Intra-Cloud and Inter-Cloud Authentication

Kevin Walsh
College of the Holy Cross

John Manferdelli
Google, Inc.

***Abstract*—Authentication mechanisms available in existing cloud platforms are inadequate for and poorly-suited to modern cloud-based systems. To support this argument, we describe a variety of authentication mechanisms commonly used in the cloud, and we detail how these mechanisms are actually used in one significant open-source service deployed to a popular cloud platform. We further report on authentication mechanisms used and supported by several of the largest cloud platform providers. The evidence shows that authentication mechanisms currently in use are too coarse-grained, require manual configuration and intervention, and are not systematically applied across components and services. We identify opportunities for cloud platforms to support more secure, fine-grained, automated, and systematic authentication within and between cloud-hosted components.**

I. Introduction

Building a system as a collection of small, mutually-suspicious components is a well established approach to scalability, performance, and security. In principle, small components are more easily analyzed and verified, and mutual suspicion is intended to produce a more robust system as a compromise of one component affects others only to the degree that components trust each other. In the cloud, these ideas underlie trends towards micro-services and containers, for example, in which systems are built as collections of many semi-independent but interconnected components [1], [2]. And with Software-as-a-Service, components are not only architecturally isolated from each other, they also execute under different administrative domains.

Lack of full trust is—or should be—common. For example, many VM images available on Amazon EC2 have hidden vulnerabilities [3], so it would not be prudent to fully trust these when deploying them inside a cloud domain. Across administrative domains, the risk cascading failures between trusting systems is also recognized [4].

When components do not fully trust each other, authentication, i.e. the reliable determination of the source of a message or the endpoints of a communication channel, becomes a critical feature. Authentication is a prerequisite for reliable logging and debugging, for example, and it provides a basis for making authorization decisions. Yet, in many systems that are deployed in the cloud, authentication is severely lacking. Authentication of end-users to cloud services has been studied extensively (e.g., [5], [6]), but authentication between components has received comparatively little attention. In practice, when components use authentication at all, they often rely on ad-hoc mechanisms, or they use a variety of mutually-incompatible authentication mechanisms where a single mechanism would suffice. With few exceptions, these mechanisms are not designed for the cloud.

Consider the common case of using a combination of HTTPS and secret *API keys* [7] to authenticate messages between processes in the cloud:

– Copies of the API key are often kept in configuration files accessible to many (or all) components within a domain.
– Administrators have access to, and must be trusted to safeguard API keys, placing those administrators squarely inside the trusted computing base (TCB) and, consequently, making system security harder to reason about.
– Ad-hoc and insecure channels are used to distribute API keys, all of which must be trusted.

Such obstacles lead to actual attacks. Developers have inadvertently included secret API keys in public repositories [8], for example. Recently, the CEO of one system misused credentials to covertly and transparently alter production databases [9]. Similarly, when HTTPS is used for authentication between components, internal operations are unnecessarily vulnerable to a weak public key infrastructure (PKI) [10], [11], e.g., as evidenced by the recent DigiNotar attack [12].

This paper explores authentication mechanisms available and actually in use in the cloud, and draws from this data observations and suggestions for cloud authentication that is both more secure and easier to use. We argue that authentication mechanisms for the cloud should take as an initial premise that the principals being authenticated are *code*, not *people*, and the differences between these should be explicitly accounted for. Ultimately, we argue for the use of *attestation* of software as a basic building block for cloud authentication, and for integration with underlying cloud platforms to leverage pre-existing trust relationships.

In Sections II and III we detail authentication mechanisms found in the cloud generally and those used by Reddit, a large and popular web site deployed on Amazon EC2. In Section IV we explore the mechanisms used and available in major cloud platforms, both those for authenticating to native services within the clouds and those useful for authentication between cloud-hosted components. We conclude in Section V with a discussion of opportunities for improvement.

II. Authentication in the Cloud

It has become common for a system, e.g., a social-media web site, to be built by interconnecting numerous semi-independent *components*, such as databases, logging servers, load balancers, caches, object managers, and payment gateways. Components may be small, e.g., a specific process

on one VM, or they may be services that encompass many machines. While some components may be designed in-house, others are adopted from third parties but deployed within the same cloud domain. Yet other components are operated by the cloud provider or by third parties, within the same cloud or elsewhere. Such a system necessarily spans multiple administrative and security domains and will make use of a variety of authentication strategies.

We consider three basic scenarios:

– *hosted authentication*, when a cloud-hosted component communicates with a service provided by the underlying cloud infrastructure;
– *intra-cloud authentication*, when two cloud-hosted components communicate and run on the same cloud platform, either in the same or separate domains; and
– *inter-cloud authentication*, when components run on different cloud platforms and in separate administrative domains.

These differ primarily in the extent to which the principals involved have pre-existing trust relationships, and this in turn influences the kinds of authentication mechanisms used.

