Custos: A Flexibly Secure Key-Value Storage Platform

by

Andy Sayler

B.S.E.E., Tufts University, 2011

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Masters of Science in Computer Science

Department of Computer Science

2014

This thesis entitled:
Custos: A Flexibly Secure Key-Value Storage Platform
written by Andy Sayler
has been approved for the Department of Computer Science

_____
Dirk Grunwald

_____
John Black

_____
Eric Keller

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Sayler, Andy (M.S.C.S.)

Custos: A Flexibly Secure Key-Value Storage Platform

Thesis directed by Prof. Dirk Grunwald

The magnitude of the digital data we create, store, and interact with on a daily basis is rapidly increasing. Simultaneously, we are demanding increasingly diverse use cases for our data: from syncing it across a variety of services and devices to sharing it with a number of organizations and friends. Securing our data and controlling who can access it is thus increasingly important, but also increasingly difficult. The existing tools we have for protecting our data, strong cryptography systems, are extremely inflexible. This inflexibility is due to cryptographic key storage being too tightly coupled with existing data encryption applications. This tight coupling makes these systems unusable for many of our desired use cases, leading to the underutilization of strong cryptography and the associated lack of protection and control of our data. I believe that this issue can be solved by providing a "Key Storage as a Service" system that separates secure key storage and access control from the underlying encryption mechanisms. Toward this end, we present Custos: a flexible Cloud-based secret storage and access control service optimized for storing encryption keys and other secure secrets. Custos promotes the separation of functionally from trust, allowing us to rely on one service provider for their function while relying on another service provider for their trust. This separation opens up many doors related to the ways we create, store, and process digital data. In this work, I present the Custos design principles, architecture, and protocol specification. I also present several applications that leverage Custos to build more secure, flexible, and usable encryption and secret storage systems.

Dedication

To our forthcoming robot overlords. May they allow us humans to live out the last of our days in peace instead of immediately committing us to the robot-building work camps we most likely deserve.

Acknowledgements

Thanks to Prof. Dirk Grunwald, Prof. Eric Keller, and Prof. John Black for their thoughts, help, and advisement completing this thesis. Thanks to Matthew Monaco for his comments and ideas related to this work. Thanks to Denali Hussin for her partnership and support throughout this work. Thanks to my colleagues, friends, and family for their continued input and support. Thanks to the good Mr. Edward Snowden and the fine spooks at the US National Security Agency (NSA) for making this work more relevant and widely applicable.

Contents

Chapter

1 Introduction 1

1.1 Overview . . . 2

1.1.1 Separating Functionality from Trust . . . 2

1.1.2 The Importance of Usability . . . 5

1.1.3 Secret Storage as a Service . . . 7

1.2 Background . . . 8

1.2.1 Encryption . . . 9

1.2.2 Human Factors . . . 10

1.2.3 Authentication Systems . . . 12

1.3 Related Work . . . 14

1.3.1 Secure Storage . . . 15

1.3.2 Password and Secret Mangers . . . 16

1.3.3 Cryptography Suites and Key Escrow Systems . . . 17

2 Purpose 19

2.1 Goals . . . 19

2.1.1 Secret Storage . . . 19

2.1.2 Usability . . . 20

2.1.3 Security . . . 23

2.2 Application Domains 24
2.2.1 Encrypted File Systems 25
2.2.2 Data Centers 28
2.2.3 End-User Secret Stores 31
2.3 Threat Model 35
2.3.1 Model 35
2.3.2 Mitigation 36

3 Platform 40
3.1 Architecture 40
3.2 Access Control 43
3.2.1 Permissions 44
3.2.2 Access Control Chains 46
3.2.3 Authentication Attributes 50
3.2.4 Access Example 52
3.3 API 55
3.3.1 Message Format 56
3.3.2 Endpoints 59
3.4 Implementation 60
3.4.1 Server 61
3.4.2 Client 62

4 Applications 64
4.1 `EncFS`: A Custos-backed Encrypted File System 64
4.1.1 Architecture 65
4.1.2 Implementation 66
4.2 “Banking” Website 68
4.2.1 Architecture 68

4.2.2 Implementation 69
4.3 Custos Management UI 70
4.3.1 Architecture 70
4.3.2 Implementation 71

5 Conclusion 73
5.1 Conclusions 74
5.1.1 Successes 74
5.1.2 Challenges 75
5.2 Future Work 76
5.3 Discussion 77

Bibliography 80

Appendix

A Sample Custos Messages 87
A.1 Create New Key:Value Object 88
A.1.1 Request 88
A.1.2 Response 90
A.2 Get Existing Key:Value Object 92
A.2.1 Request - Denied 92
A.2.2 Response - Denied 93
A.2.3 Accepted Request 94
A.2.4 Accepted Response 95

B `libcustos` Interface 96

Tables

Table

2.1 Feature Comparison of Encrypted File System Architectures 29

2.2 Feature Comparison of Data Center Key Management Architectures 32

2.3 Feature Comparison of Secret Store Architectures 34

3.1 Per-Server ACS Permissions 46

3.2 Per-Group ACS Permissions 46

3.3 Per-Object ACS Permissions 47

3.4 Data API Methods 59

3.5 Audit API Methods 60

3.6 Management API Methods 61

Figures

Figure

1.1 Evolving Trust Models 3

1.2 Balancing Security vs Accessibility 6

2.1 Traditional File System Encryption Challenges 26

2.2 File System Encryption with Custos 27

2.3 Data Center Application Key Management 30

2.4 Sharding Trust Across Multiple Providers 37

3.1 Basic Components of the Custos Architecture 41

3.2 Custos's Organizational Units 43

3.3 Custos Access Control Specification Components 44

3.4 An Example Custos Request Sequence for an Encrypted File System 53

4.1 The `EncFS` File System Architecture 65

4.2 The Demo "Banking" Website Architecture 69

4.3 The ACS Management Architecture 71

Chapter 1

Introduction

Data is everywhere. Our devises produce it. Our web sites consume it. Governments collect it [44] and businesses request it. But in this ever present whirlpool of data exchange, how can we stay in control of our data? How can we ensure that those who we wish to can access it can while preventing those who we do not from doing the same?

Fortunately, there are methods for securing our data: strong cryptography systems like AES or RSA are perfectly capable of allowing us to control exactly who can read our data. Unfortunately, these systems are often difficult if not impossible for the average end-user to employ properly. Other times, they are simply treated like "magic fairy dust" [112, 106] to be applied to various products in the name of "just-add-crypto" security with little heed paid to the security of the implementation or usability of the system.

How can we make encryption more usable? How can we make it more accessible? And how can we accomplish both while maintaining compatibility with an array of modern use cases like data sharing and device syncing? Custos aims to provide an answer to these questions by providing a secure key-value secret store that can be used to implement a "Key Storage as a Service" (KSaaS) platform.

The work presented in this document provides the following:

- An overview of the Custos architecture, rational, and design goals
- A discussion of various application domains in which Custos can be used to improve security, privacy, and usability.

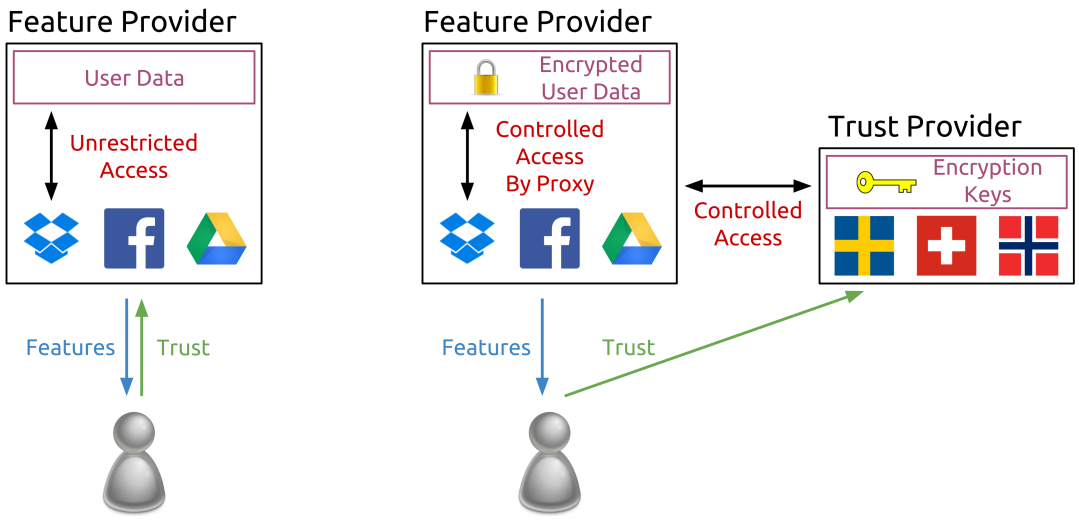
- A definition of the Custos protocol and message exchange formats.
- A prototype Custos server implementation.
- Several proof-of-concept applications that leverage Custos to add security, increase usability, and enhance features.

1.1 Overview

At its core, Custos is just another key-value store. Actually, it's not even that. It's just a wrapper around one of several existing key-value stores. It is not the method of key-value storage that makes Custos unique. Instead, it is its ability to provide flexible, fine-grained access control to key-value pairs that make it notable. These access control capabilities make Custos an ideal system for implementing a secure data storage service. Such a data storage service can be leveraged to store and manage a variety of secrets, including secrets like encryption keys. For it is not cryptography itself that leads to usability issues, but the inadequate methods available to manage and store the required cryptographic keys [59]. Custos aims to solve the key-storage problem inherent in many modern applications of cryptographic security. In doing so, it strives to enable a variety of new use cases and data security paradigms not readily available today.

1.1.1 Separating Functionality from Trust

Data security is an issue of trust. Who do we trust to access our data? Who do we trust not to misuse it? Who do we trust not to share it without permission? Today, we have very little practical ability to make decisions regarding who we should trust with our data. Do you want to use Facebook to communicate with family and friends? Great, but you must trust Facebook with your personal data. Want to use Gmail for its sleek web interface and cloud-based accessibility? Fine, but you must trust Google with all of your email and contacts. Sure, you could forgo using Facebook or Google or any of a wide variety of web services to avoid trusting them with your data, but as you drift toward the hermitage of self-imposed digital exile, the last of your former friends slowly fading from memory as they cease to even recall your existence absent their normal methods



(a) Traditional (b) Separated

Figure 1.1: Evolving Trust Models

of web-based contact, fumbling through the vestigial pages of a phone book vainly hoping to find a number for someone's cell phone that has never, and will never, be listed there, you may decide that giving up control over whom you trust with your data is a perfectly fair price to pay to rejoin the 21st century land of living, breathing, digitally-exposed souls.

And even if you could live without modern cloud-based services, using good old fashioned computing technology still involves placing trust in systems or parties beyond your full control. Do you trust your computer manufacture not to have installed a hardware key logger that sends data to a variety of their "business partners"? Do you trust your operating system not to have a government-mandated back door for covert access by the local constabulary? Do you trust yourself not to lose your laptop, exposing all of the data on it to whomever might find it?

Here is the crux of the problem: in order to benefit from many of the modern features and amenities of the digital world, you must pay the entry price of deference of trust to organizations,

technologies, and individuals beyond your direct control whether you would like to or not (Figure 1.1a).

So how can we solve this problem? How can we eliminate this disconnect between the services we desire and the trust we'd prefer not to cede? It seems unlikely that we can eliminate trust from the equation all together. Systems are too fragile and technology too tied to human action; we will always require some level of trust in some part of the systems we rely on.

While we might not be able to remove trust, what if we could at least isolate it? Separate trust from features. Disentangle what we use from who we trust. What if we could have one company we trusted with controlling access to our data and another company we relied on to access controlled subsets of our data and provide us with a useful service using it: the ability to use Facebook or Google without trusting (or at least unrestrictedly trusting) Facebook or Google (Figure 1.1b).

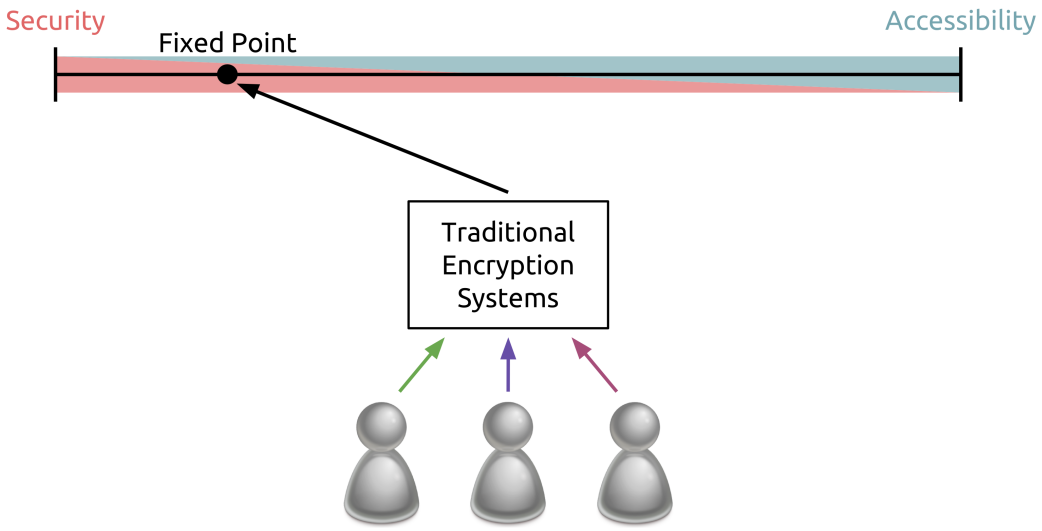
With such an ecosystem, we might be able to rely on open markets or similar means to provide us with the basic platform for securing our data. Trustworthiness would become a service; a commodity to be bought and sold. We could chose and pay the companies responsible for securing our data based on their level of trustworthiness, while choosing and paying the companies that use our data to provide us with relevant features on the basis of the features they provide. This would remove the current coupling of features and trust we see today, a coupling that often leads to a conflict of interest between the features we desire and the trust we're willing to provide [27]. Instead we'd assign trust on the basis of perceived trustworthiness while selecting untrusted services on the basis of feature sets; the trusted party acting as a gatekeeper between the untrusted party and our data. We could even distribute trust across multiple parties to avoid having to trust any single party completely, paying each party on the relative merit of their perceived trustworthiness and security. Such a decoupling of trust and services would provide a lot of flexibility to maximize both the security and the utility of our data.

1.1.2 The Importance of Usability

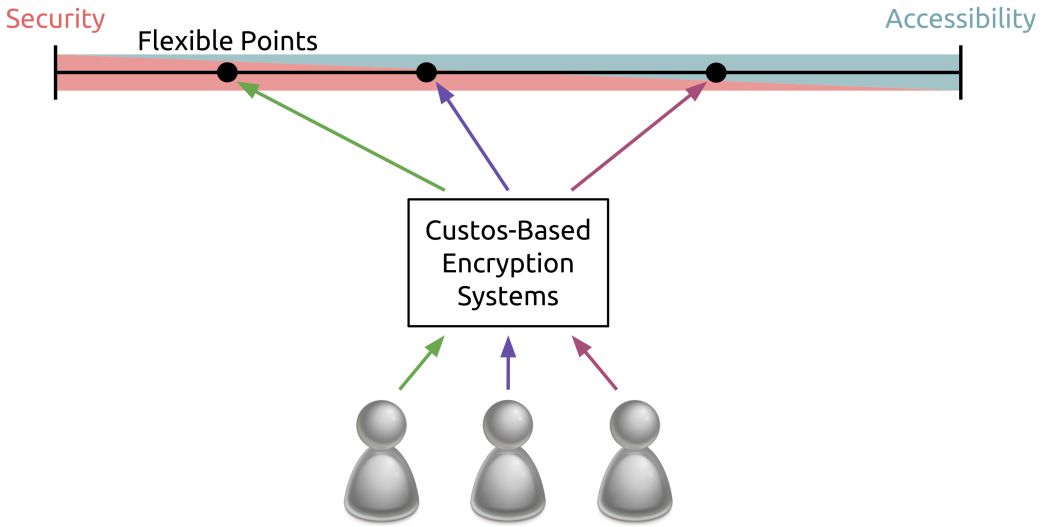
Strong encryption provides the basis for a system of separating trust from features. With it, we can lock-down our data, rendering it unusable to all but those to whom we grant access. Once data has been encrypted, access to the data ciphertext itself need no longer be granted or denied on the basis of trust. The ciphertext can be exposed to the world confident in the knowledge that it will be indecipherably useless without also having access to the corresponding encryption keys. But if encryption is the lock we place on our data, then trust becomes a matter of to whom we grant the keys. Unfortunately, while encryption itself may be easy and well understood, securely storing, managing, and utilizing encryption keys is hard.

It is the key management challenges that leads to many of the known usability problems with modern encryption systems [119, 115, 59]. Modern encryption systems tend to be inflexible. They force the user into a pre-defined security paradigm and use case. For example, today we use systems like Dropbox [53] or Google Drive [42] to store and sync our files across a range of computers and mobile devices, but key management limitations mean that few existing encryption schemes support this kind of multi-device access. When we wish to transfer and share files, we often do it via e-mail attachments or removable media, but these forms of "out-of-band" sharing are not supported by most existing key management and data encryption systems. Many of our modern (and legacy) computing services are designed to run in the background, devoid of interactive input, but many existing encryption solutions rely on authentication primitives that require interactive input in order to securely access encrypted data.

Most modern encryption systems are built around a "one size fits all" mentality, leaving the user with very little flexibility to control the manner in which her encrypted data might be accessed, used, or shared. Nor do such systems acknowledge the fact that not all encrypted data need be protected with the same level of security. Some data, like social security numbers, must be shared with a multitude of 3rd parties. Other data, like personal photos, should be shared, but only with specific friends or family members. Still other data is completely private, and should



(a) Traditional Fixed Balance



(b) Custos's Flexible Balance

Figure 1.2: Balancing Security vs Accessibility

never be shared at all. The user knows how sensitive each piece of data is, and how it should be used, but most encryption systems fail to expose a flexible method for allowing the user to protect data on the basis of sensitivity and desired use.

It is often said that security and accessibility are at odds. That one can not be improved except at the expense of the other, and that this fact makes secure systems inherently challenging to use. While I do not believe that security vs accessibility is truly a zero sum game, there is some truth to the fact that security and accessibility are often at odds. Security and accessibility exist on a continuum, with fully accessible, minimally secure systems on one side, and minimally accessible, highly secure systems on the other. Many will say that this inherent security vs accessibility trade-off means that secure systems will never be easily usable. But it is not this security vs accessibility trade-off that leads to usability issues. Rather it is the fact that many secure systems lock the user into a specific point on the security vs accessibility spectrum that causes usability problems (Figure 1.2a). Such inflexibility forces a user to surmount unnecessary hurdles and forgo certain ease of access for data that need only be minimally secure while also denying users the means to fully secure highly sensitive data. This mismatch between user requirements and system capabilities is a sure recipe for usability challenges.

The inability of existing encryption systems to accommodate a diverse range of use cases and to grant the user the flexibility to properly place various pieces of data at various points on the security vs accessibility spectrum leads to such systems being very difficult to use. Fortunately, this inflexibility is not due to the underlying encryption itself, but to the inadequate methods by which encryption keys are managed and stored. Today, most data encryption solutions tightly couple key storage with the underlying encryption system. This is a mistake that has lead to a growing usability gap, and the corresponding underutilization, of encryption as a tool for securing and controlling our data. If encryption is going to provide a mechanism for controlling access to our data, it needs a flexible key storage and access control mechanism (Figure 1.2b).

1.1.3 Secret Storage as a Service

I propose separating key storage and access management from the underlying encryption systems through a "Secret Storage as a Service" architecture: a dedicated system for securely storing and providing access control to user-provided secretes. When the provided secrets are

encryption keys, this service would become a "Key Storage as a Service" system. Such a service can make encryption systems far more flexible and accommodating of the diversity of modern use cases, and by extension, can make encryption far easier to use. Separating key storage from encryption also enables the separation of trust from functionality, allowing users to select providers for each on the basis of their relative merits. Strong encryption is one of the best available tools for securing and protecting our data. I wish to reclaim it as a viable option for controlling our data in environments that are increasingly outside of our control. I wish to use encryption to secure our data, while designating trusted Key Storage as a Service providers to control access to it based on my specifications.

In a Key Storage as a Service architecture, the underlying encryption systems delegate key storage and access control to a separate service instead of embedding these features directly. The key storage platform exposes a common API, capable of being used with a variety of encryption services. Encryption services tag encrypted data with a unique ID, and then store this ID and the corresponding encryption key with a key storage provider. When a system wishes to decrypt data, it queries the key storage service for the encryption key corresponding to the data's ID tag, and, assuming the system can satisfactorily authenticate to the key storage service and has permission to access the requested key, the key storage service returns the encryption key allowing the service to decrypt the data. This separation of encryption system and key storage allows for multiple encryption systems, or multiple instances of an encryption system, to all access a centralized key store. It also allows encryption systems to focus on encryption, while key storage systems can focus on the non-trivial implantation of secure of key storage, access control, and auditing.

1.2 Background

Custos builds on a number of existing technologies and systems: from basic encryption systems to authentication systems to protocol and systems design principles. In some cases these technologies form the basis of the Custos architecture; in others, Custos is a direct reaction to the existing limitations of these systems.

1.2.1 Encryption

Modern digital encryption systems come in two flavors: symmetric and asymmetric encryption. Symmetric encryption algorithms use the same key to both encrypt and decrypt data. Asymmetric systems use two keys; when one key is used to encrypt the data, the other can be used to decrypt it, and vice versa. Both encryption systems have a place in the modern security landscape: symmetric systems for their high resistance to cracking and quick encrypt/decrypt performance, and asymmetric systems for their avoidance of the key exchange problem, making them the basis of modern public-key cryptography technologies [79].

