := ioutil.ReadFile(path + "delegationBlob")

…

return sealedSymmetricKey, sealedSigningKey, derCert, ds, nil

}

func CreateSigningKey(t tao.Tao) (\*tao.Keys, []byte, error) {

log.Printf("CreateSigningKey\n")

self, err := t.GetTaoName()

k, err := tao.NewTemporaryKeys(tao.Signing)

…

publicString := strings.Replace(self.String(), "(", "", -1)

publicString = strings.Replace(publicString, ")", "", -1)

details := tao.X509Details{

Country: "US",

Organization: "Google",

CommonName: publicString}

subjectname := tao.NewX509Name(details)

derCert, err := k.SigningKey.CreateSelfSignedDER(subjectname)

…

cert, err := x509.ParseCertificate(derCert)

…

k.Cert = cert

s := &auth.Speaksfor{

Delegate: k.SigningKey.ToPrincipal(),

Delegator: self}

…

if k.Delegation, err = t.Attest(&self, nil, nil, s); err != nil {

return nil, nil, err

}

…

return k, derCert, nil

}

func InitializeSealedSymmetricKeys(path string, t tao.Tao, keysize int)

[]byte, error) {

unsealed, err := tao.Parent().GetRandomBytes(keysize)

…

sealed, err := tao.Parent().Seal(unsealed, tao.SealPolicyDefault)

…

ioutil.WriteFile(path+"sealedsymmetrickey", sealed, os.ModePerm)

return unsealed, nil

}

func InitializeSealedSigningKey(path string, t tao.Tao, domain tao.Domain) (\*tao.Keys, error) {

k, derCert, err := CreateSigningKey(t)

…

na, err := RequestKeyNegoAttestation("tcp", \*caAddr, k,

domain.Keys.VerifyingKey)

…

k.Delegation = na

pa, \_ := auth.UnmarshalForm(na.SerializedStatement)

var saysStatement \*auth.Says

if ptr, ok := pa.(\*auth.Says); ok {

saysStatement = ptr

} else if val, ok := pa.(auth.Says); ok {

saysStatement = &val

}

sf, ok := saysStatement.Message.(auth.Speaksfor)

if ok != true {

return nil, errors.New("says doesnt have speaksfor message")

}

kprin, ok := sf.Delegate.(auth.Term)

…

newCert := auth.Bytes(kprin.(auth.Bytes))

k.Cert, err = x509.ParseCertificate(newCert)

…

signingKeyBlob, err := tao.MarshalSignerDER(k.SigningKey)

…

sealedSigningKey, err := t.Seal(signingKeyBlob, tao.SealPolicyDefault)

err = ioutil.WriteFile(path+"sealedsigningKey", sealedSigningKey,

os.ModePerm)

err = ioutil.WriteFile(path+"signerCert", newCert, os.ModePerm)

delegateBlob, err := proto.Marshal(k.Delegation)

return k, nil

}

func SigningKeyFromBlob(t tao.Tao, sealedKeyBlob []byte, certBlob []byte,

delegateBlob []byte) (\*tao.Keys, error) {

k := &tao.Keys{}

cert, err := x509.ParseCertificate(certBlob)

…

k.Cert = cert

k.Delegation = new(tao.Attestation)

err = proto.Unmarshal(delegateBlob, k.Delegation)

…

signingKeyBlob, policy, err := tao.Parent().Unseal(sealedKeyBlob)

…

k.SigningKey, err = tao.UnmarshalSignerDER(signingKeyBlob)

k.Cert = cert

return k, err

}

func GetResponse(ms \*util.MessageStream) (\*string, \*string, \*int, error) {

…

}

func PrintResponse(status \*string, message \*string, size \*int) {

…

}

func SendResponse(ms \*util.MessageStream, status string, errMessage string, size int) error {

…

}

func SendProtocolMessage(ms \*util.MessageStream, size int, buf []byte) error {

…

}

func GetProtocolMessage(ms \*util.MessageStream) ([]byte, error) {

…

}

***Simple Client in C++***

***Simple Server in C++***

**Upgrade and key management scenarios**

Since sealed material is only provided to a Hosted System with exactly the same code identity that sealed the material while running on the exact same Host System, while isolated by that Host System. You may be worried about lost data or limitations that affect key management, software upgrade or distribution on other Host Systems. In fact, it is rather easy to accommodate all these circumstances, and many others, efficiently, securely and in most cases automatically using Cloudproxy although the Cloudproxy applications must make provisions for this during development.

Below are sever example key management techniques that can be used when a Cloudproxy application is upgraded, a new Cloudproxy application (in the same security domain) is launched, or as applications migrate to other Host Systems. All these mechanisms preserve the confidentiality and integrity of all Cloudproxy applications and data used and produced in the security domain or particular host hardware.

There is a discussion of many of the mechanisms as they might affect client software used across different security domains by users with no control over the application code while supporting consumer transparency (the most challenging case) in [4]. Here we restrict ourselves to cooperating server applications.

1. Vanilla cloud key mgmt
2. Prod-id equivalent management
3. Auto upgrade using authority
4. Auto upgrade
5. Certificate based key transfer to new version
6. Shared keys without central authority

**References**

**[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing,** <http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf>.

**[2] CloudProxy Source code,** [http:/](http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf)/github.com/jlmucb/cloudproxy.

**[3] TCG, TPM specs,** <http://www.trustedcomputinggroup.org/resources/tpm_library_specification>

[4] **Beekman, Manferdelli, Wagner,** AsiaCCS, 2016.

**The Guard**

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.

1. To do this, the Host System must be isolated and have access to secrets only it knows. The foundation for this are primitives hardware provides to the “base” Cloudproxy systems it boots. [↑](#footnote-ref-1)
2. Initramfs will have security critical code like the service that implements the Tao so it must be measured along with the kernel image to provide an accurate identity for the “running Linux OS.” [↑](#footnote-ref-2)