With hosted authentication, the cloud-hosted component necessarily trusts the underlying cloud platform, and the cloud platform, acting as one endpoint, has unique and privileged insight into the hosted component's state and identity. This could be leveraged by an authentication mechanism, much like the case of authentication between an operating system and its processes, and might generally be termed *second-party authentication*.

API keys are ubiquitous for authenticating components acting as clients in this scenario. OAuth [13] is one framework, though we found many others, both standardized and special-purpose. HTTPS is commonly used for hosted authentication as well, but reliance on HTTPS has significant drawbacks. Notably, the server must store and maintain the secrecy of a potentially long-lived private key. Clients may also need to trust the public HTTPS PKI, a system widely recognized as not particularly trustworthy. Alternatively, a system might forgo the HTTPS PKI entirely and use custom, ad-hoc means for distributing certificates, but such schemes are fragile and often untested. Although HTTPS directly supports mutual authentication by having servers verify client certificates or using HTTP digest authentication [14], we found no examples in which such approaches are used in Reddit. Instead, client components using HTTPS are authenticated using API keys, if at all.

For intra-cloud authentication, the two communicating principals do not have a direct, pre-established trust relationship, but both trust a common third party, namely the shared underlying cloud platform. Thus the cloud platform is ideally placed to assist in authentication. However, as detailed in Section IV, cloud providers vary in their level of support for acting as such a third party. When both components run in a single cloud domain, a domain administrator may similarly be trusted by both components, so the domain administrator can create a third party authentication server where cloud provider support is lacking. This scenario might generally be termed *special third-party authentication*. In addition to the mechanisms used for hosted authentication, we find some components perform mutual authentication in this scenario by using SSH or plain TLS, validating each other's public key certificates without the use of a PKI. Certificate distribution here is usually out-of-band. As with HTTPS, such components must store and maintain long-lived private keys.

For inter-cloud authentication, the components have neither direct nor indirect trust relationships with any common, neutral third parties. Nevertheless, the two cloud providers underlying the two components could serve as neutral third-parties in these scenarios as well, as we discuss in Section V. More generally, components may rely on a variety third parties, such as certificate authorities, constructed specifically to aid in authentication. Inter-cloud authentication is thus an example of *general third-party authentication*. Curiously, authentication mechanisms used by Reddit in this scenario include all those used for intra-cloud authentication, indicating that the unique trust relationships present in intra-cloud scenarios are not actually being leveraged for authentication in practice.

III. Authentication in Reddit's Cloud

To explore the extent to which modern cloud-based services use (or misuse) various authentication strategies, we audited the source code[1] of Reddit. We chose this particular service because the code is open-source and its popularity—reddit.com is ranked[2] among the top 25 sites worldwide—implies the code may be somewhat representative.

A. Reddit Threat Model

Reddit faces numerous threats, but we are specifically interested in threats due to the component-based nature of the system. This includes insiders within Reddit with low-level access to interfere with the operation of one or more components, malicious components within Reddit's own code base (e.g., a compromised front-end web server), and malicious external components (e.g., a compromised third-party storage service or external certificate authority). We assume internal attackers have full access to monitor or modify traffic on the local network within Reddit's cloud domain. We exclude the cloud platform itself or insiders within that infrastructure (e.g., an Amazon employee with access to the EC2 nodes on which Reddit components execute), because addressing such attacks typically requires hardware-level support.

Authentication can provide a foundation for defending against such threats, by allowing the cloud-based components that comprise the system to distinguish between legitimate messages sent by trusted components and malicious messages sent by untrusted principals.

B. Reddit Components and Communications Channels

Reddit includes $71,881$ lines of server-side code, primarily in Python, and 1600 lines of build and deployment scripts. We excluded $90,371$ lines of client-side Javascript, HTML,

[1] http://github.com/reddit/reddit/, version `f56e810a4882`.
[2] Alexa Internet Inc. rankings as of July 29, 2016.

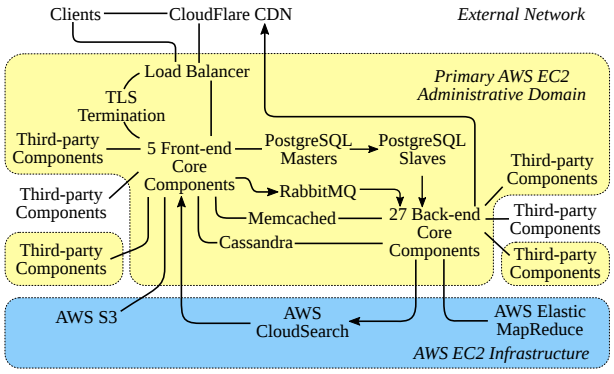

Fig. 1. Reddit architecture. Boxes indicate components executing on top of, or as part of, the AWS EC2 cloud infrastructure. Arrows indicate primarily read-only channels.

and CSS code from our analysis. Figure 1 shows an overview of the system and illustrates interconnections between the most significant components we identified.