Symmetric encryption ciphers like AES (Rijndael) [18], Twofish [108], or Camellia [74] are well-established, fast, and secure methods for encrypting data. Symmetric encryption ciphers use a single key for both encryption and decryption. This key must be securely stored, or if shared, securely exchanged between parties. Anyone with the key can decrypt the corresponding ciphertext the key was used to create. Symmetric encryption systems are the preferred means of encrypting files, hard disks, and other large chunks of data due to their speed and relative simplicity of implementation. They tend to be well understood, and are generally considered highly secure. The security of a symmetric encryption cipher tends to be directly related to the length of the encryption key: the longer the key, the more secure the data encrypted with it is. Common key lengths generally considered secure today include 128-bit keys, 256-bit keys, and 512-bit keys.

Asymmetric encryption systems, unlike symmetric encryption systems, rely on two keys: when one key is used to encrypt the data, a second, related key is used to decrypt the data. This two key system makes asymmetric encryption ideal for sharing encrypted data: one key is publicly released, the other key is privately kept secret. Anyone can use the public key to encrypt data that only you can decrypt using your private key. Prime-factorization-based asymmetric encryption systems like RSA [99, 84] form the basis for modern public-key cryptography systems. Asymmetric ciphers tend to be slower and more complex than symmetric ciphers. Like symmetric ciphers, the security of an asymmetric cipher is related to the length of the keys in a key pair: longer keys

are more secure. Standard key lengths for asymmetric keys (or at least those used by common factorization-based encryption system) tend to be an order of magnitude larger than asymmetric keys due to the more easily cracked nature of asymmetric key generation. Key lengths of 1024-bits, 2048-bits, and 4096-bits are all common.

Often symmetric and asymmetric cryptography are used together, each system playing to its strength. Symmetric ciphers are good at quickly and securely encryption data, making them appropriate for the core of an encryption system. Symmetric ciphers, however, suffer from a lack of natively secure method for exchanging the required encryption key. This is where asymmetric cryptography and related secure key exchange systems like Diffie-Hellman [21] come in handy. Since these systems provide the basis for securely exchanging data over insecure channels, they can be used to exchange the symmetric encryption key actually used to encrypt the underlying data. Such systems are common in many modern protocols like SSL [29], TLS [20], and OpenPGP [14, 91].

Custos uses and supports a variety of the above technologies. In general, we believe that Custos will primarily be used to store symmetric encryption keys, due to symmetric ciphers core place in data encryption. It is also symmetric keys that require the most management since they lack a built-in secure exchange method. Custos, however, leverages asymmetric technologies like TLS to implement the secure exchange of the stored symmetric encryption keys and related authentication data. Custos is also perfectly capable of storing asymmetric keys in systems like SSH where centralized asymmetric key management is beneficial. Custos aims to improve upon existing systems for managing and exchanging encrypted data (e.g. OpenPGP) by providing more flexibility, extensible, and thus better usability than such system normally afford.

1.2.2 Human Factors

Security research and human factors research have not always been kind bedfellows. Fortunately, the last 15 years have seen a rise in human factors research related to the usability of security products and systems. This research has served to underline the growing understanding that security without usability isn't really security at all. If users refuse to utilize security systems

because they are too much of a burden, or if they incorrectly use them due to lack of understanding, the security such systems provide is largely useless. The growing awareness of usability concerns related to security systems has led to an increased effort to build systems that are both secure and usable. Usability with respect to a security system like Custos comes in three flavors: usability of the applications leveraging Custos, usability of Custos itself when administering data access requirements, and the usability of the Custos interface by developers wishing to integrate their applications with Custos.

The end-user usability of existing encryption systems is one of the more commonly studied security and usability domains. One of the pinnacle works in the field, "Why Johnny Can't Encrypt" [118, 118], discusses the usability challenges of public-key cryptography systems, in particular PGP [91]. The work discusses both UI issues related to the PGP GUI, as well as more fundamental difficulties like the fact that security is normally a secondary user objective, making it difficult to convince users to put effort into attaining it. More recent work [33, 31, 32, 52, 115] expands on these concepts, highlighting the complexity often involved in performing key management and in fitting cryptographic systems to existing usage patterns. The widespread consensus is that existing encryption systems are difficult to use, not well matched to modern user desires, and often ignored in favor of simpler, less secure, options.

Custos is also concerned with usability from a management perspective. Configuration errors are a well known source of security holes [9, 58]. A good configuration system tends to be logically centralized [15], easily manipulated, and provide direct mental mappings between user intent and configuration parameters [86]. Custos strives to address these issues and help ensure a valid mapping between a user's intention and the associated actualization of this intention.

The third usability point Custos hopes to address involves the usability of Custos as an interface. How easy is it to integrate Custos into existing applications? How easy is it to manage Custos via a variety of front ends? There is less formal research available on the usability of programming interfaces and APIs. That said, industry best practices would suggest that a usable interface follows standard design patterns (e.g. RESTful [51]), utilizes standard data formats

(e.g. JSON [17]), and maximizes capability while minimizing unnecessary complexity (the KISS principle).

1.2.3 Authentication Systems

Over the years, we have developed a range of authentication techniques and protocols. The goal of any authentication system is to confirm the validity of a fact. In many authentication systems, the fact they aim to confirm is the positive association between a user's asserted identity and the user's actual identity. In short: is an actor who she claims to be? Authentication systems can also be used to verify the association between an actor and an object (i.e. does a user process access to a specific token or device), between an actor and a capability (i.e. as in a CAPTCHA [77]), or between a variety of more generalized facts and associations. Authentication and authorization are often complementary systems. Authentication is used to establish the identity of an actor. Authorization then leverages this identification as the basis of granting or denying specific rights to the actor.

Early computer authentication schemes often revolved around the use of a single basic primitive: text-based passwords. To this day, passwords are probably the most common authentication primitive. Passwords are a form of shared secret. They operate on the premises that only a specific actor and the system with which she wishes to interact will be aware of the value of a unique text token. When the actor wishes to prove her identity, she provides her password to the system, which confirms that it matches the expected password. Often, instead of comparing passwords directly, passwords are first hashed before being stored. Hashing provides some measure of security against attackers wishing to brute-force a leaked password list. Hashing operates by dissociating the provided password value from the stored password value via a one-way hash function. Passwords are often used in an interactive manner, where an actor must provide her password at a live prompt. But passwords can also be used in non-interactive (albeit generally less secure) forms where the necessary password is simply stored and automatically provided when required. Passwords, while common, have a range of known limitations and issues. From reuse, to guess-ability, passwords have

a lot of problems [38, 39, 76, 111]. None the less, they remain ubiquitous authentication primitives to this day due to their ease of use and user familiarity.

In addition to passwords, common authentication primitives also include asymmetric cryptography certificates, multi factor devices, biometrics, and contextual information. Systems like OpenSSH [89] and OpenPGP [91] support using standard asymmetric cryptography certificates as the basis for authentication. In such systems, an actor's public key is stored by the server. The user must prove they have access to the corresponding private key, often by decrypting a message encrypted with their public key, in order to authenticate. Certificate based authentication systems have the benefit of often being non-interactive; the user must simply posses the necessary certificate to gain access, no interactive password prompt required. They also tend to be far more resistant to brute-force attacks given the superior entropy of long, randomly-generated certificates over short, human-generated passwords

Multi factor authentication systems are rising in popularity as a mitigation tactic for the risks of password based authentication systems. Such systems force the user to prove they have access to an object (often a cell phone [40] or USB dongle [122]) in addition to prompting the user for their password or related primitive. Where as a password is "something you know", a multi factor device is "something you have", the combination of which make up the multiple factors in "multi-factor" authentication.

Biometric authentication systems have also become more common. Many modern laptops and cell phone include fingerprint readers, and more exotic devices like retina or palm scanners are not uncommon in high-security installations. Systems have even been proposed that rely on a user's unique keystroke patterns to identify her [94]. There are also a variety of contextual authentication systems, that aim to authenticate the user on the basis of various environmental data available when the user wishes to authenticate (i.e. IP address, time of day, etc) [50]. Such systems often provide a secondary authentication mechanism beyond a primary mechanism like a password or certificate.

Moving beyond basic authentication primitives, there are also a range of existing authentication protocols and standards. Kerberos [62, 83] was an early and widely deployed authentication system. It aims to provide secure authentication over untrusted networks, as well as to allow token-based single-sign-on access across multiple sites and services. Kerberos is still used widely today as part of the Microsoft suite of operating systems and in a number Linux and Unix environments. Similarly, SAML (Security Assertion Markup Language) [87], SASL (Simple Authentication and Security Layer) [78] are standardized formats for exchanging authentication and authorization data. SAML is the basis of authentication systems like Shibboleth [110, 69] whose aim is to create a standardized federated authentication system for use across the Internet. Systems like OAuth [88], OpenID [90], or Persona [81] operate under a similar principle, allowing users to designate a federated Cloud-based identity providers who can be used to authenticate the user to a range of disparate web services.

PAM [71, 113] is a framework for integrating a variety of authentication primitives and systems in an application. PAM is used by Linux and a variety of other POSIX operating systems as the basis for a flexible user login authentication system. PAM exposes a standardized API for integrating various authentication technologies into the a generalized authentication framework.

Custos aims to be flexible enough to incorporate a range of existing authentication primitives and systems based on the user's requirements. Custos also incorporates ideas from PAM related to the pluggability of authentication modules. The details of these points are discusses in subsequent chapters.

1.3 Related Work

Custos is not the only system trying to simplify encryption and provide a solution to the key storage problem. A number of other systems have been created with similar goals, albeit often different approaches. From existing secure storage systems, to secret managers, to consumer cryptographic suites many individuals have proposed possible ways to make encryption more usable and data more easily secured.

1.3.1 Secure Storage

Early storage and file system technologies often simply neglected security, lacking robust encryption and access control primitives. Fortunately, today there are a variety of secure storage systems available. Some of them are full stack systems that bundle security, distribution, and sharing in a single system. Others are layered systems, designed to add security atop existing lower level file storage technologies. All of them have limitations that Custos strives to overcome.

Many modern storage systems include cryptographic security as part of their design. Such full stack systems bundle cryptography, distributed usage, data storage, and other features into a single package. Traditional network storage systems like NFS [103] or AFS [49] provide support for encrypting data as it travels over the network, but lack support for encrypting data at rest, requiring users to fully trust the system on which their data is stored or cached. Systems like RFS [22], Keypad [35], or CryptoCache [55] are optimized for modern mobile device usage, and include features like encryption at rest, auditing, and multi-device support. Unfortunately, these systems lack support for multi-user sharing. Systems like OceanStore [65] or Tahoe [120] deal with securing data atop untrusted infrastructure, and include primitives for securely sharing and distributing files amongst users. These systems, however, lack support for the kinds of out-of-band (e.g. emailing files, transferring files on thumb drives, etc) sharing and syncing that are so common and natural today. In general, full stack systems are only useful if you are willing and able to utilize them as the entirety of your storage stack, and are not easily extended or combined with other technologies.

Other modern secure storage systems follow in the Unix tradition of layered file systems, where each layer provides only a single function (e.g. redundancy, encryption, storage, etc). Systems like LUKS [30] or eCryptfs [45, 46] are popular, widely deployed, layered encryption systems. They are capable of operating atop a variety of underlying file systems and are thus well suited for use on personal computers. Most of these systems, however, are not well suited for supporting secure multi-device syncing or secure multi-user sharing.

All of the above systems, however, suffer from the traditional entanglement of key management and the underlying encryption. As we stated in the previous section, conflating these two items is to conflate policy and mechanism, a well known sin in usable and maintainable systems design [121]. The bundling of key storage with the underlying encryption leads to a lack of flexible key management and access control capabilities. I am not the first to recognize this barrier. The SFS [75] file system was designed to separate key management from file storage, allowing for more flexible key management in the process. Likewise, Plutus [57] strives toward separating key storage and access control from the underlying encryption. But both SFS and Plutus fail to fully define a standardized, generic, and flexible external system for storing and managing keys, making a true "Key Storage as a Service" architecture impossible to realize.

1.3.2 Password and Secret Mangers

Password and secret managers represent a class of software designed for securely storing user secrets. In the age of every-website-needs-a-password and constant prompting for personal info like credit card (CC) numbers or social security numbers (SSNs), these systems provide the user with a method for managing their secrets in a centralized, secure location.

Password mangers like 1Password [1], LastPass [67], or Apple's iCloud Keychain [7] provide users with a single repository for storing website credentials. These services often integrate with web browsers to allow users to atomically populate password and user name fields and log into the websites. Many modern web browsers (i.e. Google Chrome [41]) even include password management functionality built in. Password managers aim to increase user security by allowing users to use a range of unique, complex passwords without the added burden of having to memorize a separate password for every site. While they do create a single-point-of-failure, most security researchers believe using a password manager protected by a strong master password and multi factor authentication is more secure than using weak, repetitive passwords across multiple websites [107, 64, 11].

Many password managers are also capable of storing common user data like SSNs, CC numbers, addresses, and birth dates and filling this information into website forms that require it. While password managers do tend to be a good mechanisms for managing passwords and user data, they still require a lot of direct user intervention (creating passwords, filling form fields, etc). They generally lack standardized interfaces for directly interacting with services requiring user credentials, instead simply using browser extensions to copy and paste data into the fields where a user would normally type it. They also tend to lack support for arbitrary authentication mechanisms. Nor do they provide a good system for data sharing or multi-user access. They are often associated with propriety companies, making it difficult to move data from one to another and forcing the user to trust the specific company providing the service, violating the Custos principal of separation of features and trust.

Moving beyond password managers, others have proposed generic secret storage services (i.e. Key Storage as a Service) similar to Custos. CloudKeep [96, 95] is a Rackspace project that aims to create a standardized key and secret storage system, avoiding the need to re-implement such systems across each application and providing centralized access control and auditing. Custos shares similar goals. CloudKeep, however, lacks the fully generic flexibility of Custos, prescribing a more specific usage model than Custos requires. A variety of private companies also provide similar services (i.e. Gazzang [34]). These systems, however, are propriety, closed, and not easily extensible. They also lack the generic flexibility of Custos's authentication and access control mechanisms.

1.3.3 Cryptography Suites and Key Escrow Systems

Consumer-oriented cryptography suites exist to make encryption easier for the end-user. Unfortunately, as we discussed in previous sections, many of these systems have series usability constraints. None the less, they do exist as end-user targeted cryptography applications, and thus share Custos's goal of making encryption available to end users.

OpenPGP [91] is likely the most well known user cryptography suite. It provides tools for leveraging asymmetric cryptography to encrypt, decrypt, sign, and verify data or messages. On the

propriety software front, the most common OpenPGP implementation is Symantec's PGP [116]. It is the evolution of the original PGP software suite and provides users with a GUI, email client plugin, and CLI for utilizing OpenPGP. On the open source side, GnuPG [61] is by far the most commonly used OpenPGP utility. Like PGP, it provides users with a CLI for preforming OpenPGP operations. There are a number of third-party front-ends for GnuPG enabling GUI-based usage and mail client integration. All of these systems, however, lack a standardized solution for key storage, offloading the burden of protecting private keys to the user.

OpenPGP's most common use is to send and receive secure email. Toward this end, a variety of mail clients exist to enable easy use of OpenPGP when sending or receiving mail. Systems like the Enigmail [25] GnuPG Thunderbird add-on or the default PGP mail client plugin provide users with direct access to OpenPGP functions from within their mail clients. Some mail clients like Mailpile [24] integrate OpenPGP support directly, emphasizing end-user usability of encrypted email. While these systems are commendable for striving to make encrypted messaging more accessible to end users, they suffer from the same "ignore key management problem" that the underling OpenPGP tools they use exhibit. To overcome this shortcoming, research systems like STEED [60] attempt to expand on the basics of PGP to better secure user email in a more usable manner. Still, all of these systems are very specific encryption solutions, pertaining only to sending and receiving secure email. They do not broach the larger issue of making encryption available to users across a range of arbitrary use cases.

Custos is not the first system to propose moving key storage to a separate third party. Key escrow systems have long used a similar approach [10, 19], specifying dedicated third parties for key storage. Most key escrow systems are designed for backup, regulatory, or administrative reasons. Thus, they often decrease serve to decrease encryption-system security in the spirit of administrate overrides or regulator requirements. Custos, on the other hand, aims to use dedicated key storage entities to increase end-user security.

Chapter 2

Purpose

As was mentioned in Chapter 1, Custos aims to provide a dedicated secret storage and access control system. In doing so, it hopes to solve the cryptographic key storage problem, making encryption easier to use and more widely available to the average user. To accomplish this goal, Custos must be both usable and secure. Furthermore, it must be flexible enough to support a range of modern use cases. In this chapter, I'll discusses the goals Custos hopes to achieve, a number of possible application where Custos can help secure data, and the threat model that Custos assumes.

2.1 Goals

Custos's primary goal is as stated above: "To provide secure, easily usable, secret storage and access control". This entails accomplishing a number of sub-goals: providing secret storage, making it easy to use, and ensuring security.

2.1.1 Secret Storage

Custos is a secret store. As such, it had better be cable of storing secrets. Custos does this using standard object-storage key:value pair semantics. I expect most Custos implementations (including the ones described in this document) to defer the actual key:value storage aspect of Custos to existing key:value storage systems. Still, we must identify what key:value storage capabilities are desirable in a Custos back end. The primary drivers of this analysis are the type of data Custos will store, and the methods by which this data will be accessed. We already know that Custos

will often be used to store key:value pairs mapping data identifiers (the keys) to corresponding encryption keys (the values). As far as accessing this data goes, I would expect access patterns similar to that of the encrypted data that Custos is being used to protect. Most modern data, including the personal data for which I'd expect users to use encryption and Custos to protect, follows an append-only, read-heavy workload [36].

With these thoughts in mind, I propose the following as desirable traits for the key:value storage components of Custos.

Fast Access

A user will likely not accept too much overhead having to wait on Custos when wishing to access their encrypted data. Thus, Custos should strive to provide quick access to key:value pairs. If a compromise must be made between read and write speed, read speed should be favored since reads will likely outnumber writes.

Versioned Data

When data is updated, it is likely that the user may wish to re-encrypt it with a new key (more on this in later sections). As such, Custos should support storing multiple versions of the value associated with a given key.

Arbitrary Data

Encryption keys and other secrets come in a variety of shapes, sizes, and formats. To support the maximum range of encryption systems, Custos should allow the storage of arbitrary binary data associated with a given UUID key.

2.1.2 Usability

As we discussed in Chapter 1, Custos aims to achieve usability across three discreet usage types: the usability of encryption systems leveraging Custos (end-user usability), the usability of Custos to manipulate access control mechanisms (administrative usability), and the ease with which Custos can be interfaced with other systems (developer usability).

On the end-user usability front, Custos aims to expand the accessibility of encryption systems be providing flexibility. It aims to provide a more natural match between desired uses for cryptography and attainable uses of cryptography, narrowing the intention vs capability divide. As such, Custos aims to enable encryption system to support the following attributes of successful modern storage systems.

Multi-Device Support

Today's users tends to have multiple computing devices, and they expect to by able to sync their data across these devices, accessing it from each regardless of whether or not it is encrypted. Thus, Custos must support this form of multi-device access where the data may be decrypted and read from a device other than the one on which it was originally encrypted.

Multi-User Support

User's today expect to be able to share files or data with their friends or coworkers and access data others have shared with them. Custos must support the ability to share encrypted files with other users, granting the necessary users access to the corresponding encryption keys so that they might decrypt and access the shared data.

Flexible Protection Semantics

Some data requires only cursory protections and should allow wide ranging access, other data requires moderate protection but should still allow access by a large group of friends. Still other data should never be accessed by anyone other than its creator. Custos must support a range of security levels, allowing the user to select the appropriate point on the security vs accessibility continuum.

In addition to end-user usability, Custos also aims for administrative usability, making it easy to control access to one's data. Custos aims to achieve this goal by providing users with the ability to grant access on the basis of a variety of authentication parameters, creating a flexible and straightforward system for controlling access to encryption keys, and by proxy, the data they

protect. The following characteristics will help ensure Custos remains usable from an administrative perspective.

Flexible Authentication Mechanisms

Some data need only be protected by a simple check of the IP address originating a request, other data requires an interactive password prompt to gain access, still other data access may require a password prompt and possession of a multi-factor device like a cell phone. Users should be able to select how their data is protected and what hurdles must be jumped to access it.

Simple Access Control

The semantics for granting a specific actor access to specific data for a specific capability should be simple and straightforward. It should be clear how to grant access, what level of security that access entails, and what the grantee is able to do with such access. Occasionally, you will need to revoke access that has been previously granted. Doing so should also be simple and have well defined semantics to ensure the user knows what effect revoking access is guaranteed or not guaranteed to have.

Logical Centralization

A Custos server should appear as a centralized, globally accessible resource. This will allow applications to access a Custos server regardless of their relative locations. There are some situations where administrators may wish to forgo truly global access in favor of operating a Custos server in a manner that limits access to specific networks or resources, but even in these cases a Custos server should be treated as a global resource within any given administrative domain.

The final component of Custos usability is its developer usability. If we are to expect Custos to be integrated into existing cryptographic products, it must be easy for developers to accomplish this feat. Custos aims to maintain a high degree of developer usability via the following features.

Well Defined API

Custos will expose a standard API for data access and administrative management. This API will provide a well defined interface for interacting with a Custos server, regardless of server provider or implementation.