We took each process executing Reddit-specific code to be a distinct component. Using this definition, the core comprises a TLS termination server, five additional front-end components in the critical path for requests from public HTTP and HTTPS interfaces, and twenty-seven back-end components executing asynchronously. Front-end components include a central app server, websockets server, media server, user tracker, and ad-click tracker. Back-end components are devoted to pre-rendering pages, generating site-wide statistics, scanning system logs, and other maintenance tasks. All core components share a significant amount of code. Depending on the deployment configuration, these processes can be co-located on a few cloud VM instances or spread across many isolated VM instances. All are deployed within Amazon's AWS EC2 cloud infrastructure in a single administrative domain.

We identified non-core components primarily according to provenance. In some cases we identified smaller components within a single third-party system when boundaries were clearly identifiable, particularly for systems that span multiple VM instances. We identified twenty four types of non-core components in all, including databases, a content distribution network, caches, load balancers, search engines, and message queues. Several of these are deployed within the same EC2 administrative domain as core components, sometimes even co-located on the same VMs, while others are hosted in other EC2 domains or outside of EC2. With few exceptions, all coordination is done through core components—non-core components generally do not communicate directly.

C. Reddit Authentication Mechanisms

End-users authenticate to Reddit using names and passwords, and end-users in turn authenticate Reddit using the HTTPS PKI. How (or whether) the cloud-based components of Reddit authenticate each other is less clear. With such a variety of components, it is not surprising that many different authentication mechanisms can be found in Reddit's code base. Table I summarizes some of these.

1) Implicit Authentication with Ports and Addresses: Many ostensibly independent components don't explicitly perform any authentication at all, instead relying implicitly on OS and local network routing and firewall policies and, in some cases, the correctness of DNS and routing in the wider Internet.

Several core components use CloudSearch, an indexing service provided by AWS. Because a deprecated API is used, only unencrypted HTTP channels are used, and CloudSearch authenticates the core components only by IP address. This means that principals smaller than a single VM instance can't be distinguished. Core components do not explicitly authenticate CloudSearch, relying instead on DNS and local network routing.

Ten Memcached [15] servers are deployed on VMs within the same cloud domain as core components, along with replicas and Memcached routers [16] co-located on core component VMs. All communication with core components and between the various remote replicas is unauthenticated. Memcached does support authentication of clients using SASL [17], but this isn't enabled. Core components similarly use two logging-related components, StatsD [18] and Graphite [19], and an IP-geolocation service without any explicit authentication. Core components send UDP packets to remote StatsD servers, and StatsD instances in turn connect to Graphite using unauthenticated TCP channels. Core back-end components access the IP-geolocation service over plain HTTP. All of these components are hosted on separate VMs, but within the same cloud domain as core components. For all, authentication implicitly rests on the security of the cloud local network and, because high port numbers are used, likely on all software running on those VMs as well.

Cassandra [20], a distributed data store, is configured to run replicas on several VMs, with core components acting as clients. Although Cassandra supports names and passwords for authenticating clients, and mutual TLS with manually distributed keys and certificates for both client-replica and replica-replica communication, none of these mechanisms are used. Instead, Cassandra is configured to use plain TCP with high port numbers, again leaving these channels vulnerable unless the local network and all connected VMs are trusted.

Messages between core components and SMTP daemons are unauthenticated in Reddit's default configuration, though SMTP does support a pluggable architecture [21] for authenticating clients. Because SMTP instances are configured to run on the same VMs as core components, using low port numbers and a loopback-only network, OS-level isolation will at least mitigate some risks from untrusted remote hosts. Even so, all privileged users and software on the VM instance must be trusted.

2) HTTPS Authentication and API Keys: The HTTPS PKI is perhaps the most extensively used authentication mechanism found within the Reddit code base. In all cases where core components access other components through REST-oriented APIs, HTTPS is used for authentication either exclusively or in combination with other mechanisms. For example, some core components interact with the CloudFlare CDN by making

Component	Description	Endpoint / Protocol	Authentication Mechanisms
AWS CloudSearch	text search service	TCP :80 / HTTP	DNS, IP, *AWS IAM Roles*, *API Keys*
AWS S3 & EMR	storage & compute	TCP :443 / HTTPS	PKI, AWS IAM Roles, API Keys
Cassandra	NoSQL database	TCP :9160 / binary	none, *M-TLS*, *password*
CloudFlare	content distribution	TCP :80 / HTTP	shared secret
GeoIP	IP-geolocation service	TCP :5000 / HTTP	none
Google Analytics	metrics tracking	TCP :80, :443 / HTTP, HTTPS	PKI, Public ID
Graphite	metrics analysis	TCP :2003 / plain text	none
Mcrouter	Memcached router	TCP localhost:5050 / plain text	none
Memcached	small object cache	TCP :11211 / plain text	none, *SASL*
PostgreSQL	SQL database	TCP :5432 / binary	password, *M-TLS*, *Kerberos*
RabbitMQ	message queues	TCP localhost:5672 / plain text	password, *M-TLS*, *SASL*, *Cookies*
Reddit Core	various	TCP :80 / HTTP	HMAC
REST APIs	various	TCP :443 / HTTPS	PKI, API Keys
SMTP	mail transport server	TCP localhost:25 / plain text	none, *SMTP AUTH*
StatsD	metrics aggregation	UDP :8125 / plain text	none
Stripe API	payment gateway	TCP :443 / HTTPS	PKI, API Keys, Public ID
Webhooks	various	TCP :80, :443 / HTTP, HTTPS	PKI, Bearer Tokens, *HTTP basic auth*, *RSA*, *IP*
ZooKeeper admin	distributed coordination	TCP localhost:2181 / plain text	password
ZooKeeper replicas	distributed coordination	TCP :23888 / binary	none, *various*