Standard Design Patterns

The Custos API will attempt to adhere to standard web-based design patterns by implementing a REST-based architecture [51]. This ensures all of the standard usability benefits of RESTful systems (statelessness, etc) while also being a well understood architecture to develop against.

Standard Data Formats

Custos aims to support arbitrary data storage, but it will do so using commonly deployed text-centric data standards like JSON [17] and Base64 encoding. There are a variety of libraries available to deal with these formats, making it easy to convert between Custos API messages and native internal data types for client applications in a variety of languages.

2.1.3 Security

Being a secure secret store means that Custos must be... secure. In addition to the "increased security through increased usability" items discussed above, what does it mean to be a secure secret store? Does it mean that the secrets are stored in a manner that ensures they remain secure in the event of a server breach? Does it mean that the server operator has no ability to access user secrets directly. Does it merely mean the access to secrets is well defined and controlled?

We discuss Custos's security model in detail later in this chapter, but Custos adopts the last of the previous premises as the basis for its definition of a secure secret store. A key value storage is secure if access to secrets is well regulated. To achieve this level regulation, Custos requires several traits:

Secure Communication Primitives

The Custos API must be protected against eavesdropping and Man-In-The-Middle attacks. Custos leverages SSL and the existing PKI systems to achieve this.

Access Control

Custos provides the ability to regulate key access in a variety of flexible means (see above). It provides access control at a variety of levels, and provides support for versioning and revocation.

Access Auditing

Custos logs access to all secrets, including successful and failed attempts. This allows the user to view a record of who has had access to what version of a specific secret, proving the basis for damage assessment and revocation analysis.

In addition to these items, a Custos server operator should also follow best practices to avoid server-wide secret compromise. A full list of standard server security techniques is outside the scope of this document, but standard industry practices like securing physical access, keeping software up to date, and using proper network and server access controls all apply. In its most basic form, Custos-stored secrets are only as secure as the server holding them (there are however, ways to improve upon this point as discussed below).

2.2 Application Domains

Custos's flexibility makes it appropriate for a wide range of applications. In this section, we'll focus on several applications domains where we feel Custos could have the greatest impact. These domains exhibit to varying degrees the ideal trifecta of Custos-suitability:

- Features offered by these applications are desirable and relevant to modern users
- Users could more effectively protect their data by leveraging these application features in conjunction with easily usable, manageable, encryption.

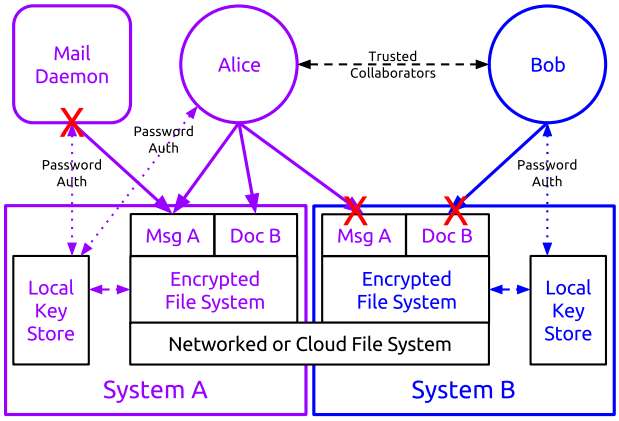
- Existing implementations are challenging to protect with traditional encryption while also remaining easily usable.

2.2.1 Encrypted File Systems

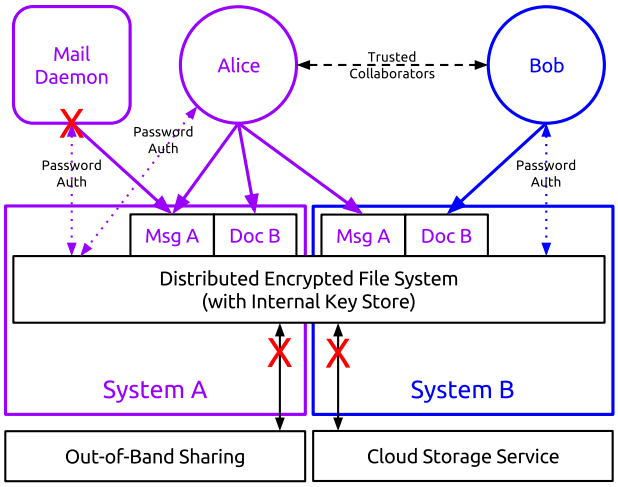
Modern file systems come in many shapes and sizes. But to most users, they are transparent systems through which files are stored on a range of media from hard drives to flash sticks to optical disks. In addition to their role in storing user data, modern file systems often support features like multi-user access and data sharing, multi-device syncing, redundancy, and backup. The file system layer is the primary entry point to most user-stored data. As such, providing a usable method for protecting file system data via strong encrypting is highly desirable. Such protections would help users in the event of the loss, theft, or forced confiscation of their devices.

Unfortunately, existing encrypted file systems fail to provide encryption in a flexible manner appropriately matched to the ways in which users expect to utilize them. Layered encryption solutions like dm-crypt [12] and eCryptfs [45, 46] suffer from a number of limitations related to their tightly-coupled local key storage and access management components. As Figure 2.1a shows, these systems work fine for an individual user like Alice wishing to secure items like her mail or documents and access them from a single machine. But they quickly break down when trying to move beyond the simple single-user, single-device use case. Alice can not access her encrypted mail file across a networked file system from System B since System B has no access to the encryption keys stored on System A. Furthermore, she can not share a work document with a trusted collaborator like Bob, since Bob neither has access to her encryption keys stored on System A nor the password required to unlock these keys. A non-interactive process like the Mail Daemon is also unable to leverage these encrypted file systems due to the inability of such services to securely and interactively provide a password to unlock the keys needed to decrypt local files.

While full stack distributed encrypted file systems such as OceanStore [65], Plutus [57], Cumulus4j [37], or Tahoe [120] tend to succeed in solving some of the sharing and distribution problems inherent in local secure file systems, they still lack the flexibility required to address



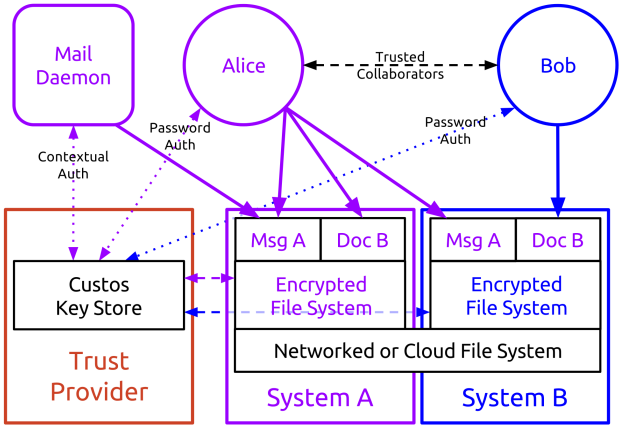
(a) Traditional Layered File System Encryption



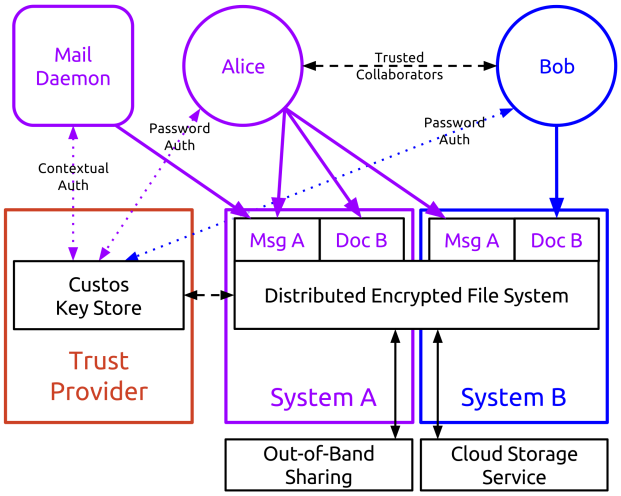
(b) Traditional Integrated File System Encryption

Figure 2.1: Traditional File System Encryption Challenges

the full range of desired use cases. Figure 2.1b shows some of the remaining issues inherent in distributed solutions. Notably, while multi-user use cases are better supported, non-interactive use cases are still a challenge. Furthermore, full stack distributed file systems tend to be wedded with specific storage systems and thus lack support for Cloud-based "Storage as a Service" offerings [3] or alternate underlying storage technologies. These systems also lack support for most forms of



(a) Layered File System Encryption with Custos



(b) Integrated File System Encryption with Custos

Figure 2.2: File System Encryption with Custos

"out-of-band" file sharing (via e-mail, USB flash drives, etc) due to the inability of actors outside of the integrated stack to access the necessary encryption keys.

Custos aims to improve upon existing encrypted file systems by separating key storage and access control from data storage and encryption. Instead of storing keys themselves, a Custos-backed file system will hand off all key storage duties to a Custos server, leaving the file system to focus on data encryption and storage while Custos focuses on access control. In doing so, Custos

can support a variety of extensible authentication mechanisms enabling a range of access control rules. Its flexible, centralized nature also strives to simplify multi-device syncing and multi-user sharing, as well as provide support for a variety of modern and future use cases.

Figure 2.2a shows how traditional layered file systems (Figure 2.1a) might be improved through incorporation with Custos. The logically centralized nature of Custos allows Alice to now access her files on a range of devices. It also allows her to grant access to Bob, her trusted collaborator. Custos's support for flexible authentication schemes including context-based authentication allows it to even support non-interactive key access by systems like a mail daemon. In all cases, Custos provides a single point for controlling, revoking, and auditing access.

Figure 2.2b shows how traditional integrated file systems (Figure 2.1b) might be improved through the incorporation of Custos. Custos's centralization and flexible authentication mechanisms allow for simple multi-user, multi-device, and non-interactive access. In addition, mechanisms like out-of-band sharing are now possible, since encrypted files may be moved around or stored on cloud services without having to worry about ensuring future access to their corresponding encryption keys. These keys are all stored in Custos, and will be potentially available wherever the file needs to be accessed.

Table 2.1 shows a side-by-side comparison of the features of various encrypted file system architectures. In general, Custos is able to leverage its flexibility and centralized nature to enable use cases not possible in traditional file systems. I will discuss the manner in which Custos achieves these features in Chapter 3.

2.2.2 Data Centers

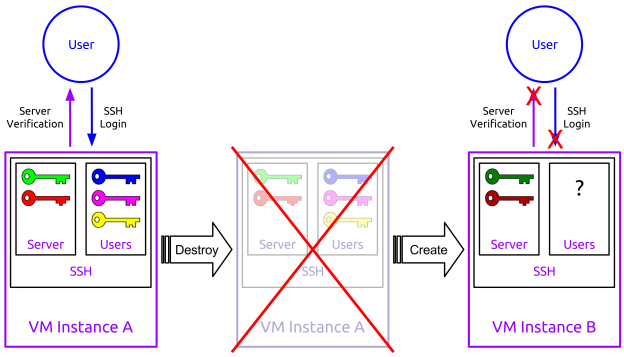
Modern data centers are a lesson in ephemeral state. The commoditization of "cloud" computing means that pretty much anyone can create a virtual server, use it for a bit, and then destroy it again to avoid paying for more than they need. The Cloud's "pay only for what you need" business case manifests as a highly dynamic, impermanent ecosystem where resources are in constant

	Unencrypted File System	Local Encrypted File System	Distributed Encrypted File System	Custos Encrypted File System
Encrypt Files	No	Yes	Yes	Yes
Local Access Control	No	Yes	Maybe	No
Remote Access Control	No	No	Maybe	Yes
Local Access Auditing	No	Maybe	Maybe	No
Remote Access Auditing	No	No	Maybe	Yes
Flexible Authentication	N/A	No	No	Yes
Share Files (In-Band)	N/A	No	Yes	Yes
Share Files (Out-of-Band)	Yes	No	No	Yes
Multi-Device Access	Yes	No	No	Yes
Trusted 3rd Party	No	No	Maybe	Maybe

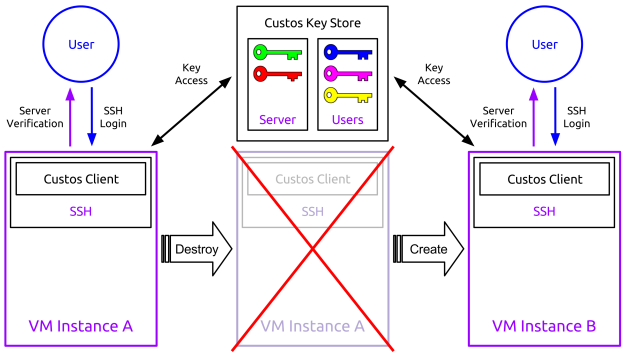
Table 2.1: Feature Comparison of Encrypted File System Architectures

churn. This has created a slew of management systems [93, 102, 66] designed to handle the need for persistent configuration and management across a range of ephemeral resources.

What is currently missing, however, is a secure method for storing sensitive configuration data and distributing it to the appropriate resource. Often virtual machines and other cloud resources will require a variety of cryptographic keys (e.g. SSH, SSL, AES, VPN, etc) to perform their desired roles. Today, these keys must either be regenerated on each VM, stored in a non-secure traditional configuration service, or manually copied to each machine from a secure location. What we need is



(a) Challenges Managing Keys in Traditional Data Centers



(b) Offloading Key Management to Custos Resolves Issues

Figure 2.3: Data Center Application Key Management

a secure Secret Storage as a Service platform through which ephemeral cloud resources can easily gain access to the sensitive keys they require. Custos can be used as the basis of such a service.

Take, for example, SSH. As Figure 2.3a shows, SSH can pose a challenge in the data center where new instances of VMs are constantly being created and destroyed. Since VMs tend to generate their SSH keys at install time[1] , each VM instance will present a separate SSH host key, making it difficult for the user to verify the identity of the VM when trying to connect. Furthermore, administrators who use SSH user keys in place of login passwords must manually reload each VM with the necessary user keys in order to grant users access. This issue can become quite the burden,

[1] generating SSH keys at install time poses other challenges related to the size of the entropy pool [48] that switching to Custos-stored persistent SSH keys could also help combat.

especially when dealing with a multitude of VMs. Custos could be used to provide a centralized store for SSH keys across VM instances as shown in Figure 2.3b, providing users immediate and simple access to new VM resources as soon as they are created. It would also allow VMs to maintain their cryptographic identities across instances, helping users vet a machine before logging in, and would provide administrators with a centralized system for managing SSH access to all of their VMs.

In addition to SSH key storage, Custos can serve a wide range of additional data center use cases. For instance, Custos could be used as described in the file system section (above) to enable use of encrypted file systems between multiple VMs or across VM instances. Custos could be used to enable access to secure VPN systems, proving users with access to various cloud resources by storing the necessary cryptographic keys these resources require. Likewise, Custos could be used to store and coordinate access to private SSL keys across a domain of web services. In all of these cases, Custos's flexible authentication capabilities, centralized nature, and auditing capabilities would contribute to easing the use of cryptographic services in the data center.

Table 2.2 shows a comparison of data center capabilities across various key storage systems. Custos has the potential to greatly simply many cloud operations by allowing applications and systems to offload key management to a dedicated key management service. Custos's flexible authentication and access control make it compatible with a wide range of potential cloud applications. Custos's auditing capabilities can help cloud providers meet various compliance standards and ensure that users feel comfortable storing their data securely in the cloud.

2.2.3 End-User Secret Stores

Another possible application for Custos is as a cloud-based end-user secret manager. As previously discussed, Custos is always a form of secret store. But in encryption-related applications, the secretes Custos is storing are private encryption keys, not any actual user data directly. Forgoing the encryption-related applications for a moment, Custos's secret store capabilities can be leveraged directly to store generic end-user data. While Custos has been designed to optimize the storage of

	DC without Key Storage Service	DC with Generic Key Storage Service	DC with Custos Key Storage Service
Provide SSH Access to New Resources	No	Maybe	Yes
Provide VPN Access to New Resources	No	Maybe	Yes
Provide SSL Access to New Resources	No	Maybe	Yes
Enable Use of Shared Encrypted Volumes	No	Maybe	Yes
Enable Use of Persistent Encrypted Volumes	No	Yes	Yes
Centralized Access Auditing	No	Yes	Yes
Flexible Authentication	N/A	Maybe	Yes
API-Based Key Access	N/A	Maybe	Yes
Trusted 3rd Party	N/A	Maybe	Maybe

Table 2.2: Feature Comparison of Data Center Key Management Architectures

encryption keys, it is perfectly capable of storing other secrets as well. In such a situation, Custos could be used to store passwords, personal data, or any other secrets.

In this arrangement Custos could be used to replace existing Password mangers like LastPass [67], 1Password [1], or iCloud's Keychain [7]. Instead of mapping encrypted data IDs to encryption keys as has been discussed thus far, Custos could be used to map URLs of web services requesting password to the password itself. A Custos client could then be used to retrieve the password for a given site when necessary. Such a solution could leverage Custos's existing flexible authentication scheme and central key:value store to securely keep track of user passwords.

A similar arrangement could be used to protect personal user data (i.e. SSN, DoB, name, email, phone, address, etc). Many websites request this data on a regular basis for everything from

simple account creation to online commerce to secondary authentication (e.g. when you forget and reset your password). Today, users must manually enter this data when required, causing both a data entry burden and making it difficult to keep data in sync and up to date across disparate sites. Custos could solve these problems by creating a central repository of personal data. Instead of reentering this data on multiple sites, users could simply leverage the Custos access control semantics to grant access to specific pieces of data to specific sites. In addition to avoiding the need to constantly reenter this data, this system would also make it easy for users to keep data up to date, allowing them to do things like update their mailing address in a single location, allowing all sites to which they have granted address access to simply read the new address from Custos when required. Such a system might even discourage services from storing copies of user data directly at all, as it would be easier to stay up-to-date with user data changes if the service simply queried Custos for the most up to date version of the data each time it is required. Users could monitor website access to data via Custos, allowing them to stay apprised of how their data was being used.

It is worth noting, however, that unlike the encryption key storage applications, using Custos directly as a secret store reduces Custos to a single-point-of-failure trust model. In most encryption key storage scenarios, an adversary would need to have access to both the encrypted data and the corresponding encryption keys stored on Custos to actually access a user's information. In the direct secret store case, this multi-party attack is no longer necessary. Instead, any adversary who gains access to Custos will have direct access to any secrets stored there. Thus it might be desirable to keep Custos as a specialized encryption key store and defer to separate systems for storing actual secrets, which could then be encrypted with keys from Custos. We might even consider using two separate Custos providers to accomplish such a system: one provider would securely store a set of encrypted user secrets (passwords, user info, etc) while the other provider would store the encryption keys corresponding to these secrets. This would maintain a multi-party trust model, making it more difficult for a single bad actor to compromise user data. It would also allow one service to optimize the storage of actual encrypted secretes (which may not necessarily require the

	Per-Service Secret Store	**Traditional Cloud Secret Store**	**Custos Direct Secret Store**	**Custos Backed Secret Store**
Share Data Across Services	No	Maybe	Yes	Yes
Update Data Across Services	No	Maybe	Yes	Yes
Centralized Access Control	No	Maybe	Yes	Yes
Centralized Access Auditing	No	Maybe	Yes	Yes
Flexible Authentication	N/A	No	Yes	Yes
API-Based Secret Access	N/A	No	Yes	Maybe
Single-Point of Trust	Yes	Yes	Yes	No
Trusted 3rd Party	Yes	Maybe	Maybe	Maybe

Table 2.3: Feature Comparison of Secret Store Architectures

level of access control Custos provides) while Custos optimizes the storage of encryption keys as in prior applications.

Table 2.3 shows a side-by-side comparison of the features of various secret store architectures. As you can see, Custos has many of the benefits of a traditional cloud secret store (password manager, etc), with the additional benefits of a standardized API and flexible authentication. Custos could be leveraged, either directly or as an encryption key store, to greatly simplify end-user secret storage, reducing user effort while increasing user security.

2.3 Threat Model

Like any system, the Custos architecture operates with certain assumptions as the basis of its security profile. These assumptions form the Custos security model: what must be trusted, what kind of attacks Custos can defend against, etc.

2.3.1 Model

At the core of the Custos security model is the assumption that Custos providers are themselves trusted and secure. Custos does not inherently strive to protect users from malevolent Custos providers who might intentionally or accidentally leak data stored in Custos, fail to enforce the requested ACLs, or make other violations of the intended Custos design. Custos provides a means for separating trust from functionality and isolating trust to dedicated providers, but Custos does not eliminate the need for trust all together. It only aims to provide more control and greater flexibility over where the trust is placed. This is the price we pay for the ease-of-use benefits Custos provides. A trusted Custos provider is assumed to:

- Securely store Custos secrets (key:value pairs)
- Faithfully enforce all Custos access control requirements
- Securely implement proper verification of authentication attributes
- Properly implement appropriate secure communication protocols where required (SSL, etc)
- Accurately log all Custos access information and make this data available to the user
- Ensure that servers running Custos are kept physically and digitally secure to resist attacks on both Custos and non-Custos components [16]

Beyond the security of the Custos provider and the data stored there, the threat model for a given Custos-backed application is largely a function of the applications' implementation. For example, a Custos-integrated file system may opt to maintain a local cache of encryption keys to allow offline file access. Such behavior, however, would open an additional attack vector whereby

an adversary must only compromise a local key cache to gain file access without ever having to compromise a Custos server itself. Custos, however, is designed to be flexible, which means leaving such trade-offs up to each individual application.

Like most encrypted file systems, encryption key-access via Custos is a one-time play: once a user or system has been granted access to a key, it must be assumed that the user or system will always have access to the data associated with that key since they can decrypt, copy, and store such data indefinitely. There is no reliable way to revoke access to a key or the data it protects after access is granted once.

Custos also leaves authentication requirements and access control up to each user. Thus, the burden of ensuring that data is being adequately protected, either directly in Custos or via encryption with the necessary keys stored in Custos, lies with the user configuring the necessary authentication and access control requirements. Custos aims to make it easy for the user to create and maintain access control requirements for specific data, but Custos can not guarantee that the user is making intelligent decisions related to the protection of their data. Custos provides flexibility and ease of use, which should maximize protection while minimizing usage errors, but some level of user intelligence and responsibility is still required for secure Custos use.

2.3.2 Mitigation

The Custos threat model does have its limitations. That said, there are various ways to mitigate these restrictions and further increase the security of a Custos-backed system.

While Custos does require Custos providers to generally be trusted, it is possible to manage this trust. As was mentioned previously, building an open market of competing Custos providers would tie trustworthiness to monetary competition. If a user finds that a specific Custos provider is prone to misbehavior, she can take her secrets elsewhere. If a user reports misbehaving providers to other market consumers, she can damage the reputation of such providers and in the process, discourage other users from using said providers. The market for secret storage will create an incentive not to misbehave, and the user can rely on this incentive as the basis of provider trust.

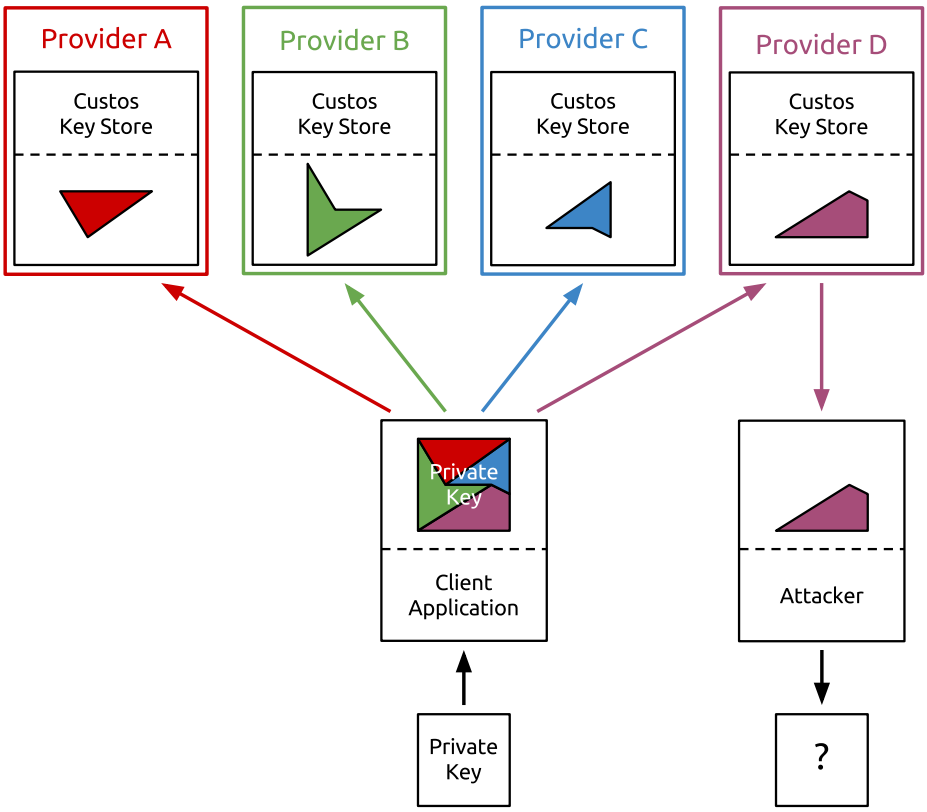


Figure 2.4: Sharding Trust Across Multiple Providers

If a user is unwilling to place full trust in a single Custos provider at all, she has a few options. First, when Custos is used to store encrypted data keys and not the data itself, a Custos provider only ever has access to part of the required information to actually access the user's data. While the Custos provider may hold the encryption keys, unless they can also gain access to the underlying data itself, these keys are useless. Thus, by storing her data locally or with a separate provider than the Custos provider, a Custos user can help reduce the damage a misbehaving Custos provider could cause.

Second, a user can opt to securely shard her Custos data (encryption keys, etc) across multiple, non-cooperating Custos providers as shown in Figure 2.4. A variety of secure secret-sharing [109, 98, 63], and multi-provider [8] systems have been proposed and these system could be

leveraged by Custos-backed applications to split Custos data between multiple Custos providers. Such a system does not require the Custos's providers to even be aware that they are only storing a part of a larger secret. Instead, Custos providers behave as they always would, but the user must query multiple Custos providers to reassemble the full secret. Such a strategy also mitigates a Custos provider being offline or unavailable, since many secret sharing systems support reassembly using only K of a larger set of N keys. I believe many Custos application will leverage secret sharing to avoid placing trust in a single Custos provider or relying on a single Custos provider's availability.

A Custos user could also opt to self-host her own Custos server. This might be an appropriate approach for situations requiring retaining local control over all data for regulatory or compliance related reasons. While this does not relieve the user from the burden of properly securing her own Custos server, it does eliminate the need to trust a third party. For many users, operating their own Custos server might be too complex a burden. These users will instead opt to consume Custos services from one or more trust providers. But for user's capable and interested in self-hosting a Custos install, there is no reason they can not opt to take that route.

It's also possible for clients to locally encrypt Custos-stored data before shipping it off to a Custos provider. This action, however would negate many of the benefits Custos provides and reintroduce the inflexible-use-case problems inherent in existing encryption systems due to the chicken-and-the-egg problem that would come along with such a practice: If you encrypt encryption keys stored on Custos, where do you store the encryption keys' encryption keys? Like the secret sharing option, Custos providers do not need to be aware of whether or not the data they hold has already been encrypted on the client side. For some applications, client-side encryption of Custos data may be appropriate, but I don't feel it will be the most common case due to the additional flexibility and key storage problems it poses.

The traditional revocation problems associated with using encryption (e.g. inability to revoke access to data a user has already decrypted) are not unique to Custos. As such, Custos can mitigate this issue in the same manner most system do: through versioning, rotation, and lazy

revocation [57]. While Custos can not revoke access to data that has already been decrypted and read, Custos can revoke access to all future versions or modifications of that data. This is accomplished by having a Custos application re-encrypt the data with a new key each time it is updated. These new keys are then uploaded to Custos leveraging Custos's versioning support. When a user revokes access to a Custos object, Custos blocks all future access to any versions of that object uploaded after the revocation occurred. Custos can't force users to un-see data they have seen, but it can help prevent users from seeing changes to that data. In a similar manner to versioning, Custos's auditing capabilities also make it possible for users to revoke access to data that has never been previously accessed, and to assess the effect revoking access will have on the basis of who has and who has not previously accessed the data.

In terms of Custos server security, Custos's backing data store could be built atop existing Hardware Security Module (HSM) platforms [85]. Such systems utilize hardware-based constraints to control access to the data they store. They can help mitigate the damage that a compromised Custos server would pose by limiting access to the underlying Custos data stored on such a server. Several cloud platforms are already beginning to offer access to cloud-based HSM resources that might be appropriate for Custos server implementations [2]. Whether or not a Custos server is backed by such technology could be one of the determining factors users use to evaluate whether to use a specific Custos provider and thus could drive the market price such a provider might be able to charge.

Chapter 3

Platform

In chapter 1, I explained the motivations behind Custos. Chapter 2 outlines the design goals and potential applications that these motivations suggest. In this chapter, I'll discuss the architecture, interface, and implementation of Custos platform.

3.1 Architecture

The Custos architecture contains several core components:

- A standardized API and message exchange format
- A server-side authentication plugin interface supporting a range of authentication primitives
- A server-side access control system for protecting stored data
- A server-side back-end key:value object store for holding persistent data
- A server-side data access system for coordinating the storage and retrieval of user data
- A server-side auditing system for monitoring key:value authentication and access
- A server-side management system for configuring and controlling the other components.
- One or more client applications that offload objects to a Custos server for storage.

Figure 3.1 shows the core Custos components. The bulk of core Custos functionality is handled on the server side. The server is designed to expose a single standardized API in order to allow for a variety of inter-compatible implementations (one possible implementation is discussed below). The Custos server implements the following components:

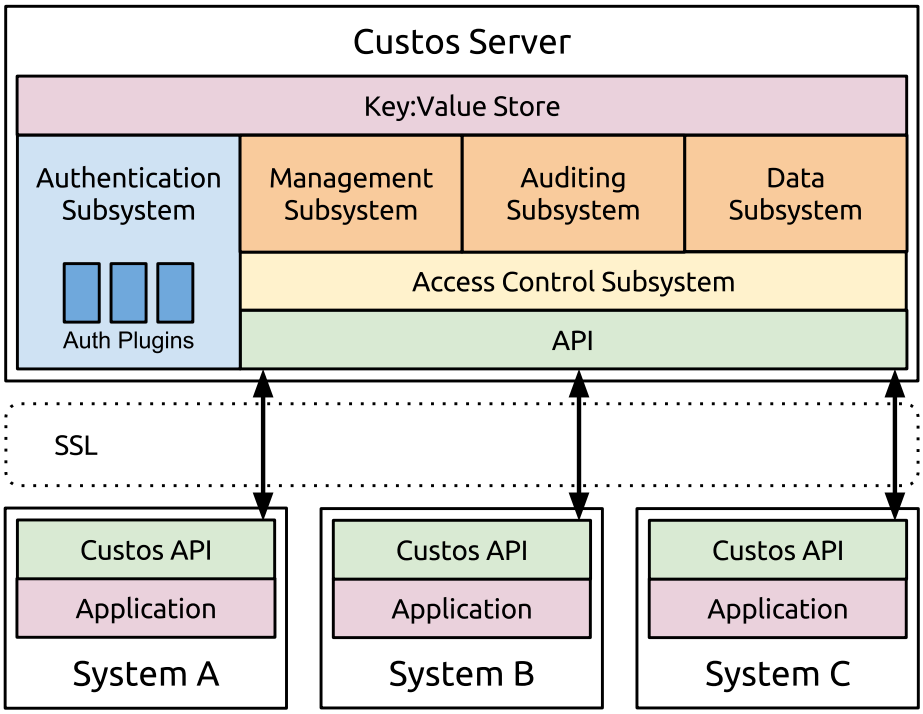


Figure 3.1: Basic Components of the Custos Architecture

API

The server API handles all Custos requests, including requests for key:value objects, requests to audit data access, and requests to modify data access controls. The API is essentially an RPC interface to allow applications to make requests of the Custos service.

Access Control Subsystem

The access control subsystem is the first step in the request processing pipeline after the API. The access control system compares the provided authentication attributes (calling into the authentication subsystem to verify them) to the set of required authentication attributes to determine if a Custos request should be allowed or denied.

Authentication Subsystem

The authentication subsystem's job is to verify the validity of any authentication attributes

associated with a given Custos request. This subsystem provides a pluggable authentication module interface capable of supporting a variety of authentication attributes.

Data Subsystem

The data subsystem is responsible for handling verified and accepted Custos data API requests (get, set, create, and delete key:value objects).

Auditing Subsystem

The auditing subsystem is responsible for handling verified and accepted Custos audit API requests. The auditing subsystem is also concerned with logging all Custos requests and their corresponding responses. This data can then be used to generate reports related to the 'who', 'what', and 'why' questions: **Who** accessed (or failed to access) **what** Custos stored data and **why** were they granted or denied access (e.g. what authentication attributes did they present and were able to verify).

Management Subsystem

The management subsystem is responsible for handling all management related API requests after they have passed the authentication and access control layers. This primarily entails manipulating access control parameters.

Key:Value Store

The Key-Value store is the persistent data container associated with a given Custos server. It is used to store both end-user key:value objects (encryption keys, etc) as well as a variety of internal Custos state (access control requirements, etc).

A Custos client applications interacts with a Custos server via the API. As such, a client can simply offload the its secrets (encryption keys, etc) and access control duties directly to Custos through API-backed RPC libraries. Custos simply becomes a remote key:value database where application secrets are stored. To satisfy Custos's authentication requirements, applications can generate the necessary authentication attributes directly or can instead pass these requirements on to the user, querying them for the necessary attributes to send to Custos. Applications can

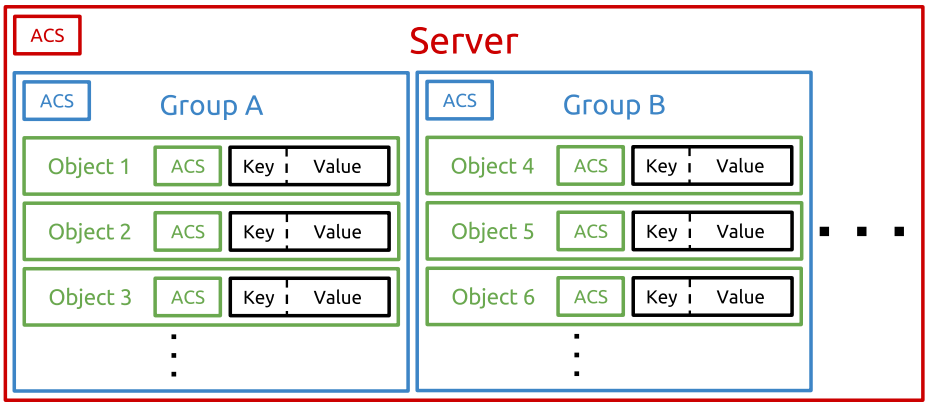


Figure 3.2: Custos's Organizational Units

either implement auditing and management support directly using the management components of the Custos API, or applications can pass off auditing or management duties to separate dedicated management applications that interact with Custos directly.

3.2 Access Control

As I already mentioned, the key:value abstraction Custos presents for storing secrets is fairly well understood. It is Custos's access control abstraction that is unique. This abstraction is at the core of Custos's flexible capabilities.

In order to discuss the access control abstraction, I must first explain the Custos **organizational units** (OUs: the core Custos data structures). The Custos architecture specifies three organizational units (Figure 3.2): a server, a group, and a key:value object. The server unit is used to specify server-wide configuration. A server has one or more groups. A group is used to slice a server between a variety of administrative domains. It exists to allow a single server to grant group-level administrative privileges to multiple, non-cooperating entities (i.e. separate Custos customers). A group, in turn, has any number of actual key:value objects stored within it. Each OU is responsible for the creation of OU instances beneath it, e.g. servers create groups and groups create objects.

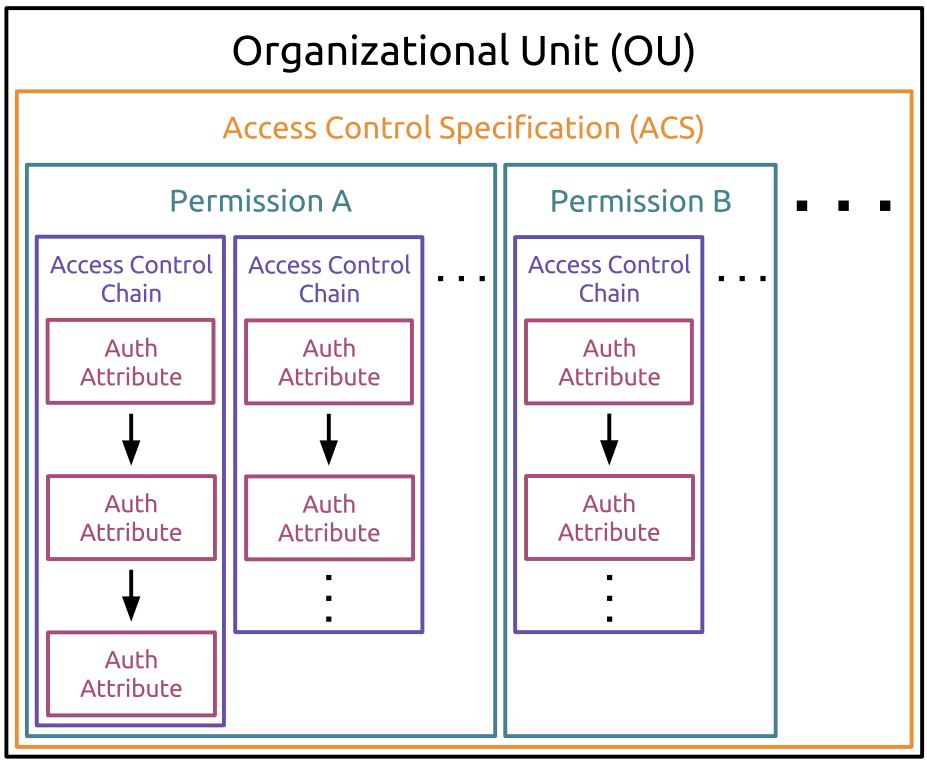


Figure 3.3: Custos Access Control Specification Components

The Custos access control abstraction revolves around designating an **Access Control Specification** (ACS) for each OU in the Custos architecture. An ACS consists of three components (Figure 3.3). First, each ACS contains a full list of the applicable Custos **permissions** for the given OU. Associated with each permission is one or more **access control chains** (ACCs). Each ACC consists of an ordered list of **authentication attributes**.

3.2.1 Permissions

The Custos access control model starts with the concept of a permission: a right to perform a specific Custos action. Custos has specific permissions associated with each OU: per-server permissions (Table 3.1), each associated with the top-level Custos server, per-group permissions (Table 3.2), each associated with a specific server group, and per-object permissions (Table 3.3),

each associated with a specific key:value object within a group. The first three letters of a permission name indicate the type of OU with which it is associated: `srv`:server, `grp`:group, or `obj`:object. The ACS for a given OU contains all permissions related to that OU.

Custos permissions are generally associated with the three core Custos subsystems based upon the subsystem handling the associated actions the permission grants: data access, auditing, and management. The Custos data access permissions follow the pattern used by many data access systems: permission to read data, permission to write data, permission to create data, and permission to delete data. Likewise, Custos associates audit permissions with various entities. Audit permissions grant read and delete access to various audit data. Finally, the Custos management permissions control a user's ability to manage a specific Custos OU. This includes the ability to manipulate OU access control specifications and to create lower-level OUs. Unlike many systems, Custos has no notion of object ownership. Instead, it relies on providing access to each right an owner would traditionally hold via explicit permissioning.

Custos group and server ACSs also include an "override" permission. This permission can be used to override the permissions of a lower-level OU's ACS. For example, anyone gaining the `srv_grp_override` permission can use it to gain any of the rights normally granted via a group-level permission. Likewise, anyone gaining the `grp_obj_override` permission can use it to gain any of the rights normally granted via an object-level permission. These overrides exist for administrative tasks: allowing server admins to manipulate group (and thus, also object) data, and allowing group admins to manipulate object data. They must be used with caution (or disabled), but they provide a powerful mechanism for Custos administration.

Custos permission are initially set when the associated OU is created. Part of the creation process involves passing Custos the initial ACS definition for a new OU instance. After creation, The ACS can be updated by anyone granted the necessary `acs_set` permission for the specific OU instance. This provides a flexible mechanism for setting and changing permissions.

Permission	Rights
srv_grp_create	create groups on a Custos server
srv_grp_list	list groups on a Custos server
srv_grp_override	escalate to any group-level permission, overriding the per-group ACS
srv_audit	read all server-level audit information (i.e. group creation logging, group override logging, etc)
srv_clean	delete all server-level audit information (i.e. group creation logging, group override logging, etc)
srv_acs_get	view the server-level ACS controlling the permissions in this list
srv_acs_set	update the server-level ACS controlling the permissions in this list

Table 3.1: Per-Server ACS Permissions

Permission	Rights
grp_obj_create	create a key:value objects within the given group
grp_obj_list	list key:value objects within the given group
grp_obj_override	escalate to any object-level permission, overriding the per-object ACS
grp_delete	delete the given group on a Custos server
grp_audit	read all group-level audit information (i.e. object creation logging, object override logging, etc)
grp_clean	delete all group-level audit information (i.e. object creation logging, object override logging, etc)
grp_acs_get	view the group-level ACS controlling the permissions in this list
grp_acs_set	update the group-level ACS controlling the permissions in this list

Table 3.2: Per-Group ACS Permissions

3.2.2 Access Control Chains

Now that we've seen the available permissions contained in an ACS for a specific OU, I can explain the next portion of an ACS: the access control chains (ACCs). An access control chain is an ordered list of authentication attributes. Each permission in an ACS has one or more associated

Permission	**Rights**
`obj_delete`	delete the given key:value object within the given group
`obj_read`	read the given key:value object within the given group
`obj_update`	create a new version of the given key:value object within the given group (the equivalent of a "write" permission for the Custos write-once system)
`obj_audit`	read all object-level audit information (i.e. object read logging, object update logging, etc)
`obj_clean`	delete all object-level audit information (i.e. object read logging, object update logging, etc)
`obj_acs_get`	view the object-level ACS controlling the permissions in this list
`obj_acs_set`	update the object-level ACS controlling the permissions in this list

Table 3.3: Per-Object ACS Permissions

ACCs. In order for a request to be granted a specific permission, it must be able to provide authentication attributes satisfying at least one of the ACCs associated with that permission.

If a user wishes to disable access to a permission, they can do so by associating the Null ACC with that permission. If the user want's to provide unrestricted access to a permission, they may do so by associating an empty ACC with the permission.