TABLE I. Authentication between core and selected components. Italics denotes available, but unused, authentication mechanisms.

requests over HTTPS, so these back-end components authenticate the remote CloudFlare CDN component by checking a certificate during connection setup. Similarly, HTTPS is used for authenticating components that provide APIs for newsletter subscriptions, bug support, DMCA notices, image and video metadata, and payments. Mutual authentication is achieved for all of these services using API keys to authenticate the core.

Communication between CloudFlare CDN and some core components relies on an unusual variation of API keys, used when proxying HTTP requests from end-users. The proxying is mostly transparent and mostly unauthenticated. However, CloudFlare CDN injects a custom header into each HTTP request to indicate the end-user IP address, along with an HMAC tag for the injected headers based on a secret shared with the core components. This allows the front-end servers to authenticate CloudFlare as the source of this specific HTTP header. Note that the HMAC key is generated by the sender, rather than the recipient as would be more typical for API keys. Also, unlike CloudFlare's REST API, authentication here is not mutual and covers only part of each request, leaving the channel vulnerable to splicing attacks.

3) Bearer Tokens: Webhooks [22] provide another an example of using shared secrets, but with core components acting as servers and other components acting as clients. Webhooks are used to notify core components of events monitored by external, third-party components. For each type of event, a system administrator registers a URL to receive a webhook callback. In Reddit, these URLs resolve to one of the core front-end components. During registration, an administrator generates a secret which third-party components include with each subsequent notification. Core front-end components verify the presence of the secret upon receiving each notification. These secrets act as bearer tokens to authenticate the external components as the source of the notifications. HTTPS URLs are used, so the external components can authenticate the core components, and to protect the secrecy of the bearer tokens. This mechanism is used with four separate payment processor gateways. Some of these gateways support alternative authentication mechanism as well. Stripe, for example, supports HTTP basic authentication for notifications, and CoinBase can sign notifications using an RSA key, the public key for which is distributed manually out-of-band. Both services suggest that core components should also verify the IP address of the external service as a further means of authentication, though this is not done in Reddit.

Core components use bearer tokens to authenticate webhooks from Mailgun, a third-party email-tracking component. Unexpectedly, the same underlying shared secret is also used as an API key to authenticate requests from core components to Mailgun's REST API. Using a common key for independent protocols can be a source of vulnerabilities [23].

4) HMAC Authentication: The Reddit code base includes numerous authentication mechanisms using HMAC or similar keyed-hash message authentication codes. One example involves URLs generated by the front-end app server which, when clicked by a user, result in HTTP requests to the ad-click tracking service. Certain of these URLs include a tag generated using HMAC-SHA256 to authenticate the request as having originated at the front-end app server. A similar scheme using HMAC-SHA1 is used for a tracking pixel that links to a user-tracking server. Neither the ad-click tracker nor user-tracker are external services. Both execute on VM instances within the same cloud infrastructure as the front-end app server, making it simple for the app server and tracking components to share HMAC keys.

There are several scenarios where the app server must, essentially, authenticate requests sent to itself. When generating HTTP cookies for some accounts, the app server includes an HMAC-SHA256 tag. For other cookies, a non-standard keyed-hash scheme is used, taking a SHA1 hash over a sequence number, timestamp, user password, and a secret key, though many such non-standard schemes are known

to be cryptographically weak [24]. In yet other cases, the app server includes HMAC tags or keyed-hashes in shareable URLs that act as capabilities which allow users to access various protected resources hosted by the same app server, such as subscription-management settings or images that are normally private to specific user accounts. Authentication here relies on either HMAC-SHA1 (two cases), HMAC-SHA256 (three cases), non-standard SHA1-based schemes (two cases) or, in one case, a non-standard MD5-based scheme.

5) Public IDs: API keys are normally kept secret from end-users and others. But in some cases, we found so-called API keys that are treated more as public identifiers than as secrets. For example, Reddit uses three separate Google Analytics tracking IDs, each serving as a public identifier to authenticate the entire system—i.e. all of the server-side components, together with all end-users—when reporting page view metrics. Similarly, the Stripe payment gateway has a public identifier in addition to a private API key. While the secret API key authenticates core components to the payment gateway when performing security-critical requests, like accessing or modifying customer payment data, the public identifier is used to authenticate the entire system, including end-users, but only for requests deemed less sensitive.

6) AWS Native Authentication: Reddit makes only modest direct use of AWS cloud services beyond EC2 VM hosting, but three AWS services are accessed by core components: Simple Storage Service (S3), CloudSearch, and the Elastic MapReduce (EMR) data processing service. For CloudSearch, a legacy form of AWS authentication is used, as discussed previously in Section III-C1. For S3 and EMR, mutual authentication is supported by using HTTPS to authenticate the AWS services, and using native AWS authentication mechanisms to authenticate the core components. Depending on the deployment configuration, the later involves either AWS IAM Roles, which we discuss in Section IV-A1, or API keys. In both cases, however, the underlying mechanism relies on secret API keys shared by core components and the AWS services.

7) Names and Passwords: PostgreSQL databases and RabbitMQ message queues are critical to the functionality and security of the entire Reddit service, since nearly all data is stored in or passes through these components. Even so, at best, authentication here is based on only names and passwords.

Core components authenticate to eight independent PostgreSQL databases using a common name and password. Core components do not explicitly authenticate the databases in any way, and all communication is over plain TCP connections. For databases configured with a hot standby, communication between master and standby servers again relies on a common name and password, sent in the clear and without mutual authentication. This leaves credentials vulnerable to even passive attackers on the local network, and the use unprivileged port numbers bypasses even the meager isolation provided by operating systems.

Authentication for RabbitMQ in Reddit's default configuration is similar, with names and passwords to authenticate core components, no mutual authentication, and communication over plain-text TCP connections and high port numbers.

Both PostgreSQL and RabbitMQ support more sophisticated authentication mechanisms. PostgreSQL supports Kerberos [25] for authenticating clients, and it supports mutual authentication between clients, servers, and hot standby servers via TLS with manually distributed keys and certificates, for example. RabbitMQ can use TLS mutual authentication with manually distributed keys and certificates, both for communication between clients and servers and between servers within a cluster. Authentication between RabbitMQ replicas can also be configured to use SASL or cookie-based authentication.

D. Reddit Configuration and Deployment

Many components must be configured with secrets, such as passwords, API keys, bearer tokens, and private TLS keys. Based on the deployment scripts included in the Reddit distribution, these secrets are provisioned as follows. First, an administrator writes all secrets into a single configuration file, along with various other less sensitive configuration parameters. Next, the administrator pushes the configuration file to a ZooKeeper [26] replica. Finally, during startup, core components contact a local ZooKeeper replica to obtain all necessary secrets before establishing channels with other components. These steps are among the most security-sensitive ones an administrator will take, where a single mistyped command or a single well-targeted attack could compromise the security of all components, Surprisingly, there appears to be relatively little attention to security here.

When publishing secrets to ZooKeeper, the administrator does not explicitly authenticate the local ZooKeeper replica. Zookeeper authenticates the administrator with a password, but the password is sent in the clear over an unauthenticated TCP socket. In the default configuration, a ZooKeeper daemon runs on the local host, so the password does not traverse the network, but even so, the daemon uses a high port number that is accessible to any user on the local machine. In effect, Reddit's default configuration trusts all user accounts on the local machine, since they could easily intercept the ZooKeeper password and all other secrets. Furthermore, while ZooKeeper supports a variety of authentication mechanisms for communication between replicas, none is enabled. This means ZooKeeper replicas, while handling the most security sensitive data in the system, communicate over unencrypted, unauthenticated TCP channels with high port numbers. This renders moot all intra-cloud authentication performed by Reddit components, and it makes all other authentication more coarse-grained than would otherwise be possible.

IV. Native Authentication in Cloud Platforms

Given the lack of strong authentication seen within Reddit—and there is little reason to suspect that many other significant cloud-deployed systems have stronger authentication—it is natural to ask whether cloud providers can play a role in providing strong, comprehensive authentication mechanisms for these systems. Ideally, such a mechanism would allow for *direct authentication* of software components, without

requiring developers, administrators, or the components themselves to manage API keys, bearer tokens, password, or other difficult-to-contain secrets. Direct authentication associates a unique identifier with each component, which may involve taking a hash of all the software and configuration underlying a component, or it may rely on other means for choosing unique identifiers for processes, VMs, and tenants. Any mechanism must accommodate not just hosted authentication, but intra-cloud and inter-cloud authentication as well, and it should support components both larger and smaller than a single VM.

A. Amazon Web Services

AWS services support authentication using HTTPS, API keys, network-level names and, for user-facing services, TLS or SSH with manually-distributed public keys. Although network-level names are easily spoofed on the wider internet, EC2 VM instances are prevented from spoofing IP and MAC addresses [27], and firewalls can be configured to filter incoming traffic with spoofed addresses. This mitigates some risk of using network-level names for authentication within the confines of a trusted cloud local network.

1) AWS IAM Roles: Many native AWS resources support authentication using AWS Identity and Access Management (IAM) roles, opaque names associated with an EC2 VM instance during deployment and included on access control lists maintained by AWS services.