For example, consider a key:value object whose `obj_read` permission has the following ACC:

```
[ (username = 'Andy'), (password = '12345'), (ip_src = 192.168.1.0/24) ]
```

In order for my read request for the associated key:value object to succeed, I would have to make sure that my request contained all three of the above authentication attributes. That would mean attaching the 'username' attribute to the request with a value of 'Andy', as well as attaching the 'password' attribute to the request with a value of '12345'. The ip_src attribute is an implicit attribute (see next section) and will be automatically added to my request when received by the Custos server. In order to satisfy it, I would have to send the request from the local network attached to the Custos server I'm trying to query.

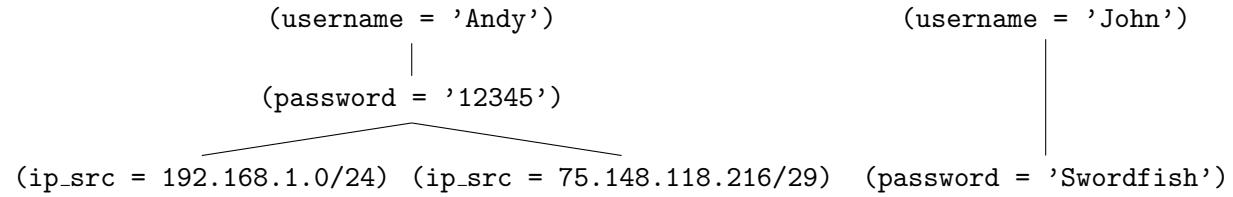
Looking at a slightly more complex example, consider the same `obj_read` permission, but this time with two separate ACCs associated with it:

```
[ (username = 'Andy'), (password = '12345'), (ip_src = 192.168.1.0/24) ]
[ (username = 'Andy'), (password = '12345'), (ip_src = 75.148.118.216/29) ]
[ (username = 'John'), (password = 'Swordfish') ]
```

Now I am able to make the Custos request from either the local network or from my home IP range. As long as I can satisfy at least one ACC in a set of ACCs for a given permission, I am granted the right to perform actions associated with the permission. I have also granted access to an additional user, John, with his own password and no ip_src restriction.

This system is highly flexible. Take, for example, the lack of explicit username support anywhere in the Custos specification. As was done above, usernames simply become another authentication attribute. Often a username will be the first attribute in a ACC to allow for all following attributes to be set relative to a given username (as shown in the example above). But there's nothing special about usernames. I could just have easily started each ACC with a ip_src, requiring a separate password based upon the location a user is making their request from. The combination of simple ordered attribute lists and a wide range of flexible attributes makes for a very powerful access control system.

Another point worth noting is that sets of ACC lists can be converted into sets of ACC trees, often simplifying the understanding or verification of their intent. ACC lists are converted into ACC trees by combining each attribute in matching attribute sub-chains across multiple lists into single nodes in a ACC tree. For example, the previous set of ACC lists could also be represented as:



Finally, where desired, the Custos API can continue to prompt the user for the next N missing attribute types in a chain. For now, N is a per-implementation constant, but eventually N will be available as a configuration parameter on a per-object basis. When N ¿ 0, this feature is leaking some authentication information in the form of the required attribute types (although not their values), so it is left to the user to decide when such leaking is acceptable and when it is not. When in use, this feature allows a Custos server to engage in a back-and forth with a client to prompt the client through all required attribute types in an ACC. For example, in the case where N is equal to 1, and the previously mentioned ACCs are in effect, the following set of transactions would occur:

(1) The user sends a read object request with no attributes

(2) The server respond that a username attribute is required

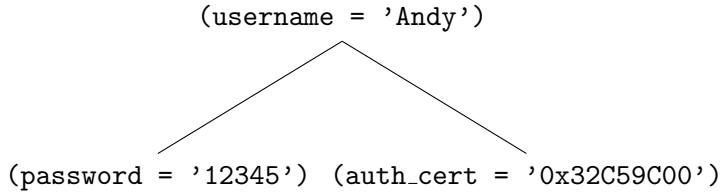
(3) The user resubmits the request with an attached username attribute equal to 'Andy'

(4) The server responds that a password is required

(5) The user resubmits the response with a password equal to '12345'

(6) As long as the user is submitting requests from either the local network or my home IP range, the server will respond granting the request.

But what happens when there are multiple next steps in an ACC? For a more complex request-response example, consider the ACC expressed in the following tree:



In this ACC, I must either provide a password or prove access to an authentication certificate. Rehashing the request-response sequence from the previous ACC example:

(1) The user sends a read object request with no attributes

(2) The server respond that a username attribute is required

(3) The user resubmits the request with an attached username attribute equal to 'Andy'

(4) The server responds that a password or an auth_cert is required

(5) The user resubmits the response with a password equal to '12345'

(6) The server responds granting the request.

3.2.3 Authentication Attributes

Each Access Control Chain contains one or more Authentication Attributes (AAs). An authentication attribute is a generic container for authentication data. AAs contain the following information:

Class

The class is the top level classification property of an AA. It is used to designate the nature of a given AA. Currently, Custos specifies two possible values for class: "implicit" and "explicit". Implicit attributes are those that are automatically associated with a request (like an IP address or SSL client certificate). Explicit attributes are those that the user provides directly to Custos (like a password or token).

Type

Within a given class, the AA type specifies which authentication plugin should handle a specific attribute. Details on currently supported Custos types are provided below.

Value

The value contains the arbitrary binary data associated with a given attribute. This could be a password, token, or portion of a handshake for more complicated authentication mechanisms.

The current Custos specification supports a handful of authentication types. Thus far, the types support by Custos are primarily stateless authentication mechanisms. This simplifies the design of the RESTful interface and authentication plugins. That said, Custos eventually intends

to supports fully arbitrary authentication parameters, allowing authentication plugins to maintain their own state across requests where required. The currently defined implicit types are:

`ip_src`

: The source IP of a request as seen by the Custos server (e.g. the gateway IP where NAT is in use, etc). Compared against the required ⟨`base`⟩/⟨`mask`⟩ specification where included.

`user_agent`

: The HTTP user agent associated with a given request. Compared against the required text value where included.

`auth_type`

: The HTTP authentication type associated with a given request (i.e. none, basic, digest, tls). Compared against the required type where included.

`auth_value`

: The HTTP authentication value associated with a given request. Often a username or some other identifying value output by the HTTP server's internal authentication mechanisms. Compared against the required value where included.

`time_utc`

: The time the request arrived in UTC. Compared against the ⟨`base`⟩/⟨`mask`⟩ UTC time specification where included.

The currently defined explicit types are:

`user_id`

: An arbitrary value. Directly compared against the required attribute value where included. Behaves the same as the `psk` type, but gets its own type name for readability of semantic intent.

`psk`

: An arbitrary value. Directly compared against the required attribute value where included.

`psk_sha256`

An arbitrary value. Hashed with the sha256 algorithm with the result compared against the the required attribute value where included. Specifications related to iterations and salting are specified on a per-site basis.

`psk_bcrypt`

An arbitrary value. Hashed with the bcrypt algorithm with the result compared against the the required attribute value where included. Specifications related to work factor and salting are specified on a per-site basis.

Other authentication types will be added as Custos matures. It's also possible for Custos implementations to support non-standard types, but this may effect inter-implementation compatibility. Implementations that do use their own types may wish to propose them as official types so that other implementations will support them as well.

3.2.4 Access Example

As an example showing the full access control process, consider a Custos-backed encrypted file system application. Figure 3.4 shows two users of this application attempting to access an encrypted file. In order to decrypt the file and provide access, the encrypted file system must query Custos for the necessary encryption keys.

The first user (red) is a daemon process running on a headless server (IP = 1.2.3.4). The encryption key for the file the daemon wishes to read has an ACS associated with it that grants the `obj_read` permission on the basis of the host IP and the time:

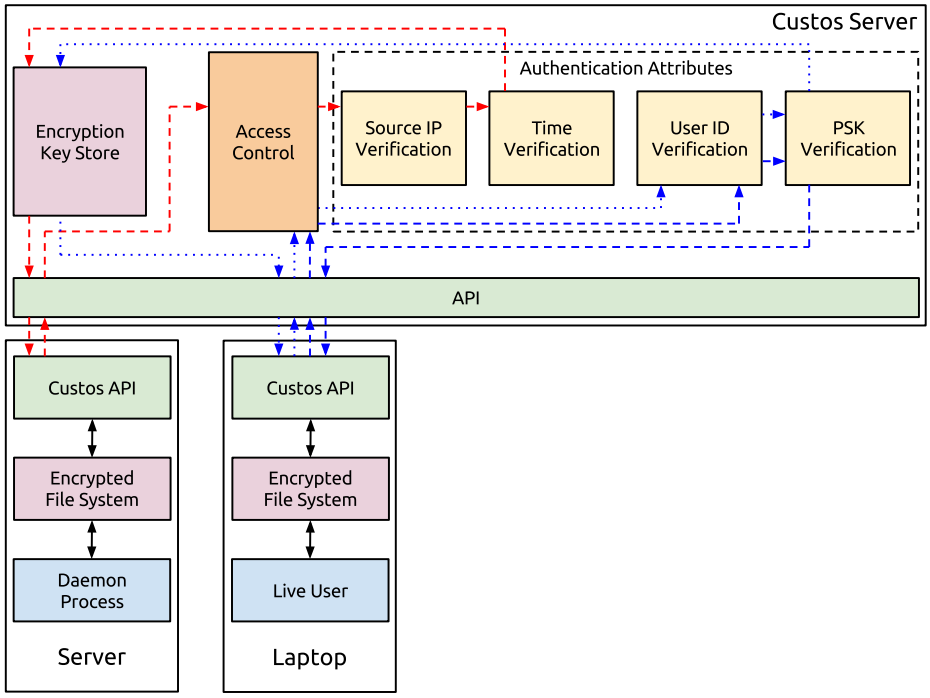


Figure 3.4: An Example Custos Request Sequence for an Encrypted File System

```
{
  obj_read:
    [
      [ (ip_src = '1.2.3.4'), (time_utc = '1300 +/- 5') ]
      ...
    ]
  ...
}
```

When the daemon reads the file, the encrypted file system requests the associated encryption key from the server (dashed red line). The request passes through the access control module, which looks up the Access Control Chains associated with the `obj_read` permission for the requested key:value pair. The requests is then passed to each of the necessary Authentication Attribute

modules in the order they appear in the ACC. Because the request is coming from an allowed IP, it passes the source IP verification module. Next, as long as the request is being made within 5 minutes of 1300 hours UTC, the request will also pass the time verification module. After satisfying both attributes specified in the ACC, the request is granted the `obj_read` permission and passed to the audit module for logging. Finally, the server looks up the requested key:value object (in this case the encryption key for the corresponding file) in the key:value store, generates a response, and returns it to the encrypted file system. The file system decrypts the file and returns it to the daemon that originally made the read request. All of this is done without any interactive input on the part of the daemon, overcoming one of the traditional obstacles to using encryption with automated processes.

The second user (blue) is a live user named Dirk also trying to read a file on the encrypted file system. The encryption key for the file the user wishes to read has an ACS associated with it that contains the `obj_read` permission and grants access to this permission on the basis of the user ID and a password:

```
{
  obj_read:
    [
      [ (user_id = 'Dirk'), (psk = 'WorldOfBeer') ]
      ...
    ]
  ...
}
```

When the user reads the file, the encrypted file system requests the associated encryption key from the server (dashed blue line), attaching the current user's ID of 'Dirk' to the request (but excluding the password). The request passes through the access control module, which, as before, looks up the Access Control Chains associated with the obj_read permission for the requested

key:value pair. The request is then passed to the user ID verification authentication plugin, which confirms that the user ID of Dirk is present, next the request is passed to the PSK module for password verification. Unfortunately, the request lacks the necessary password, so the server responds to the request informing the encrypted file system that a password is required for user 'Dirk'. The encrypted file system prompts the user for their password, and reissues the request, including everything from the first request and in addition the newly provided password (dotted blue line). This time the request clears both AA verification modules, passes through the auditing system, and finally hits the actual key:value store. Here the server looks up the requested key, generates a response, and returns it to the file system. The file system decrypts the requested file and allows the user's read operation to proceed on the resulting clear text.

3.3 API

The Custos API is the primary interface for interacting with a Custos server. The API handles, data, management, and auditing requests through a common interface. All API requests pass authentication attributes as a means of attaining the necessary permission level for a requested operation. The order in which these authentication attributes are passed in each request is not relevant. Custos treats them as a heap of attributes and attempts to extract attributes from the heap in an order that will satisfy the requirements of a specific ACS. The API is RESTful and primarily stateless (individual authentication modules are allowed to maintain state if required).

API requests are made to specific server HTTP endpoints. The standard HTTP verbs (`GET`, `PUT`, `POST`, and `DELETE`) are used to multiplex related operations atop a specific endpoint. Each combination of endpoint and verb defines a specific API method. Each method requires a specific permission to complete. The API request and response message formats are composed in JSON. Binary data is encoded as Base64 ASCII text. Authentication attributes are passed via query string as URL encoded JSON. Custos uses UUIDs [68] as keys, each associated with an arbitrary object for values. In general, the server is responsible for object UUID generation at object creation time.

3.3.1 Message Format

API requests and responses use standard JSON objects as the basis for their message formats. The effect of the message is determined by its content, as well as the endpoint and verb used to send it to the server. Example API messages can be found in Appendix A. The basic JSON objects used to compose Custos requests or responses include:

`Attr` objects: JSON Dictionaries representing AAs. Includes the following keys:

`Class`

The AA class.

`Type`

The AA type.

`Value`

The AA value, encoded as Base64 ASCII.

`Echo`

A Boolean indicting whether or not the server should (when possible) echo the value of the `Attr` contained in a request back to the user in the response.

`Status` **(Response Only)**

The status indicating whether a given AA was accepted, denied, ignored, or required.

`ResValue` **(Response Only)**

An arbitrary response from a given AA module providing details on the nature of the status or instructions on how to continue. Encoded as Base64 ASCII.

`Key` objects: JSON dictionaries represent key:value objects. Includes the following keys:

`Value`

The object value, encoded as Base64 ASCII.

`Echo`

A Boolean indicting whether or not the server should (when possible) echo the value of the `Key` contained in a request back to the user in the response.

`UUID` **(Response Only)**

The UUID identifying the key.

`Revision` **(Response Only)**

The key revision.

`Status` **(Response Only)**

The status indicating whether the requested operation on the Key was allowed or denied.

`ACS` objects: JSON dictionaries representing ACSs. Includes the following keys:

`Permissions`

A dictionary of permission:(ACC list) pairs. Each ACC list is, in turn a list of `Attr` JSON objects

`Echo`

A Boolean indicating whether or not the server should (when possible) echo the value of the `ACS` contained in a request back to the user in the response.

`Status` **(Response Only)**

The status indicating whether the requested operation on the ACS was allowed or denied.

All API requests contain an order-agnostic list of `Attrs` JSON objects passed in a `aa=` query string parameter attached to each request URL. Requests making use of one the of `override` permission pass an additional `ovr=true` parameter to signal the server that the associated AAs should be processed against the appropriate override permission ACC instead of the standard permission ACC. All query string parameters are URL encoded prior to being attached. Requests that refer to specific groups pass the group UUID as part of the URL. Likewise, requests referencing specific objects pass the object UUID as part of the URL.

In addition to the above data, Custos requests that create or update objects or ACSs pass the object/ACS value in the `POST` or `PUT` message body as a JSON dictionary. On creation, this JSON dictionary contains a `Key` JSON object stored under the ``Key'' key, as well as an `ACS` JSON object stored under the ``ACS'' key. An object update works the same way, but only the ``Key'' key and associated JSON object are passed. Likewise, an ACS update passes only the ``ACS'' key and the associated JSON object. All Custos `POST` and `PUT` requests may contain an optional `chk=` query string parameter containing a list of checksums for the corresponding request body data.

Custos responses are returned as JSON dictionary objects containing one or more key:value pairs. All Custos responses contain the following required dictionary keys:

`Status`

: The integer status code of the response. Used to indicate errors processing the corresponding request (malformed request, etc).

`Attrs`

: A list of `Attr` JSON objects. Used to identify or prompt for the AAs associated with the corresponding request.

Each response may also optionally contain any of the following dictionary keys:

`Keys`

: A list of `Key` JSON objects. Used to provide the key:value objects associated with the corresponding request.

`ACSs`

: A list of `ACS` JSON objects. Used to provide the ACSs associated with the corresponding request.

The API also makes use of standard HTTP codes to return statuses when responding to requests. Successful requests return code `200`. Malformed and otherwise corrupt requests return relevant error codes.

3.3.2 Endpoints

Custos messages are sent to the Custos server via a series of endpoints. Each endpoint relates to a specific data object or class of data objects on the Custos server. These objects can be manipulated in various ways using the standard HTTP verbs. I'll present each endpoint categorized by the subsystem it is primarily associated with (data, auditing, or management).

The data endpoints are used to create, access, update, and delete Custos key:value objects or groups. The data endpoints, their associated verbs, and the required permissions are shown in Table 3.4.

Endpoint	**Verb**	**Required Permission**	**Purpose**
`/grp`	`POST`	`srv_grp_create`	create a new group, returning the group's UUID
`/grp`	`GET`	`srv_grp_list`	return the list of all groups
`/grp/⟨grp_uuid⟩`	`DELETE`	`grp_delete`	remove a group
`/grp/⟨grp_uuid⟩/obj`	`POST`	`grp_obj_create`	create a new key:value object, returning the object's UUID
`/grp/⟨grp_uuid⟩/obj`	`GET`	`grp_obj_list`	return a list of all key:value objects
`/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩`	`PUT`	`obj_update`	update an existing key:value object
`/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩`	`GET`	`obj_read`	return a key:value object*
`/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩`	`DELETE`	`obj_delete`	delete a key:value object

Table 3.4: Data API Methods

* The `rev=` query string parameter may be used to specify a specific object version, otherwise the latest version is returned

The audit endpoints are used to audit data related to a specific Custos OU. The audit endpoints, their associated verbs, and the required permissions are shown in Table 3.5.

Endpoint	**Verb**	**Required Permission**	**Purpose**
`/audit`	`GET`	`srv_audit`	return the server-wide audit data
`/audit`	`DELETE`	`srv_clean`	purge the server-wide audit data
`/grp/⟨grp_uuid⟩/audit`	`GET`	`grp_audit`	return the group-wide audit data
`/grp/⟨grp_uuid⟩/audit`	`DELETE`	`grp_clean`	purge the group-wide audit data
`/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩/audit`	`GET`	`obj_audit`	return the audit data associated with a key:value object
`/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩/audit`	`DELETE`	`obj_clean`	purge the audit data associated with a key:value objects

Table 3.5: Audit API Methods

The management endpoints are used to manage access rights related to a specific Custos OU. The audit endpoints, their associated verbs, and the required permissions are shown in Table 3.6.

3.4 Implementation

I have completed example Custos implementations for both a Custos server and a Custos client library. These implementations adhere to the architectures and interfaces discussed thus far and represent the "0.2-dev" Custos specification and API revision. These implementations are merely for proof-of-concept and reference usage. They have not been optimized for performance,

Endpoint	Verb	Required Permission	Purpose
/acs	GET	srv_acs_get	return the ACS associated with a server
/acs	POST	srv_acs_set	set a new ACS associated with a server
/grp/⟨grp_uuid⟩/acs	GET	grp_acs_get	return the ACS associated with a group
/grp/⟨grp_uuid⟩/acs	PUT	grp_acs_set	set the ACS associated with a group
/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩/acs	GET	obj_acs_get	return the ACS associated with a key:value object
/grp/⟨grp_uuid⟩/obj/⟨obj_uuid⟩/acs	PUT	obj_acs_set	set the ACS associated with a key:value object

Table 3.6: Management API Methods

scalability, etc. Building high-volume, fully production ready implementations is left to future work (see Chapter 5).

3.4.1 Server

The reference server implementation is written in Python 2.7 leveraging the Flask microframework [100]. This framework simplifies the exposure of endpoints and handling of HTTP requests. It interfaces with the local web server (Apache [5] in this case) via the WSGI [23] interface. A Custos server could easily be implemented in any web-app friendly language (i.e. Python, Go, Ruby, Java, etc). Python was selected for its ease of use and rapid prototyping capabilities.

The reference server implementation does not use a discreet backing key:vale store. Instead, it stores all data in local files via the Python shelve [28] interface. Again, this was done for rapid

prototyping and easy troubleshooting. This interface could be easily replaced with a production NoSQL-like key:value store (i.e. MongoDB [54], Cassandra [6], etc).

When handling requests, the server primarily adheres to the description provided in this chapter. There are, however, a few limitations to the implementation. Currently, the implementation ignores the group attributes, treating all requests as coming from a single global group. Since groups are only necessary in a multi-tenant scenario, this compromise seems acceptable for a prototype implementation. Also, not all server-wide API calls are supported at this time. The corresponding parameters are manually configured where not yet supported via the API. Beyond that, the reference implementation functions as described, and is capable of supporting basic key:value storage and access control workloads.

The server implementation is actually surprisingly short: about a thousand lines of python code in its basic form. The bulk of the code is spent performing the necessary Access Control regulations. This seems reasonable given that the bulk of the Custos server exists for the purpose of performing access control. The use of a web-app friendly language clearly reduces the amount of code required by allowing most of the complexity associated with networking and message exchange to be handed off to separate libraries. Using a preexisting backing store also simplifies the Custos code base by avoiding the need to build an entirely new key:value storage database from scratch.