Superficially, AWS IAM roles appear to have many properties of the ideal mechanism outlined above. For example, components need not explicitly maintain secret keys, making them immune to the types of secret-management challenges seen in Reddit. Similarly, developers can be removed from the TCB by granting them sufficient privileges to deploy specific VM instances in specific roles, without giving developers privileges to access protected services directly. Associating a role with a VM instance is a privileged and audited operation, so it is feasible to identify all VM instances, hence all software, that can access critical services.

Unfortunately, the full benefits of IAM roles are not realizable in practice because roles are implemented as a series of rotating API keys created and distributed by the AWS cloud infrastructure and managed directly by VM instances. Actually, these keys are directly accessible to all software running on a VM instance associated with a role, through a special HTTP-based *instance metadata service* interface. This means principals using IAM role authentication can't be smaller than a VM instance, and all software executing on the VM must be trusted to safeguard the keys. Additionally, any component that accepts DNS names or IP addresses from an untrusted source must take special care to avoid the confused deputy problem [28], in which access to data from the instance metadata service is inadvertently exposed.

IAM roles work for hosted authentication but are unsuitable for intra-cloud or inter-cloud authentication. In particular, while AWS native services can validate IAM role keys, there is no mechanism for third parties to validate the keys or messages signed with them.

2) AWS Instance Identity Documents: The AWS instance metadata service exposes an *instance identity document*, describing certain details of the instance, along with a signature of that data using a per-region AWS public key. In contrast to IAM roles, the public key signature allows an instance identity document to be verified by third parties, making it seem attractive as a means for directly authenticating software components in intra-cloud or inter-cloud scenarios. Without the inclusion of a nonce, however, the instance identity document is not directly usable for authentication. Several attempts have been made to work around the lack of nonce, though these are somewhat convoluted and not yet complete.[3]

B. Microsoft Azure

Services provided by or hosted within the Microsoft Azure cloud, like AWS, make use of a wide variety of authentication mechanisms. For hosted authentication scenarios, many Azure services use a combination of HTTPS and API keys. API keys are typically used with HMAC to authenticate messages sent to Azure services or, in some cases, as part of a proof-of-knowledge challenge-response protocol. Some Azure services support *shared access signature tokens*, which act as capabilities that can either reference or directly embed policies. The tokens are minted using HMAC with a master secret API key. The policy contained in (or referenced by) these tokens can include IP addresses, expiration time, user names, and other factors, and they support revocation.

1) Azure API Management Service: Azure provides an API Management service to simplify the creation of APIs and associated back-end servers, specifically targeting APIs that allow intra-cloud and inter-cloud communication. As a trusted Azure service, the API Management gateway has the ability to validate Azure tokens on behalf of the API implementation without exposing the secret tokens to the API's back-end servers. This would be a convenient place to support the direct authentication of software components, but the API Management service only supports authentication using API keys, tokens, and IP addresses.

2) Azure Active Directory: Azure Active Directory (AD) provides an identity management service that can be used to authenticate users, VM instances, applications, or other principals within or outside the Azure cloud. Once authenticated to AD, the principal can obtain from it a variety of *authentication tokens*. These tokens are signed with a private key and can be verified by any third party that holds Azure's public key certificate. Tokens can be restricted for use at a specific service, making them useful for intra-cloud and inter-cloud authentication. For example, consider how component A, executing on a VM instance within Azure's cloud, could authenticate some component B that executes elsewhere. B would first authenticate to AD to obtain a restricted token

[3]See: https://github.com/hashicorp/vault/issues/828, http://ryandlane.com/blog/2015/06/16/custom-service-to-service-authentication-using-iam-sts, http://ryandlane.com/blog/2015/06/16/custom-service-to-service-authentication-using-iamkms, and https://aws.amazon.com/blogs/apn/identity-federation-and-sso-for-saas-on-aws.

identifying A by name. B can then send the token along with requests to A. A can then authenticate the requests as coming from B: first, verify the token using Azure's public key certificate; then check to ensure the token includes the appropriate restriction to A's service. Because the token restrictions identify A explicitly, A can't then use this token to impersonate B at some other service. However, ultimately any such use of Azure AD is rooted in secrets—either a private key or a password—that must be maintained by the component B to authenticate to AD itself. Thus Azure AD does not currently support direct authentication of software components.

C. Rackspace and OpenStack

Rackspace runs a version of the OpenStack cloud infrastructure that uses a common mechanism for all hosted authentication. Principals first authenticate to an identity service, which then returns a bearer token that can be used for authentication to other services. For hosted authentication, a component would typically use HTTPS to first authenticate the service, then include the bearer token verbatim in the HTTP headers of each subsequent request.

In the current implementation, each bearer token is generated using an authenticated encryption scheme and contains all associated metadata such as timestamps, user and tenant IDs, and some restrictions on token use. Encrypted tokens are validated using Rackspace's public key, but a database of revoked tokens is also maintained by the identity service and must be checked during each token validation.