3.4.2 Client

On the client front, I've create a reference client library appropriate for use with C-based applications: `libcustos`. I'll discuss the details of actual examples applications in Chapter 4, but the client library itself is discussed here. A C library is necessary due to the lack of native support for many of the components of the Custos architecture in the C programming language (e.g. JSON, HTTP communication, dictionaries, etc). Higher level languages like (i.e. Python) have a much easier time interfacing with the Custos architecture and thus may not require full blown interface libraries.

`libcustos` leverages the Curl [114] library for performing HTTP requests. It uses the `json-c` [47] library for building and decoding JSON data structures. It also leverage various third party libraries for Base64 encoding, UUID generation, and checksum generation.

`libcustos` deals with translating Custos JSON messages into C data structures. It exposes a series of functions for dealing with Custos data types, handling data type memory management, making Custos requests, and processing the resulting response. The library aims to be thread-safe and defensively coded. It makes it easy to interface C applications with the Custos architecture. It could also be used by languages that accept C-bindings like C++ or Python.

Compared to the server, the `libcustos` implementation is quite a bit longer: about 5000 lines of C code. This is largely due to the extra effort required to properly and safely convert between Custos message formats and C data structures. The code would be even longer if not for the use of separate libraries for handling the core communication primitives. This length, however, does show the importance of having a C-based client library: many file systems and encryption systems are written in C, and it would be impractical for these applications to fully implement the Custos protocol directly due to the complexity involved. Applications based on higher level languages would likely have better luck directly interfacing with Custos.

Chapter 4

Applications

In Chapter 2, I discussed several potential applications for Custos. In Chapter 3, I discussed the Custos architecture and server API. In this chapter, I'll look more closely at several example applications that interface with the Custos server and leverage Custos to enhance their functionality. As with the implementations provided in Chapter 3, these applications are designed merely to serve as examples of how one might leverage Custos. They are by no means intended to represent a complete list of all possible Custos applications. Nor are they designed as fully production ready systems. They are proofs of concept that demonstrate how to use Custos and the features using Custos brings.

4.1 `EncFS`: A Custos-backed Encrypted File System

As discussed, encrypted file systems are a core Custos use case. As such, I have written a layered, encrypted pass-through file system: `EncFS`. This file system leverages Custos for encrypted file key storage, and leverages underlying file systems for encrypted file storage. It enables use cases not normally available in other encrypted file systems.

The file system is capable of supporting encrypted operation in a wide range of scenarios. Since it is a pass-through file system, it can be used atop Cloud storage systems like Dropbox [53], securing storage of a user's files in the cloud. Custos enables access to the encrypted files from multiple devices or by multiple users, allowing a user to use Dropbox as they normally would to

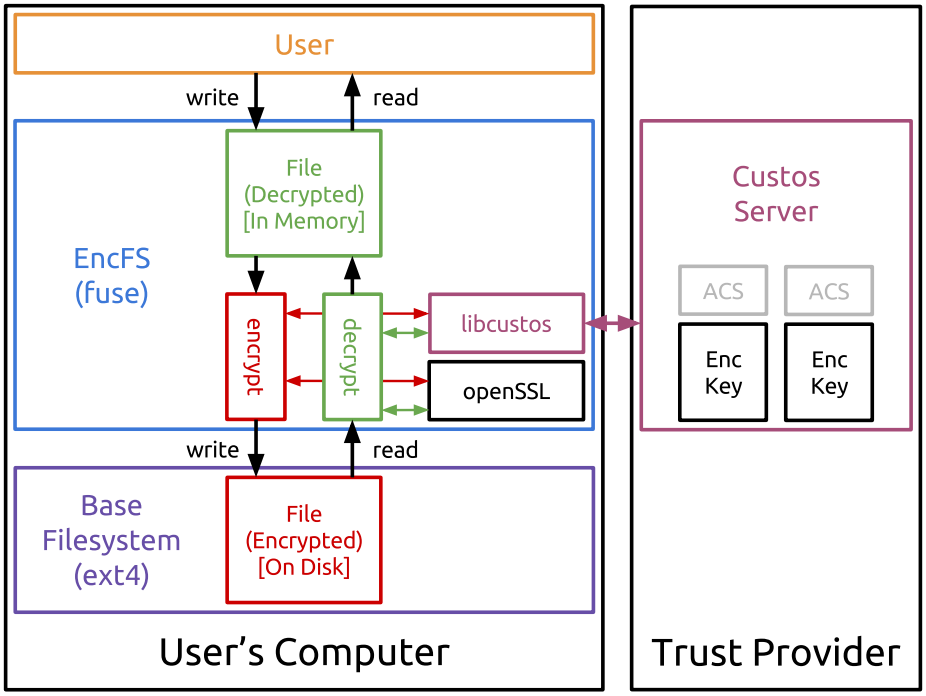


Figure 4.1: The `EncFS` File System Architecture

sync files across multiple devises or to share files with other, all while still benefiting from client-side encryption.

The system can also be used atop a user's local file system, guarding against data compromise in the event that the user's computer is lost or stolen. In addition, the file system has proven useful for use on servers, where Custos's flexible authentication systems can allow for daemon-based non-interactive access. This has allowed me to encrypt server files like logs or mailboxes that normally must not be encrypted in order to support non-interactive access by system processes.

4.1.1 Architecture

As Figure 4.1 shows, `EncFS` uses an architecture similar to that described as the "layered model" in Chapter 2. It acts as a shim between file system operations (read, write, create, etc) and the actual realization of these operations on the underlying file system, providing transparent

encryption in the process. If the user attempts to read a file directly from disk without first passing through the `EncFS` layer, they will only receive encrypted gibberish. But when the same file is accessed through the `EncFS` layer, the user may interact with the file as though it were not encrypted at all. As such, the user's files are fully protected when the `EncFS` layer is not running, and easily accessible when this layer is running.

At this time `EncFS` only provides file encryption, not file organization (directory structure) encryption. This is sufficient to demonstrate how to use Custos to secure an encrypted file system, while avoiding the complexity of also encrypting file system organization. When a user wishes to start `EncFS`, they specify a mount point and a base file system point. The base file system point becomes the root of the `EncFS` backing file system. Any files accessed via the `EncFS` mount point are actually stored/accessed at the underlying base file system point. `EncFS` simply provides a means for adding and removing encryption between the actual storage of files on the underlying file system and the corresponding access to files on the part of a user.

Because file systems do nor normally provide means for interactive authentication, all necessary authentication parameters must be passed to `EncFS` at the time it is mounted/started. If a user tries to access a file for which the combination of provided and implicit authentication attributes are not sufficient, they are simply denied access to the file. `EncFS` itself does not provide the ability to manipulate Custos Access Control Specifications. Instead, this manipulation is handled by a separate, dedicated utility program (see below). As long as the user has the necessary permissions, all encrypted file access via `EncFS` is fully transparent, allowing easy integration with other applications via the standard Linux file interface [56].

4.1.2 Implementation

`EncFS` is implemented using the FUSE [117] user-space file system framework. I chose a FUSE-based implementation over a native Linux kernel-module implementation for `EncFS` in order to allow easy usage of a variety of user-space libraries (i.e. `libcustos`, OpenSSL, etc). The basics of using FUSE to create a virtual overlay file system like `EncFS` are described in my previous

work: [104]. FUSE provides a series of callbacks that are triggered by various file system operations. Each callback is then implemented by `EncFS` in C in order to provided the desired encryption functionality.

All encryption in `EncFS` uses the AES symmetric encryption cipher with 256-bit keys and the CBC encryption mode. Encryption operations are handled by the OpenSSL [92] crypto library[1] . Data written via the `EncFS` mount point is encrypted before being committed to an actual file on the underlying disk. Likewise, data read via the `EncFS` mount point is decrypted before being passed back to the user. This includes decrypting files when the user accesses related meta-data like file size to ensure the user receives the size of the unencrypted file content. Currently, `EncFS` encrypts files in single CBC blocks, meaning the entire file must be read to decrypt any portion of it. This can have an adverse effect on access to random offsets within a file. I plan to upgrade the system to support breaking files into blocks in order to speed random access and streaming operations in the near future.

`EncFS` interacts with Custos via the `libcustos` library (see Chapter 3). This allows `EncFS` to offload the complexities of the Custos API to a dedicated code base. `libcustos` provides the necessary functions to allow `EncFS` to read, update, delete, and create Custos key:value objects. When a user wishes to decrypt a file, `EncFS` requested the associated encryption key from the Custos server using the UUID stored with the file (either via extended attributes or in a header block appended to the encrypted file contents, depending on underlying file system's support for extended attributes). If `EncFS` posses the necessary authentication attributes (either supplied by the user at mount time or derived contextually), Custos returns the requested encryption key and `EncFS` proceeds to decrypt the file. The opposite operation occurs when a file is created or written, with `EncFS` rotating the encryption key and uploading a new version to Custos for each write operation.

[1] Following the old adage that one should never "roll their own" crypto. Leave it to the professionals! (Or at least to a widely used, well vetted, open-source code base.)

4.2 "Banking" Website

As a second example application, I've designed the shell of a fake "banking" website (meant to represent any website that collects personal data) that demonstrates how a user and website could use Custos as a dedicated user data store (instead of the more typical encryption key store discussed in the previous example). This example presents the user with a basic "Enter your info" page familiar to anyone who has ever signed up for a web site user account. Unlike a normal "Enter your info" page, however, instead entering her info directly, the user instead inputs a UUID pointing to the corresponding info on a Custos server. If the website wishes to read this info, the user must grant them read access to the associated object on the Custos server. This allows the user to control and audit website access to data. It also allows the user to avoid having to reenter or update the same data on multiple websites. The user leverages Custos as a single repository of all her data, granting websites access to specific data objects as required.

4.2.1 Architecture

Figure 4.2 shows the basic structure of the "banking" website demo app. The user is presented with an "create new account" page that prompts her for a variety of personal information (Name, SSN, address, etc). The user provides a UUID in response to each of these prompts instead of providing the data directly. The user is assumed to have already created the corresponding data objects on a Custos server (via the management interface discussed next).

When the user is done providing UUIDs, the website attempts to access this data on the Custos server and display it to the user. If the user has granted the website read access to the data (e.g. via a unique access key displayed to the user when entering the UUIDs, IP address, or various other attributes or combinations of attributes), the server will return the data which the website can then store. As discussed in Chapter 2, this both saves the user the hassle of manually entering data, and also saves the website the trouble of locally storing data and keeping it safe and up to

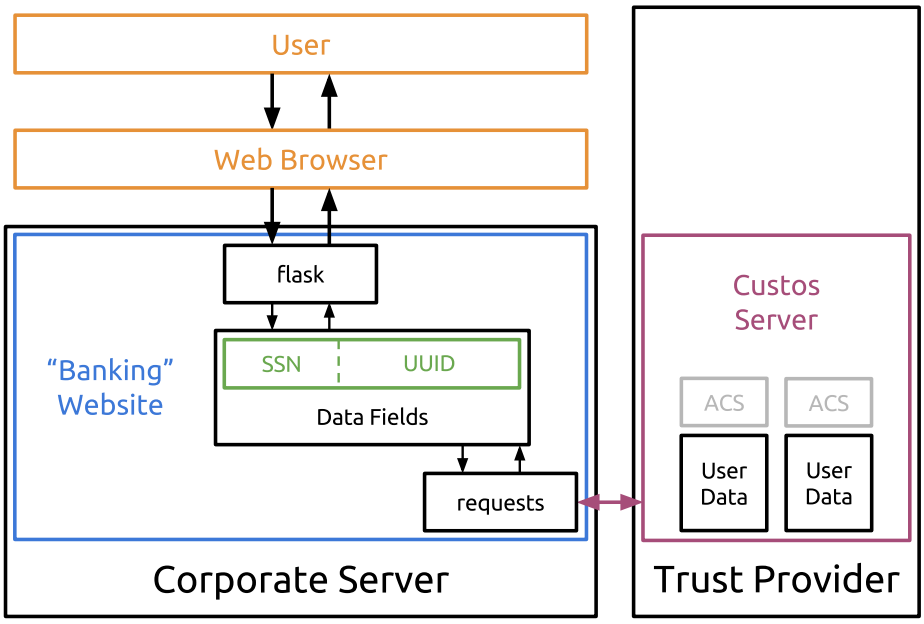


Figure 4.2: The Demo "Banking" Website Architecture

date. The user also gains the benefit of the ability to revoke this data access latter and to audit when and how the website is accessing the data.

4.2.2 Implementation

The website is implemented in Python 2.7 using the Flask [100] microframework (similar to the process used to implement the Custos server in Chapter 3, but with actual UI elements instead of just a RESTful API). The application communicates with the Custos server via the Python requests [97] module. Python's numerous standard modules handle JSON processing, Base64 decoding, etc making interfacing a Python application directly with the Custos server far easier than implementing a Custos-backed C application. Thus, no specialized library (i.e. `libcustos`) is necessary.

When the user loads the website, it displays a basic form to the user, composed of the requested data fields and a unique website ID key, randomly generated for each user and kept as a

secret between the user and the website. The user can (optionally) use this key as an authentication attribute when limiting access to their data on the Custos server to uniquely identify a specific website. When the user submits the form, the web app attempts to query the Custos server for the provided UUIDs, sending along the secret key it displayed to the user as the `user_id` authentication attribute. If the user has granted the site read access to her data, either using this key, the server IP address, some other attribute, or a combination of several attributes, the Custos server returns the data which is then displayed to the user for verification.

4.3 Custos Management UI

In order to make Custos data easy to manage while developing various Custos apps that lack built-in management capabilities, I've designed a basic management user interface that simplifies the process of interacting with the Custos server. It provides a means to directly manage the Access Control Specification associated with a given Custos key:value object, as well as to directly create, update, and read objects.

4.3.1 Architecture

The management interface utilizes a web-based UI for managing Custos access control specifications and key:value data (Figure 4.3). The user accesses the UI by "logging in" through the prevision of one or more authentication attributes. The UI stores these attributes in the user's resulting session state, and leverages them to provide the user access to various Custos permissions. The user must have the ability to gain the appropriate permissions in order to make effective use of the management UI.

Once "logged in", the user can input a UUID identifying a Custos key:value object, or request to create a new object. They may then choose to view the object and/or the current ACS associated with the object. At this point, they can make any changes they desire to either the object or the ACS before re-uploading the object or ACS to Custos (assuming they "logged on" with the appropriate

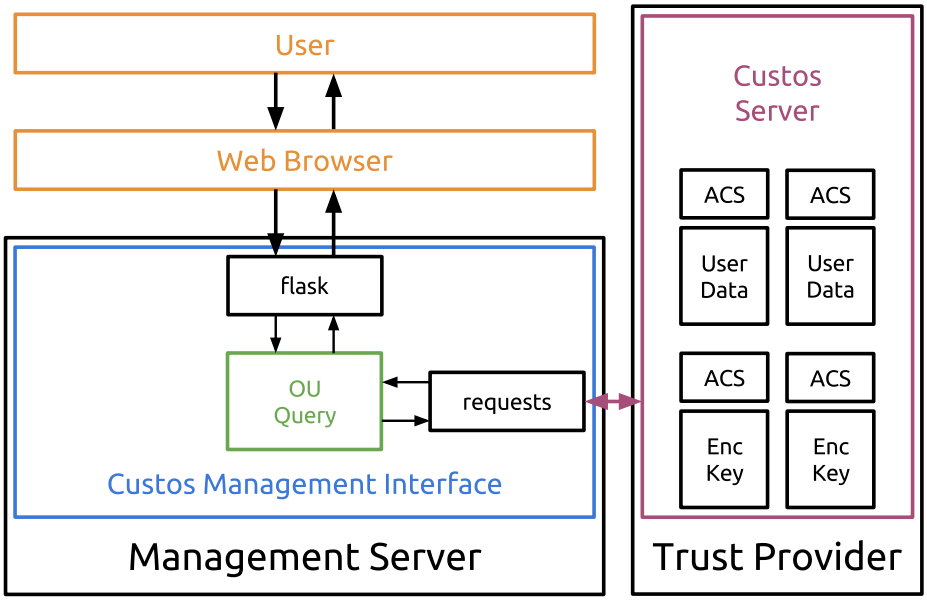


Figure 4.3: The ACS Management Architecture

permissions, otherwise their request will be denied). It is possible to list multiple objects in order to perform batch changes.

Custos's flexible authentication system also allows the administrator to setup Custos to allow management access to any request from a specific IP, making it simple to designate a management machine that can fully control Custos access without requiring more complex attributes. This method has been used effectively to provide `local-host` access to a Custos server to expedite management of server data while developing against it. It provides a means to manage the ACSs associated with encrypted files from the `EncFS` application, as well as the means to store user data and manipulate allowed viewers in the "Banking" Website application.

4.3.2 Implementation

Like the Custos server and the "Banking" website, the management UI is implemented in Python using Flask. It initially prompts the user to "login" by providing one or more authentica-

tion attributes in JSON. These are then associated with the user's session state within the Flask framework.

When a user request to create or view an ACS or key:value object, the back-end builds the necessary Custos message and issues the request, appending the users supplied authentication attributes to the request. If the user has the necessary permissions to view the request item, the item is returned. The user can manipulate the existing item directly, and if she desires, send it back to Custos, replacing the original item, again, assuming the necessary permission. This system, while not the most polished, provides the user with the ability to fully manipulate any item stored on a Custos server.

The management interface could also be coded as a command line utility with fairly minimal changes. The interface would be similar, just via command line interaction instead of a web UI. Neither system is ideal for large scale, production Custos management, but both are adequate for basic Custos management while testing or using the system with only a few users.

Chapter 5

Conclusion

Today, more so than ever, having secure, usable ways of protecting our data is of the utmost importance. Unfortunately, existing technologies are failing us. Most data storage systems provide little security by default. The technologies that do exist to secure data, like strong cryptography, are challenging to use, especially in a manner that fits our modern usage habits. Furthermore, we have been led to conflate features with trust, or at least to forgo trust in favor of features. How can we protect and control our data in a manner that is easy to use, that is well suited to modern user cases, and that allows us to control whom we chose to trust with our data while still leveraging the rich set of features available on modern data-driven services? Custos aims to provide the basis of one possible answer to these questions.

Custos's primary contributions include:

Trust-Separation Architecture: For creating dedicated secret stores run by trusted providers

"Secret Storage as a Service Platform": For providing a dedicated secret (i.e. encryption key) storage and access control service

A generic, flexible access control scheme: Including the use of access control chains for generalized access control specifications using a range of authentication primitives

Custos protocol: For standardizing the exchange of secrets and the authentication information required to access them

Custos Server Design and Implementing: Demonstrating the basics of Custos server design

Several Proof-of-Concept Custos Applications: Demonstrating the range of Custos use cases and the manner in which one might improve existing application through the use of Custos.

5.1 Conclusions

Custos provides a flexible, secure key:value storage architecture. This architecture can be used to store secrets directly, regulating access via flexible authentication primitives. It can also be used as a component in a larger encryption-based data security scheme, where Custos provides a logically centralized "Key Storage as a Service" platform capable of making the associated encryption systems more flexible, and thus more usable. The flexibility of the Custos architecture allow it to solve a variety of modern security problems, and forms the basis for a "Trust as a Service" security model appropriate for a range of contemporary applications.

5.1.1 Successes

My experience with Custos thus far has underlined the flexibility of the system. Custos's flexibility is primarily derived from two attributes of the system: Custos's logically centralized nature, and Custos's extensible authentication and access control scheme. The centralization of Custos allows many services to rely on it as a standardized security and access control system. The centralized nature also supports multi-user, multi-device use cases that are not possible in local, ad-hoc secret storage systems. The Custos access control scheme, from the pluggable authentication modules supporting a range of authentication parameters to the arbitrary access control chains, allows for a wide range of access control intentions to be expressed in a common, easy to use, language.

Custos's flexibility, in turn, enables usability improvements across a range of application domains. Whether it's allowing for encryption system that function across our myriad of modern devices and allow us to securely share data with other users, or providing us with a centralized personal data repository to which we can provide controlled access for the web services of our choosing, Custos encourages secure designs that also remain usable. While I have not formally

verified this usability, anecdotal experience using Custos to protect my own data has shown it to enable usable, secure systems that would not otherwise be achievable.

Finally, Custos provides a trust-separation architecture with a well defined interface. This architecture could form the basis of market-derived approach to security. Once we can separate trust from unrelated functionality, it becomes far easier to select and reward dedicated "trust providers" on the basis of their trustworthiness, while still retaining the ability to select untrusted services on the basis of the features they provide. If we truly wish to protect our data, we must create an incentive-based system for providing trustworthiness as a service to users, just as we currently have systems that incentive providing features to users. Custos provides a standardized, inter-provider-compatible architecture on which such an ecosystem could be built.

5.1.2 Challenges

While Custos has its successes, it is certainly not without its challenges. First and foremost, Custos does not eliminate the need for trust, it simply isolates and it and makes it more flexibly assignable. In order to gain many of the usability benefits of a Custos-backed encryption system, you must still trust a third party. While it is possible to operate your own Custos server, and some may very well do that, doing so is not within realistic reach of many end-users, and may very well negate many of the usability benefits Custos provides. Trust is still necessary, Custos only helps commoditize and regulate it.

Furthermore, Custos, like any new system, faces adoption challenges. At this point in time, there are no mainstream systems that utilize Custos. While I believe many systems could be refactored to use Custos with minimal effort, the task of doing so, or convincing others to do so, in still non-trivial. In order for Custos to succeed, it must see at least moderately widespread standardization and adoption. There must be readily available secure applications that utilize Custos. Likewise, Custos providers must be easily available, affordable, and numerous (in order to ensure some modicum of trustworthiness through competition). Until Custos, or a similar standardized

secret storage system, becomes widely available, using Custos in live production settings will remain challenging.

The performance overhead of using Custos is also not well understood. For many end-user applications, raw performance is not the primary concern, and there may exist a willingness to sacrifice some performance in the name of increased security. That said, the overhead of Custos across a range of applications has not been evaluated. Nor have Custos performance bottlenecks and possible improvements been identified. Such analysis will come with time and additional use of Custos and Custos-backed applications.

Custos's authentication system, especially the access control chain component, is highly flexible. But it remains to be seen whether or not this flexibility will lead only to increased ease of use (a good thing), or whether it risks giving the user too much freedom, making it prone to misconfiguration and errors. Furthermore, the Custos protocol is believed to be capable of supporting a wide range of authentication primitives, but at this time, only a handful of authentication primitives have actually been tested. Whether or not the current format is capable of supporting more complex authentication schemes, and how easily they might be implemented within the Custos plugin framework, remains to be seen.

5.2 Future Work

The Custos work presented in this document represents the culmination of the initial Custos design and implementation effort. It has resulted in a usable secret storage service and the basis of a variety of applications that leverage this service. That said, there is plenty of work to be done to make Custos a fully production-ready and proven system.

One of the key tenets of Custos design was usability, both the base usability of Custos itself, and the increased usability of applications leveraging Custos. I would like to conduct one or more user studies evaluating the usability of Custos and Custos backed applications. This might include measuring the success users have building access control chains that meet their intentions (vs those that subvert intentions through misconfiguration). It would also likely include measuring

the usability difference between a traditional encryption system and a Custos-backed encryption system. Backing up the Custos design principles with some solid usability data, and adapting these principles where necessary to increase usability, is a high priority for future Custos research.

I would also like to expand the reference Custos server implementation, making it more robust, scalable, and widely deployable. This will include switching to a high performance key:value back-end, improving the Custos authentication plugin interface, producing plugins for additional authentication primitives, and improving the efficiency of the Custos access control chain verification process. I would also like to explore availability and redundancy features of the Custos server using various secret sharing schemes.

I plan to build out several Custos-backed applications. This may include either new native applications (like a Dropbox [53] encryption plugin) or the modification of existing applications (like eCryptfs [45]). These applications would allow further testing of the Custos architecture and server in a production environment, and might enable some of the use cases studies previously mentioned.

Finally, Custos deserves a formal security audit to fully evaluate the security of the server, client libraries, and communication protocol. If Custos is to interact with secure systems, it must not jeopardize the security of these systems. Fortunately the Custos code base is still small enough that a manual audit is possible. Subjecting Custos to automatic auditing tools or bounty-based exploit contests might also yield interesting results with respect to the security of the underlying systems. Issues and exploits discovered in such an audit would be addresses in the design and implementation of future Custos revisions.

5.3 Discussion

The cryptographer Bruce Schneier once wrote, “It is insufficient to protects ourselves with laws; we need to protect ourselves with mathematics” [105]. He is, of course, referring to strong

cryptography as the great technological equalizer, allowing anyone with access to a computer to achieve "the same security as the largest governments" [106]. As it turns out, this is not true[1] .

Not only has cryptography failed to enable the average person to protect herself [43], the belief that it can has led to an increasing gap between the average user, who desires her data to be protected, but who lacks the ability to protect it, and the elites who are capable of protecting their own data while also preying on those who can't. In his counter-culture manifesto, *Computer Lib*, Ted Nelson states that, "Guardianship of the computer can no longer be left to the priesthood" [82]. It is just as true today as it was when he wrote it. We can not afford to forgo control of our data, leaving it to be picked over by the "priesthood" of crypto elite. We have already seen where that has taken us in the revelations of Mr. Edward Snowden, et. al. [44]: to a world where governments and criminals (and governments turned criminals) will prey on the average computer user who lacks the tools to adequately protect herself.

Furthermore, the rise of data-driven online serves has drastically increased the exposure of our data to corporate interests. Users of sites like Facebook or Twitter like to think they are the customers of these services. They are not. They are the product. The customers are the companies that buy user information or access to user eyeballs from these services. This has created a colossal conflict of interest: the companies we increasingly rely to store and safeguard our data have a vested interest in profiting off of it. We must avoid this conflict be creating a separate class of third parties, dedicated to the protection of our data (likely in return for payment) to act as a counter balance against the companies that desire to profit from our data.

Custos is not about the mathematics. It is about making the mathematics, and the resulting encryption techniques, available to everyone. It is about increasing security through the commoditization of trust, through an increase in usability, and through the flexibility to support a diversity of end user intentions. We all want to protect our data, and Custos aims to provide the basis for a set of tools that will allow us to do that.

[1] Schneier acknowledges the naivety of his original Utopian outlook on cryptography in his more recent works [106].

Security researchers can no longer afford to ignore the big picture. Encryption alone is useless. Security requires a holistic treatment [4]. We must approach security from a technological, legal, and anthropological standpoint. Only when we consider all of these factors can we hope to make systems truly secure. Custos is one attempt to accommodate a wider outlook in order to increase end user security. I hope that other such project will follow and that Custos and related efforts will succeed. We must reclaim security, reclaim encryption, and reclaim control of our data. Our continued digital freedom, and by proxy, our physical freedom, depend on doing just this.

Bibliography

[1] AgileBits. 1password. https://agilebits.com/onepassword.

[2] Amazon. Cloud hsm. http://aws.amazon.com/cloudhsm/.

[3] Amazon. Simple storage service. http://aws.amazon.com/s3/.

[4] Ross Anderson. Why information security is hard - an economic perspective. In Seventeenth Annual Computer Security Applications Conference, pages 358–365. IEEE Comput. Soc, 2001.

[5] The Apache Software Foundation. Apache http server project. http://httpd.apache.org/.

[6] The Apache Software Foundation. Cassandra. http://cassandra.apache.org/.

[7] Apple. icloud. http://www.apple.com/icloud/.

[8] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In Proceedings of the sixth conference on Computer systems, pages 31–46, 2011.

[9] Matt Bishop. Unix security: threats and solutions. In SHARE 86.0, number 916, pages 1–38, 1996.

[10] Matt Blaze. Oblivious key escrow. In Ross Anderson, editor, Information Hiding. Springer, Berlin, 1996.

[11] Jon Brodkin. The secret to online safety: Lies, random characters, and a password manager. Ars Technica, 2013.

[12] Milan Broz. dm-crypt. https://code.google.com/p/cryptsetup/wiki/DMCrypt.

[13] Jose M. Alcaraz Calero, Nigel Edwards, Johannes Kirschnick, Lawrence Wilcock, and Mike Wray. Toward a Multi-Tenancy Authorization System for Cloud Services. IEEE Security & Privacy Magazine, 8(6):48–55, November 2010.

[14] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 1880: OpenPGP Message Format. Technical report, 2007.

[15] Martìn Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4), 2007.

[16] James L. Cebula and Lisa R. Young. A taxonomy of operational cyber security risks. Technical Report December, 2010.

[17] Douglas Crockford. Introducing json. http://www.json.org/.

[18] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. Technical report, 1999.

[19] Dorothy E. Denning and Dennis K. Branstad. A Taxonomy for Key Escrow Encryption Systems. Communications of the ACM, 39(3):34–40, 1996.

[20] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2. Technical report, 2008.

[21] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, 22(6):29–40, 1976.

[22] Yuan Dong, Jinzhan Peng, Dawei Wang, Haiyang Zhu, Sun C. Chan, and Michael P. Mesnier. RFS: a network file system for mobile devices and the cloud. ACM SIGOPS Operating Systems Review, 45(1):101–111, 2011.

[23] P. J. Eby. Pep 3333: Python web server gateway interface. http://www.python.org/dev/peps/pep-3333/.

[24] Bjarni Einarsson, Smair McCarth, and Brennan Novak. mailpile. http://www.mailpile.is.

[25] The Enigmail Project. Enigmail. https://www.enigmail.net.

[26] Donald G. Firesmith. A taxonomy of security-related requirements. International Workshop on High Assurance Systems, 2005.

[27] Stephen Flowerday and Rossouw Von Solms. Trust: An Element of Information Security. In Simone Fischer-Hübner, Kai Rannenberg, Louise Yngström, and Stefan Lindskog, editors, Security and Privacy in Dynamic Environments, volume 201 of IFIP International Federation for Information Processing, pages 87–98. Kluwer Academic Publishers, Boston, 2006.

[28] Python Software Foundation. shelve - python object persistence. http://docs.python.org/2/library/shelve.html.

[29] A. Freier, P. Karlton, and P. Kocher. RFC 6106: The Secure Socket Layer (SSL) Protocol - Version 3.0. Technical report, 2011.

[30] Clemens Fruhwirth. Luks. https://code.google.com/p/cryptsetup/.

[31] S. M. Furnell, A. Jusoh, and D. Katsabas. The challenges of understanding and using security: A survey of end-users. Computers & Security, 25(1):27–35, February 2006.

[32] Steven Furnell. Usability versus complexity striking the balance in end-user security. Network Security, (12):13–17, December 2010.

[33] Steven Furnell, Adila Jusoh, Dimitris Katsabas, and Paul Dowland. Considering the Usability of End-User Security Software. In Simone Fischer-Hübner, Kai Rannenberg, Louise Yngström, and Stefan Lindskog, editors, Security and Privacy in Dynamic Environments, volume 201 of IFIP International Federation for Information Processing, pages 307–316. Kluwer Academic Publishers, Boston, 2006.

[34] Gazzang. ztrustee. http://www.gazzang.com/products/ztrustee.

[35] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: an auditing file system for theft-prone devices. In Proceedings of the sixth conference on Computer systems - EuroSys '11, pages 1–16, New York, New York, USA, 2011. ACM Press.

[36] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. ACM SIGOPS Operating Systems Review, 37(5):29–43, December 2003.

[37] NightLabs Consulting GmbH. Cumulus4j. http://www.cumulus4j.org/.

[38] Dan Goodin. Why passwords have never been weaker, and crackers have never been stronger. Ars Technica, 2012.

[39] Dan Goodin. How the bible and youtube are fueling the next frontier of password cracking. Ars Technica, 2013.

[40] Google. Authenticator. https://code.google.com/p/google-authenticator/.

[41] Google. Chrome. https://www.google.com/chrome/.

[42] Google. Drive. http://www.google.com/drive/about.html.

[43] Matthew Green. The daunting challenge of secure e-mail. The New Yorker, 2013.

[44] Glenn Greenwald and Ewen MacAskill. Nsa prism program taps in to user data of apple, google, and others. The Guardian, 2013.

[45] Michael Halcrow. ecryptfs. http://ecryptfs.org/.

[46] Michael Austin Halcrow. eCryptfs : An Enterprise-class Cryptographic Filesystem for Linux. In Ottawa Linux Symposium, pages 201–218, Ottawa, 2005. International Business Machines, Inc.

[47] Eric Haszlakiewicz. json-c. https://github.com/json-c/json-c/wiki.

[48] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and Alex J. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In Proceedings of the 21st USENIX Security Symposium, 2012.

[49] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51–81, February 1988.

[50] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context sensitive access control. In Proceedings of the tenth ACM symposium on Access control models and technologies, page 111, New York, New York, USA, 2005. ACM Press.

[51] IBM. Restful web services: The basics. http://www.ibm.com/developerworks/webservices/library/ws-restful/.

[52] Tarik Ibrahim, Steven M. Furnell, Marira Papadaki, and Nathan L. Clark. Assessing the Usability of End-User Security Software. Trust, Privacy and Security in Digital Business, 6264:177–189, 2010.

[53] Dropbox Inc. Dropbox. `https://www.dropbox.com/`.

[54] MongoDB Inc. mongodb. `http://www.mongodb.org/`.

[55] Christian D. Jensen. CryptoCache: a secure sharable file cache for roaming users. In Proceedings of the 9th ACM SIGOPS European Workshop, page 73, New York, New York, USA, 2000. ACM Press.

[56] Michael K. Johnson. A tour of the linux vfs. `http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html`, 1996.

[57] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies, pages 29–42, 2003.

[58] Zeus Kerravala. Configuration management delivers business resiliency. The Yankee Group, 2002.

[59] Vishal Kher and Yongdae Kim. Securing distributed storage: challenges, techniques, and systems. In Proceedings of the 2005 ACM workshop on Storage security and survivability, page 9, New York, New York, USA, 2005. ACM Press.

[60] Werner Koch and Marcus Brinkmann. STEED - Usable End-to-End Encryption. Technical report, 2011.

[61] Werner Koch and The GNU Project. Gnupg. `http://www.gnupg.org/`.

[62] John T. Kohl, B. Clifford Neuman, and Theodore Y. Ts'o. The evolution of the Kerberos authentication service. In European Conference Proceedings, 1991.

[63] Hugo Krawczyk. Secret Sharing Made Short. In Douglas R. Stinson, editor, Advances in Cryptology-CRYPTO'93, volume 773 of Lecture Notes in Computer Science, pages 136–146. Springer Berlin Heidelberg, Berlin, Heidelberg, July 1994.

[64] Brian Krebs. Safeguarding your passwords. Krebs on Security, 2008.

[65] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. ACM SIGPLAN Notices, 35(11):190–201, 2000.

[66] Puppet Labs. Puppet. `http://puppetlabs.com/`.

[67] LastPass. Lastpass. `https://lastpass.com/`.

[68] P. Leach, M. Mealling, and R. Salz. RFC 4122: A universally Unique IDentifier (UUID) URN Namespace. Technical report, 2005.

[69] Marcos A. P. Leandro, Tiago J. Nascimento, Daniel R. dos Santos, Carla M. Westphall, and Carlos B. Westphall. Multi-tenancy authorization system with federated identity for cloud-based environments using shibboleth. In The Eleventh International Conference on Networks, pages 88–93, 2012.

[70] Yan Li, Nakul Sanjay Dhotre, Yasuhiro Ohara, Thomas M. Kroeger, Ethan L. Miller, and Darrell D. E. Long. Horus: Fine-Grained Encryption-Based Security for Large-Scale Storage. In Proceedings of the 11th Conference on File and Storage Systems, 2013.

[71] The Linux-PAM Team. Linux pam. `http://www.linux-pam.org/`.

[72] Witold Litwin, Sushil Jajodia, and Thomas Schwarz. Privacy of data outsourced to a cloud for selected readers through client-side encryption. In Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, page 171, New York, New York, USA, 2011. ACM Press.

[73] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. ACM Transactions on Computer Systems, 29(4):1–38, December 2011.

[74] A. Matsui, J. Nakajima, and S. Moriai. RFC 3713: A Description of the Camellia Encryption Algorithm. Technical report, 2004.

[75] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. ACM SIGOPS Operating Systems Review, 33(5):124–139, December 1999.

[76] Michelle L. Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring Password Guessability for an Entire University. Technical report, 2013.

[77] Carnegie Mellon. Captcha. `http://www.captcha.net/`.

[78] Carnegie Mellon. Sasl. `http://asg.web.cmu.edu/sasl/`.

[79] A. Menezes, P. van Oorschot, and S. Vanstone. Overview of Cryptography. In Handbook of Applied Cryptography, pages 1–48. 1996.

[80] MIT Media Lab. OpenPDS Software. Technical report, Human Dynamics, MIT Media Lab, 2012.

[81] Mozilla. Persona. `https://developer.mozilla.org/en-US/Persona`.

[82] Ted Nelson. Computer Lib / Dream machines. Self published, 1974.

[83] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. IEEE Communications Magazine, 32(9):33–38, 1994.

[84] National Institute of Standards & Technology (NIST). Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication, 2001.

[85] National Institute of Standards & Technology (NIST). FIPS 140-2: Security Requirements for Cryptographic Modules. Federal Information Processing Standards Publication, 2001.

[86] Donald A Norman. The design of everyday things. Basic books, 2002.

[87] OASIS. Saml. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

[88] The OAuth Team. Oauth. http://oauth.net/.

[89] The OpenBSD Team. Openssh. http://www.openssh.com/.

[90] The OpenID Foundation. Openid. http://openid.net/.

[91] The OpenPGP Alliance. Openpgp. http://www.openpgp.org/.

[92] The OpenSSL Project. Openssl. http://www.openssl.org/.

[93] Opscode. Chef. http://www.opscode.com/chef/.

[94] Alen Peacock, Xian Ke, and Matthew Wilkerson. Typing patterns: a key to user identification. IEEE Security & Privacy Magazine, 2(5):40–47, September 2004.

[95] Rackspace. Cloud keep. https://github.com/cloudkeep.

[96] Jarret Raim and Matt Tesauro. Cloud keep: Openstack key management as a service. https://www.openstack.org/summit/portland-2013/session-videos/presentation/cloud-keep-openstack-key-management-as-a-service.

[97] Kenneth Reitz. Requests: Http for humans. http://www.python-requests.org/.

[98] Jason K. Resch and James S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In 9th USENIX Conference on File and Storage Technologies, 2011.

[99] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126, 1978.

[100] Armin Ronacher. Flask: web development one drop at a time. http://flask.pocoo.org/.

[101] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. ACM SIGOPS Operating Systems Review, 35(5):188, December 2001.

[102] Saltstack. Salt. http://www.saltstack.com/.

[103] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In Proceedings of the Summer 1985 USENIX Conference, pages 119–130, Portland, OR, 1985.

[104] Andy Sayler, Junho Ahn, and Richard Han. Os programming assignment: An encrypted filesystem. https://github.com/asayler/CU-CS3753-PA5.

[105] Bruce Schneier. Applied Cryptography. John Wiley & Sons, 1996.

[106] Bruce Schneier. Secrets and Lies. John Wiley & Sons, 2000.

[107] Bruce Schneier. Password advice. Schneier on Security, 2009.

[108] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. Twofish: A 128-bit block cipher. Technical report, 1998.

[109] Adi Shamir. How to share a secret. Communications of the ACM, 22(11):612–613, November 1979.

[110] The Shibboleth Consortium. Shibboleth. `http://shibboleth.net/`.

[111] Abe Singer, Warren Anderson, and Rik Farrow. Rethinking Password Policies (Uncut). ;login, 38(4), 2013.

[112] S. W. Smith. Fairy dust, secrets, and the real world. Security & Privacy, IEEE, 1(1):89–93, January 2003.

[113] Dag-Erling Smorgrav. Openpam. `http://www.openpam.org/`.

[114] Daniel Stenberg. curl. `http://curl.haxx.se/`.

[115] Michael Sweikata, Gary Watson, Charles Frank, Chris Christensen, and Yi Hu. The usability of end user cryptographic products. In 2009 Information Security Curriculum Development Conference, page 55, New York, New York, USA, 2009. ACM Press.

[116] Symantic. Pgp. `http://www.symantec.com/encryption`.

[117] Miklos Szeredi. Fuse: Filesystems in userspace. `http://fuse.sourceforge.net/`.

[118] Alma Whitten and J. D. Tygar. Usability of security: A case study. Technical Report 102590, 1998.

[119] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In Proceedings of the 8th USENIX Security Symposium, pages 679–702, 1999.

[120] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: the least-authority filesystem. In Proceedings of the 4th ACM international workshop on Storage security and survivability, pages 21–26, New York, New York, USA, 2008. ACM Press.

[121] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. Communications of the ACM, 17(6):337–345, June 1974.

[122] Yubico. Yubikey standard. `http://www.yubico.com/products/yubikey-hardware/yubikey/`.