Unfortunately, Rackspace bearer tokens can't be restricted for use at a single service, so they can't easily be used for inter-cloud or intra-cloud authentication. For example, if a component B were to use a bearer token to authenticate to component A, A would need to be trusted to not use the token to later impersonate B by using that token. Additionally, the entire scheme is rooted in secrets managed directly by components, since components must authenticate to the Rackspace identity service itself using either passwords or API keys.

D. Google Cloud Platform

Google Cloud Platform (GCP) uses API keys for certain hosted authentication scenarios. GCP also supports IAM and role-based access controls in a manner similar to AWS, signed policy documents, and signed URLs that act as capabilities to access restricted resources.

GCP provides extensive support for the use of short-lived OAuth 2.0 [13] bearer tokens. Components running on a GCP VM instance obtain these tokens through an instance metadata service, as follows. Each VM is associated at creation with one or more *service accounts* containing key pairs. The service account public key, or an email address linked to it, can be used as a principal in authorization policies, and the private key is installed in the instance metadata service. A component running within a VM instance sends a request to the metadata service and obtains in response a JWT [29] token signed by the private key. It then sends the JWT token to a GCP OAuth server to obtain an OAuth bearer token that can be used to authenticate to GCP services.

GCP's use of service accounts is specifically meant to allow for direct authentication of software, since "service accounts are special accounts that represent software rather than people."[4] Private keys associated with service accounts are not directly exposed to components during execution, but are instead maintained by the metadata service, a trusted third party. Service accounts can be used for intra-cloud and inter-cloud authentication as well, since GCP provides an API for validating OAuth tokens derived from a service account and, much like Azure AD tokens, GCP OAuth tokens can include the names of both endpoints when components are authenticating each other. Still, service accounts do not directly authenticate specific software components, but are instead opaque identifiers. Moreover, they can't be associated with principals smaller than a VM instance, and developers can request that additional key pairs be generated for a service account, with the private key managed directly by the developer, making it more difficult to audit the system.

V. Discussion

Authentication in Reddit and other cloud-deployed systems is hampered by the need to provision components with secrets, such as passwords, API keys, bearer tokens, and TLS keys. One approach to address this problem relies on a dedicated key-management service (KMS) or hardware security module (HSM). AWS provides a KMS, for example, and has more recently introduced a CloudHSM service. Tutament [30] and a variety of industry solutions have recently been proposed along similar lines. AlBelooshi et al. [31] survey recent work in this area. However, while these services can play a vital role in safeguarding secrets, they don't addresses the fundamental problem of how to bootstrap authentication of components in the cloud. They instead only shift the problem to authentication between components and the KMS or HSM itself.

Hatman [32] is a alternative reputation-based approach to dealing with components within a cloud that are not fully trusted. Component topology can also be engineered to be more tolerant to some failures [4], reducing the need for components to trust each other.

We argue for direct authentication of software components, without reliance on administrators, developers or components managing API keys, passwords, TLS keys, or other secrets. In this scheme, cloud providers, by acting as trusted third parties, would play a central role in inter-cloud and intra-cloud authentication of components hosted in the cloud. This is essentially a call for remote attestation in a manner similar to that provided by a trusted platform module [33] or various software-based attestation schemes [34]. But here, we can leverage the fact that components running within a cloud platform typically already trust the cloud platform infrastructure. In fact, the cloud infrastructure already has unique and privileged insight into the specific software and configuration for every VM

[4]https://cloud.google.com/storage/docs/authentication

instance, along with information about tenant, machine, VM, and even process identity. This information could be the basis for strong authentication between components. Yet while some cloud platforms expose some of this information, even going so far as signing it so it can be verified by third parties, none of the cloud platforms we examined appear to allow this information to be used as part of an authentication mechanism.

We envision two possible mechanisms: a certificate-based approach, and a token-granting service. In the first, a cloud provider would issue certificates to hosted components, on demand, attesting to details about the components, such as a hash of the VM image or a tenant ID. These certificates could be bound to transient secrets generated by the components, e.g., temporary API keys to sign HTTP requests or a temporary TLS key to authenticate an HTTPS session, or the certificates could include nonces generated as part of an online authentication protocol. For inter-cloud authentication, the cloud provider key used to sign certificates can be installed in remote components, either manually or through a PKI.

A token-granting approach provides each cloud-hosted component a means for obtaining from the cloud provider a token detailing information about the component itself. But unlike with existing cloud identity servers, components would not need to authenticate to the token-granting services using a secret. Instead the cloud platform would leverage its privileged position to directly attest to the information in a token.

Either of the above approaches would serve to allow software components to be directly authenticated, independently of developers and the administrators responsible for deploying and managing the components, thus allowing for a smaller TCB that is easier to audit and analyze. In fact, by replacing many disparate, ad-hoc mechanisms, it could serve to reduce the TCB substantially. In the case of Reddit, it would remove from many TCBs all of the domain name system and the HTTPS PKI, IP routing on the internet and within the cloud, and network firewalls, and it would allow for the security of each component to be analyzed independently of most others.

VI. Acknowledgments

Tom Roeder suggested the classification in Section II and provided invaluable comments on this manuscript.