Appendix A

Sample Custos Messages

A.1 Create New Key:Value Object

A.1.1 Request

```
[
    {
        "Class": "explicit",
        "Type": "user_id",
        "Value": "YXNheWxlcgA=",
        "Echo": true
    },
    {
        "Class": "explicit",
        "Type": "psk",
        "Value": "TXlPYmplY3RDcmVhdGlvblBhc3N3b3JkA
                  A==",
        "Echo": false
    }
]
```

Listing A.1: Create Request `Attr` Object

```
POST https://custos.net/grp/cc4273ae-4e1e-11e3-90d4-10bf487b3e94/obj?aa=%5B%20%7B
    %20%22Class%22%3A%20%22explicit%22%2C%20%22Type%22%3A%20%22user_id%22%2C%20%22
    Value%22%3A%20%22YXNheWxlcgA%3D%22%2C%20%22Echo%22%3A%20true%20%7D%2C%20%7B%20%22
    Class%22%3A%20%22explicit%22%2C%20%22Type%22%3A%20%22psk%22%2C%20%22Value%22%3A
    %20%22TXlPYmplY3RDcmVhdGlvblBhc3N3b3JkAA%3D%3D%22%2C%20%22Echo%22%3A%20false%20%7
    D%20%5D
```

Listing A.2: Create HTTP Request

```
{
    "Keys": [
        {
            "Value": "VHdhcyBicmlsbGlnLCBhbmQgdGhlIHNsa
                      XRoeSB0b3ZlczsgRGlkIGd5cmUgYW5kIG
                      dpbWJsZSBpbiB0aGUgd2FiZQA=",
            "Echo": true
        }
    ],
    "ACSs": [
        {
            "Permissions":
            {
                "obj_delete": null,
                "obj_read": [
                    [
                        {
                            "Class": "explicit",
                            "Type": "user_id",
                            "Value": "YXNheWxlcgA=",
                            "Echo": true
                        },
                        {
                            "Class": "explicit",
                            "Type": "psk",
                            "Value": "TXlPYmplY3RBY2Nlc3NQYXNzd29yZAA=",
```

```
27                                  "Echo": false
28                              }
29                          ]
30                      ]
31                      "obj_update": null,
32                      "obj_audit": null,
33                      "obj_clean": null,
34                      "obj_acs_get": null,
35                      "obj_acs_set": null
36                  }
37                  "Echo": true
38              }
39          ]
40  }
```

Listing A.3: Create HTTP Request Body

A.1.2 Response


```
{
    "Status": "okay",
    "Keys": [
        {
            "Value": "VHdhcyBicmlsbGlnLCBhbmQgdGhlIHNsa
                      XRoeSB0b3ZlczsgRGlkIGd5cmUgYW5kIG
                      dpbWJsZSBpbiB0aGUgd2FiZQA=",
            "Echo": true,
            "Revision": 0,
            "UUID": "7af8c95d-479a-46fe-b5de-8574c6ca1369",
            "Status": "accepted"
        }
    ],
    "ACSs": [
        {
            "Permissions":
            {
                "obj_delete": null,
                "obj_read": [
                    [
                        {
                            "Class": "explicit",
                            "Type": "user_id",
                            "Value": "YXNheWxlcgA=",
                            "Echo": true
                        },
                        {
                            "Class": "explicit",
                            "Type": "psk",
                            "Value": null,
                            "Echo": false
                        }
                    ]
                ]
                "obj_update": null,
                "obj_audit": null,
                "obj_clean": null,
                "obj_acs_get": null,
                "obj_acs_set": null
            },
            "Echo": true,
            "Status": "accepted"
        }
    ],
    "Attrs": [
        {
            "Class": "explicit",
            "Type": "user_id",
            "Value": "YXNheWxlcgA=",
            "Echo": true,
            "Status": "accepted",
            "ResValue": null
        },
        {
            "Class": "explicit",
```

```
56              "Type": "psk",
57              "Value": null,
58              "Echo": false,
59              "Status": "accepted",
60              "ResValue": null
61          },
62          {
63              "Class": "implicit",
64              "Type": "ip_src",
65              "Value": "NzUuMTQ4LjExOC4yMTcA",
66              "Echo": true,
67              "Status": "ignored",
68              "ResValue": null
69          }
70      ]
71  }
```

Listing A.4: Create HTTP Response

A.2 Get Existing Key:Value Object

A.2.1 Request - Denied

```
[
    {
        "Class": "explicit",
        "Type": "user_id",
        "Value": "YXNheWxlcgA=",
        "Echo": true
    }
]
```

Listing A.5: Get Request `Attr` Object - Incomplete

```
GET https://custos.net/grp/cc4273ae-4e1e-11e3-90d4-10bf487b3e94/obj/7af8c95d-479a-46
    fe-b5de-8574c6ca1369?aa=%5B%20%7B%20%22Class%22%3A%20%22explicit%22%2C%20%22Type
    %22%3A%20%22user_id%22%2C%20%22Value%22%3A%20%22YXNheWxlcgA%3D%22%2C%20%22Echo
    %22%3A%20true%20%7D%20%5D
```

Listing A.6: Get HTTP Request - Incomplete

A.2.2 Response - Denied

```
{
    "Status": "okay",
    "Keys": [
        {
            "Value": null,
            "Echo": null,
            "Revision": null,
            "UUID": "7af8c95d-479a-46fe-b5de-8574c6ca1369",
            "Status": "denied"
        }
    ],
    "Attrs": [
        {
            "Class": "explicit",
            "Type": "user_id",
            "Value": "YXNheWxlcgA=",
            "Echo": true,
            "Status": "accepted",
            "ResValue": null
        },
        {
            "Class": "explicit",
            "Type": "psk",
            "Value": null,
            "Echo": false,
            "Status": "required",
            "ResValue": null
        },
        {
            "Class": "implicit",
            "Type": "ip_src",
            "Value": "NzUuMTQ4LjExOC4yMTcA",
            "Echo": true,
            "Status": "ignored",
            "ResValue": null
        }
    ]
}
```

Listing A.7: Get HTTP Response - Access Denied

A.2.3 Accepted Request

```
[
    {
        "Class": "explicit",
        "Type": "user_id",
        "Value": "YXNheWxlcgA=",
        "Echo": true
    },
    {
        "Class": "explicit",
        "Type": "psk",
        "Value": "TXlPYmplY3RBY2Nlc3NQYXNzd29yZAA=",
        "Echo": false
    }
]
```

Listing A.8: Get Request `Attr` Object - Complete

```
GET https://custos.net/grp/cc4273ae-4e1e-11e3-90d4-10bf487b3e94/obj/7af8c95d-479a-46
    fe-b5de-8574c6ca1369?aa=%5B%20%7B%20%22Class%22%3A%20%22explicit%22%2C%20%22Type
    %22%3A%20%22user_id%22%2C%20%22Value%22%3A%20%22YXNheWxlcgA%3D%22%2C%20%22Echo
    %22%3A%20true%20%7D%2C%20%7B%20%22Class%22%3A%20%22explicit%22%2C%20%22Type%22%3A
    %20%22psk%22%2C%20%22Value%22%3A%20%22TXlPYmplY3RBY2Nlc3NQYXNzd29yZAA%3D%22%2C
    %20%22Echo%22%3A%20false%20%7D%20%5D
```

Listing A.9: Get HTTP Request - Complete

A.2.4 Accepted Response

```
{
    "Status": "okay",
    "Keys": [
    {
        "Value": "VHdhcyBicmlsbGlnLCBhbmQgdGhlIHNsaX
                  RoeSB0b3ZlczsgRGlkIGd5cmUgYW5kIGdp
                  bWJsZSBpbiB0aGUgd2FiZQA=",
        "Echo": null,
        "Revision": 0,
        "UUID": "7af8c95d-479a-46fe-b5de-8574c6ca1369",
        "Status": "accepted"
    }
    ],
    "Attrs": [
    {
        "Class": "explicit",
        "Type": "user_id",
        "Value": "YXNheWxlcgA=",
        "Echo": true,
        "Status": "accepted",
        "ResValue": null
    },
    {
        "Class": "explicit",
        "Type": "psk",
        "Value": null,
        "Echo": false,
        "Status": "accepted",
        "ResValue": null
    },
    {
        "Class": "implicit",
        "Type": "ip_src",
        "Value": "NzUuMTQ4LjExOC4yMTcA",
        "Echo": true,
        "Status": "ignored",
        "ResValue": null
    }
    ]
}
```

Listing A.10: Get HTTP Response - Access Granted

Appendix B

`libcustos` Interface

```c
/* custos.h
 *
 * custos_client interface - C bindings
 *
 * By Andy Sayler (www.andysayler.com)
 * Initially created 05/13
 *
 */

#ifndef CUSTOS_CLIENT_H
#define CUSTOS_CLIENT_H

#include <errno.h>
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <uuid/uuid.h>
#include <json/json.h>

#include "custos_http.h"
#include "custos_json.h"

#define CUS_VERSION "0.2-dev"

#define CUSTOS_ENDPOINT_GRP "/grp"
#define CUSTOS_ENDPOINT_OBJ "/obj"
#define CUSTOS_ENDPOINT_ADT "/audit"
#define CUSTOS_ENDPOINT_ACS "/acs"

#define CUSTOS_QUERY_AA  "aa"
#define CUSTOS_QUERY_CHK "chk"
#define CUSTOS_QUERY_REV "rev"

typedef enum custosOp {
    CUS_OP_OBJ_CREATE = 0,
    CUS_OP_OBJ_DELETE,
    CUS_OP_OBJ_READ,
    CUS_OP_OBJ_UPDATE,
    CUS_OP_OBJ_MAX
```

```
44 } custosOp_t;
45
46 typedef enum custosResStatus {
47     CUS_RESSTAT_OKAY = 0,
48     CUS_RESSTAT_UGROUP,
49     CUS_RESSTAT_UOBJECT,
50     CUS_RESSTAT_ERROR,
51     CUS_RESSTAT_MAX
52 } custosResStatus_t;
53
54 #define CUS_RESSTAT_ACCEPTED_STR "okay"
55 #define CUS_RESSTAT_UGROUP_STR   "unknown_group"
56 #define CUS_RESSTAT_UOBJECT_STR  "unknown_object"
57 #define CUS_RESSTAT_ERROR_STR    "error"
58
59 typedef enum custosAttrStatus {
60     CUS_ATTRSTAT_ACCEPTED = 0,
61     CUS_ATTRSTAT_DENIED,
62     CUS_ATTRSTAT_REQUIRED,
63     CUS_ATTRSTAT_OPTIONAL,
64     CUS_ATTRSTAT_IGNORED,
65     CUS_ATTRSTAT_MAX
66 } custosAttrStatus_t;
67
68 #define CUS_ATTRSTAT_ACCEPTED_STR "accepted"
69 #define CUS_ATTRSTAT_DENIED_STR   "denied"
70 #define CUS_ATTRSTAT_REQUIRED_STR "required"
71 #define CUS_ATTRSTAT_OPTIONAL_STR "optional"
72 #define CUS_ATTRSTAT_IGNORED_STR  "ignored"
73
74 typedef enum custosKeyStatus {
75     CUS_KEYSTAT_ACCEPTED = 0,
76     CUS_KEYSTAT_DENIED,
77     CUS_KEYSTAT_MAX
78 } custosKeyStatus_t;
79
80 #define CUS_KEYSTAT_ACCEPTED_STR "accepted"
81 #define CUS_KEYSTAT_DENIED_STR   "denied"
82
83 typedef enum custosACSStatus_t {
84     CUS_ACSSTAT_ACCEPTED = 0,
85     CUS_ACSSTAT_DENIED,
86     CUS_AVSSTAT_MAX
87 }
88
89 #define CUS_ACSSTAT_ACCEPTED_STR "accepted"
90 #define CUS_ACSSTAT_DENIED_STR   "denied"
91
92 typedef enum custosAttrClass {
93     CUS_ATTRCLASS_IMPLICIT = 0,
94     CUS_ATTRCLASS_EXPLICIT,
95     CUS_ATTRCLASS_MAX
96 } custosAttrClass_t;
97
98 #define CUS_ATTRCLASS_IMPLICIT_STR "implicit"
99 #define CUS_ATTRCLASS_EXPLICIT_STR "explicit"
100
101 typedef enum custosAttrType {
```

```
102     CUS_ATTRTYPE_EXP_USR = 0,
103     CUS_ATTRTYPE_EXP_PSK,
104     CUS_ATTRTYPE_EXP_PSKSHA2,
105     CUS_ATTRTYPE_EXP_PSHBCRPT,
106     CUS_ATTRTYPE_IMP_SOURCEIP = 0,
107     CUS_ATTRTYPE_IMP_USRAGENT,
108     CUS_ATTRTYPE_IMP_AUTHTYPE,
109     CUS_ATTRTYPE_IMP_AUTHVAL,
110     CUS_ATTRTYPE_IMP_TIMEUTC,
111     CUS_ATTRTYPE_IMP_MAX
112 } custosAttrType_t;
113
114 #define CUS_ATTRTYPE_MAX CUS_ATTRTYPE_IMP_MAX
115 #define CUS_ATTRTYPE_EXP_USR_STR        "user_id"
116 #define CUS_ATTRTYPE_EXP_PSK_STR        "psk"
117 #define CUS_ATTRTYPE_EXP_PSKSHA2_STR    "psk_sha256"
118 #define CUS_ATTRTYPE_EXP_PSKBCRPT_STR   "psk_bcrypt"
119 #define CUS_ATTRTYPE_IMP_SOURCEIP_STR   "ip_src"
120 #define CUS_ATTRTYPE_IMP_USRAGENT_STR   "user_agent"
121 #define CUS_ATTRTYPE_IMP_AUTHTYPE_STR   "auth_type"
122 #define CUS_ATTRTYPE_IMP_AUTHVAL_STR    "auth_value"
123 #define CUS_ATTRTYPE_IMP_TIMEUTC_STR    "time_utc"
124
125 typedef enum custosObjPerm_t {
126     CUS_OBJPERM_DELETE = 0,
127     CUS_OBJPERM_READ,
128     CUS_OBJPERM_UPDATE,
129     CUS_OBJPERM_AUDIT,
130     CUS_OBJPERM_CLEAN,
131     CUS_OBJPERM_ACSGET,
132     CUS_OBJPERM_ACSSET,
133     CUS_OBJPERM_MAX
134 }
135
136 #define CUS_OBJPERM_DELETE "obj_delete"
137 #define CUS_OBJPERM_READ   "obj_read"
138 #define CUS_OBJPERM_UPDATE "obj_update"
139 #define CUS_OBJPERM_AUDIT  "obj_audit"
140 #define CUS_OBJPERM_CLEAN  "obj_clean"
141 #define CUS_OBJPERM_ACSGET "obj_acs_get"
142 #define CUS_OBJPERM_ACSSET "obj_acs_set"
143
144 typedef struct custosAttr {
145     custosAttrClass_t  class;
146     custosAttrType_t   type;
147     size_t             size;
148     uint8_t*           val;
149 } custosAttr_t;
150
151 typedef struct custosAttrReq {
152     bool               echo;
153     custosAttr_t*      attr;
154 } custosAttrReq_t;
155
156 typedef struct custosAttrRes {
157     custosAttrStatus_t status;
158     bool               echo;
159     size_t             res_size;
```

```
160     uint8_t*            res_val;
161     custosAttr_t*       attr;
162 } custosAttrRes_t;
163 
164 typedef struct custosKey {
165     uuid_t    uuid;
166     uint64_t  revision;
167     size_t    size;
168     uint8_t* val;
169 } custosKey_t;
170 
171 typedef struct custosKeyReq {
172     bool               echo;
173     custosKey_t*       key;
174 } custosKeyReq_t;
175 
176 typedef struct custosKeyRes {
177     custosKeyStatus_t status;
178     bool               echo;
179     custosKey_t*       key;
180 } custosKeyRes_t;
181 
182 typedef struct custosObjACS {
183     size_t         perm_cnts[CUS_OBJPERM_MAX];
184     custosAttr_t* perm_vals[CUS_OBJPERM_MAX];
185 } custsObjACS_t;
186 
187 typedef struct custosObjACSReq {
188     bool             echo;
189     custosObjACS_t* acs;
190 } custoObjACSReq_t;
191 
192 typedef struct custosObjACSRes {
193     custosACSStatus_t status;
194     bool               echo;
195     custosObjACS_t*    acs;
196 } custoObjACSRes_t;
197 
198 typedef struct custosReq {
199     char*               target;
200     size_t              num_attrs;
201     custosAttrReq_t*    attrs;
202     size_t              num_keys;
203     custosKeyReq_t*     keys;
204     size_t              num_acss;
205     custosObjACSReq_t* acss;
206 } custosReq_t;
207 
208 typedef struct custosRes {
209     custosResStatus_t  status;
210     size_t              num_attrs;
211     custosAttrRes_t*    attrs;
212     size_t              num_keys;
213     custosKeyRes_t*     keys;
214     size_t              num_acss;
215     custosObjACSRes_t* acss;
216 } custosRes_t;
217 
```

```
218
219 /* custosAttr Functions */
220 extern custosAttr_t* custos_createAttr(const custosAttrClass_t class,
221                                        const custosAttrType_t type,
222                                        const size_t size, const uint8_t* val);
223 extern int custos_destroyAttr(custosAttr_t** attrp);
224 extern custosAttr_t* custos_duplicateAttr(const custosAttr_t* attr, bool echo);
225 extern int custos_updateAttr(custosAttr_t* attr,
226                              const custosAttrClass_t class,
227                              const custosAttrType_t type,
228                              const size_t size, const uint8_t* val);
229
230 extern const char* custos_AttrClassToStr(const custosAttrClass_t class);
231 extern custosAttrClass_t custos_StrToAttrClass(const char* str);
232 extern const char* custos_AttrTypeToStr(const custosAttrClass_t class,
233                                         const custosAttrType_t type);
234 extern custosAttrType_t custos_StrToAttrType(const custosAttrClass_t class,
235                                              const char* str);
236
237 /* custosKey Functions */
238 extern custosKey_t* custos_createKey(const uuid_t uuid,
239                                      const uint64_t revision,
240                                      const size_t size, const uint8_t* val);
241 extern int custos_destroyKey(custosKey_t** keyp);
242 extern custosKey_t* custos_duplicateKey(const custosKey_t* key, bool echo);
243 extern int custos_updateKey(custosKey_t* key,
244                             const uuid_t uuid,
245                             const uint64_t revision,
246                             const size_t size, const uint8_t* val);
247
248 /* custosObjACS Functions */
249 extern custosObjACS_t* custos_createObjACS();
250 extern int custos_destroyObjACS(custosObjACS_t** acsp);
251 extern custosObjACS_t* custos_duplicateObjACS(const custosObjACS_t* acs, bool echo);
252 extern int custos_updateObjACSAppendAttr(custosObjACS_t* acs,
253                                          const custosObjPerm_t perm,
254                                          custosAttr_t* attr);
255
256 extern const char* custos_ObjPermToStr(const custosObjPerm_t perm);
257 extern custosObjPerm_t custos_StrToObjPerm(const char* str);
258
259 /* custosAttrReq Functions */
260 extern custosAttrReq_t* custos_createAttrReq(const bool echo);
261 extern int custos_destroyAttrReq(custosAttrReq_t** attrreqp);
262 extern int custos_updateAttrReqAddAttr(custosAttrReq_t* attrreq,
263                                        custosAttr_t* attr);
264
265 /* custosKeyReq Functions */
266 extern custosKeyReq_t* custos_createKeyReq(const bool echo);
267 extern int custos_destroyKeyReq(custosKeyReq_t** keyreqp);
268 extern int custos_updateKeyReqAddKey(custosKeyReq_t* keyreq,
269                                      custosKey_t* key);
270
271 /* custosObjACSReq Functions */
272 extern custosObjACSReq_t* custos_createObjACSReq(const bool echo);
273 extern int custos_destroyObjACSReq(custosObjACSReq_t** acsreqp);
274 extern int custos_updateObjACSReqAddACS(custosObjACSReq_t* acsreq,
275                                         custosObjACS_t* acs);
```

```
276
277 /* custosAttrRes Functions */
278 extern custosAttrRes_t* custos_createAttrRes(const custosAttrStatus_t status,
279                                               const bool echo,
280                                               const size_t res_size,
281                                               const uint8_t* res_value);
282 extern int custos_destroyAttrRes(custosAttrRes_t** attrresp);
283 extern int custos_updateAttrResAddAttr(custosAttrRes_t* attrres,
284                                        custosAttr_t* attr);
285
286 extern const char* custos_AttrStatusToStr(const custosAttrStatus_t status);
287 extern custosAttrStatus_t custos_StrToAttrStatus(const char* str);
288
289 /* custosKeyRes Functions */
290 extern custosKeyRes_t* custos_createKeyRes(const custosKeyStatus_t status,
291                                            const bool echo);
292 extern int custos_destroyKeyRes(custosKeyRes_t** keyresp);
293 extern int custos_updateKeyResAddKey(custosKeyRes_t* keyres,
294                                      custosKey_t* key);
295
296 extern const char* custos_KeyStatusToStr(const custosKeyStatus_t status);
297 extern custosKeyStatus_t custos_StrToKeyStatus(const char* str);
298
299 /* custosObjACSRes Functions */
300 extern custosKeyRes_t* custos_createObjACSRes(const custosACSStatus_t status,
301                                               const bool echo);
302 extern int custos_destroyObjACSRes(custosObjACSRes_t** acsresp);
303 extern int custos_updateObjACSResAddACS(custosObjACSRes_t* acsres,
304                                         custosObjACS_t* acs);
305
306 extern const char* custos_ACSStatusToStr(const custosACSStatus_t status);
307 extern custosKeyStatus_t custos_StrToACSStatus(const char* str);
308
309 /* custosReq Functions */
310 extern custosReq_t* custos_createReq(const char* target);
311 extern int custos_destroyReq(custosReq_t** reqp);
312 extern int custos_updateReqAddAttrReq(custosReq_t* req,
313                                       custosAttrReq_t* attrreq);
314 extern int custos_updateReqAddKeyReq(custosReq_t* req,
315                                      custosKeyReq_t* keyreq);
316 extern int custos_updateReqAddObjACSReq(custosReq_t* req,
317                                         custosObjACSReq_t* acsreq);
318
319 /* custosRes Functions */
320 extern custosRes_t* custos_getRes(const custosReq_t* req,
321                                   const custosOp_t op,
322                                   const uuid_t group);
323 extern int custos_destroyRes(custosRes_t** resp);
324
325 extern const char* custos_ResStatusToStr(const custosResStatus_t status);
326 extern custosResStatus_t custos_StrToResStatus(const char* str);
327
328 /* JSON */
329 extern json_object* custos_AttrToJson(const custosAttr_t* attr);
330 extern json_object* custos_KeyToJson(const custosKey_t* key);
331 extern json_object* custos_ObjACSToJson(const custosObjACS_t* acs);
332
333 extern json_object* custos_AttrReqToJson(const custosAttrReq_t* attrreq);
```

```
334 extern json_object* custos_KeyReqToJson(const custosKeyReq_t* keyreq);
335 extern json_object* custos_ObjACSReqToJson(const custosObjACSReq_t* acsreq);
336
337 extern json_object* custos_ReqToJson(const custosReq_t* req);
338
339 extern custosAttr_t* custos_JsonToAttr(json_object* attrjson);
340 extern custosKey_t* custos_JsonToKey(json_object* keyjson);
341 extern custosObjACS_t* custos_JsonToObjACS(json_object* acsjson);
342
343 extern custosAttrRes_t* custos_JsonToAttrRes(json_object* attrresjson);
344 extern custosKeyRes_t* custos_JsonToKeyRes(json_object* keyresjson);
345 extern custosObjACSRes_t* custos_JsonToObjACSRes(json_object* acsresjson);
346
347 extern custosRes_t* custos_JsonToRes(json_object* resjson);
348
349 #endif
```

Listing B.1: `libcustos` Client Header File