References

[1] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Proc. Int. Conf. on Cloud Engineering (IC2E)*, Apr. 2016.

[2] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[3] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A security analysis of Amazon's Elastic Compute Cloud service," in *ACM Symp. on Applied Comput.*, 2012, pp. 1427–1434.

[4] K. M. Lhaksmana, Y. Murakami, and T. Ishida, "Cascading failure tolerance in large-scale service networks," in *IEEE Int. Conf. on Services Comput.*, 2015, pp. 1–8.

[5] J. Oh, J. Park, S. Park, and J.-J. Won, "TAaaS: Trustworthy authentication as a service," in *IEEE Int. Conf. Cloud Comput.*, 2016.

[6] H. K. Lu, "Keeping your API keys in a safe," in *IEEE Int. Conf. Cloud Comput.*, 2014, pp. 962–965.

[7] S. Farrell, "API keys to the kingdom," *Internet Computing*, vol. 13, no. 5, pp. 91–93, Sep. 2009.

[8] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *Proc. Working Conf. on Mining Software Repositories*, May 2015, pp. 396–400.

[9] "TIFU by editing some comments and creating an unnecessary controversy," https://www.reddit.com/r/announcements/comments/5frg1n/, Nov. 2016.

[10] S. B. Roosa and S. Schultze, "Trust darknet: Control and compromise in the internet's certificate authority model," *IEEE Internet Comput.*, vol. 17, no. 3, pp. 18–25, Feb. 2013.

[11] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS certificate ecosystem," in *Proc. Internet Measurement Conf.*, Oct. 2013.

[12] N. Leavitt, "Internet security under attack: The undermining of digital certificates," *Computer*, vol. 44, no. 12, pp. 17–20, 2011.

[13] D. H. (Ed.), "The OAuth 2.0 authorization framework," Internet Eng. Task Force RFC 6749, 2012.

[14] R. T. F. (Ed.) and J. F. R. (Ed.), "Hypertext transfer protocol (http/1.1): Authentication," Internet Eng. Task Force RFC 2617, 2014.

[15] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, no. 124, Aug. 2004.

[16] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, , and V. Venkataramani, "Scaling memcache at Facebook," in *Proc. Networked Syst. Design & Implementation*, 2013, pp. 385–398.

[17] A. Melnikov and K. D. Zeilenga, "Simple authentication and security layer (SASL)," Internet Eng. Task Force RFC 4422, 2006.

[18] I. Malpass, "Measure anything, measure everything," Feb. 2011.

[19] C. Davis, "Graphite," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds., 2014, ch. 7.

[20] M. Y. Becker and P. Sewell, "Cassandra: Flexible trust management, applied to electronic health records," in *Proc. IEEE Comput. Security Found. Workshop*, Jun. 2004.

[21] J. Myers, "SMTP service extension for authentication," Internet Eng. Task Force RFC 2554, 1999.

[22] H. Lampesberger, "Technologies for web and cloud service interaction: A survey," *Service Oriented Computing and Applications*, vol. 10, no. 2, pp. 71–110, Jun. 2016.

[23] J. Kelsey, B. Schneier, and D. Wagner, "Protocol interactions and the chosen protocol attack," in *Security Protocols*, ser. Lecture Notes in Comput. Sci., B. Christianson, B. Crispo, M. Lomas, and M. Roe, Eds., vol. 1361, 1998, pp. 91–104.

[24] "MDx-MAC and building fast MACs from hash functions," in *Crypto'95*, 1995.

[25] J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Proc. USENIX Winter*, Feb. 1988, pp. 191–202.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. Annu. Tech. Conf.*, 2010.

[27] "Amazon Web Services: Overview of security processes," https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf, Oct. 2016.

[28] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *ACM Operating Syst. Review*, vol. 22, 1988.

[29] B. Campbell, C. Mortimore, and M. Jones, "JSON web token (JWT) bearer token profiles for OAuth 2.0," IETF Network Working Group Draft Standard, 2012.

[30] "Tutamen: A next-generation secret-storage platform," in *ACM Symp. on Cloud Comput.*, 2016.

[31] B. AlBelooshi, E. Damiani, K. Salah, and T. Martin, "Securing cryptographic keys in the cloud: A survey," *IEEE Cloud Computing*, vol. 3, no. 4, pp. 42–56, 2016.

[32] S. M. Khan and K. W. Hamlen, "Hatman: Intra-cloud trust management for Hadoop," in *IEEE Int. Conf. Cloud Comput.*, 2012, pp. 494–501.

[33] Trusted Computing Group, "Trusted platform module (TPM) specification, version 1.2," https://www.trustedcomputinggroup.org/specs/TPM/, Mar. 2011.

[34] A. Ghosh, A. Sapello, A. Poylisher, C. J. Chiang, A. Kubota, and T. Matsunaka, "On the feasibility of deploying software attestation in cloud environments," in *IEEE Int. Conf. Cloud Comput.*, 2014, pp. 128–135.