Authentication When You Aren’t Human

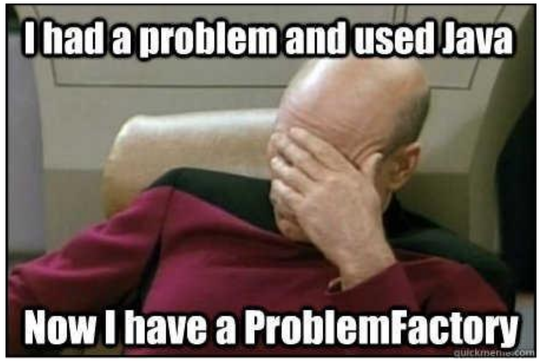
John Haldeman
john.haldeman[at]gmail.com

Hackforge Software Guild Meeting: August 26, 2015

My Context

University of Calgary:
BSc. (CPSC) Degree + BComm Degree

Software Engineering and writing code mainly in Java (maybe because of James Gosling - but other languages too when it made sense for the course)



Databases, Data security, Business Intelligence, Enterprise Search, Partner Enablement:

Writing Code as often as I can (and when it's useful) – Mainly to integrate systems

Security. Specifically Data encryption, database activity monitoring, database vulnerability assessments, sensitive data classification:

Writing Code as often as I can (and when it's useful) – Mainly to extend core products or build tools

In Short

- Someone who is ***not*** a practitioner in identity management (authentication related stuff), ***nor*** software development is going to talk to software developers about identity management

- Nonetheless, it’s an interesting area...

Your Context?

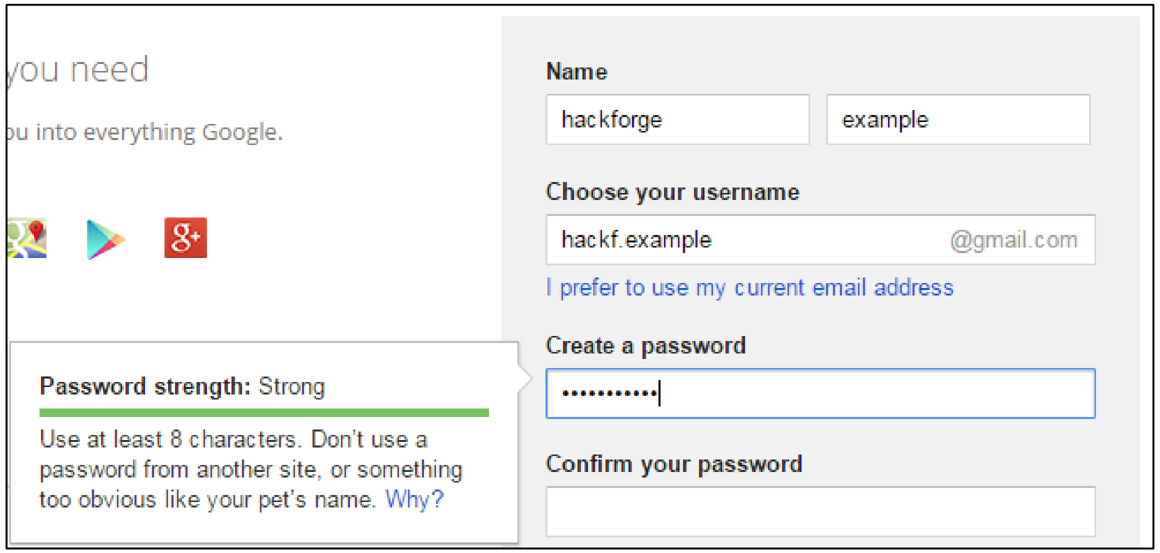
- Anyone in the room a “security person”?
- Anyone ever had to go through a security audit of their code/systems?
- Anyone build software that logs into other systems (eg: any application that logs into a database)?

Authentication

- Verifying the client is who they say – Most often done with a password
- This presentation is not about every day passwords people use. For every day passwords, we all should know about:
  - Password Complexity
  - Changing Passwords
  - Limiting Password Reuse
  - Two Factor Authentication

Authentication When You *Are* Human

- You sign up for a gmail account and set up 2 factor authentication:
- You use something you know:
  - You pick a password that is hard to guess (or have a computer guess)
- In addition you use something you have (2nd factor):
  - You register a phone number to receive a call that gives you a code to sign in



you need

u into everything Google.

Name

hackforge

example

Choose your username

hackf.example @gmail.com

I prefer to use my current email address

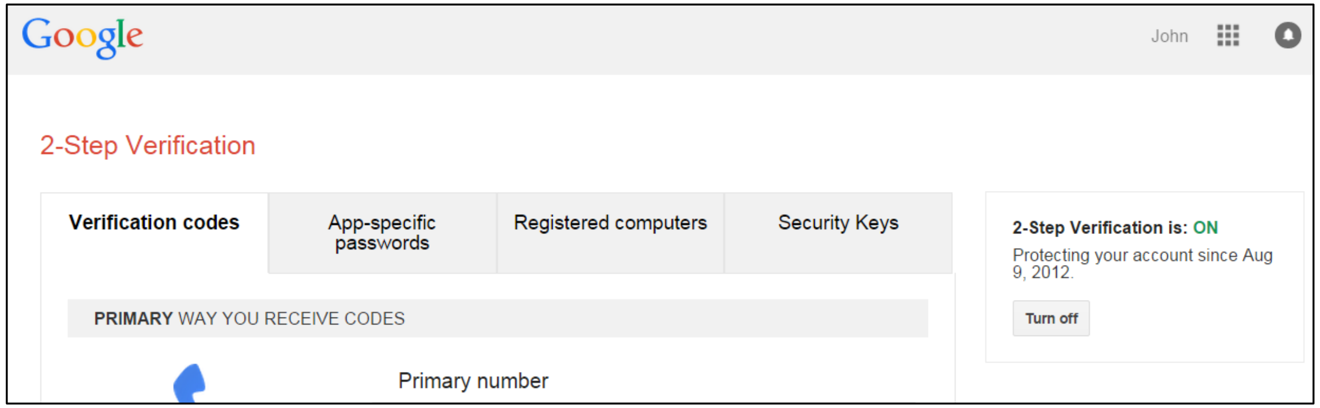
Create a password

••••••••••••

Confirm your password

Password strength: Strong

Use at least 8 characters. Don't use a password from another site, or something too obvious like your pet's name. Why?



2-Step Verification

Verification codes	App-specific passwords	Registered computers	Security Keys

PRIMARY WAY YOU RECEIVE CODES

Primary number

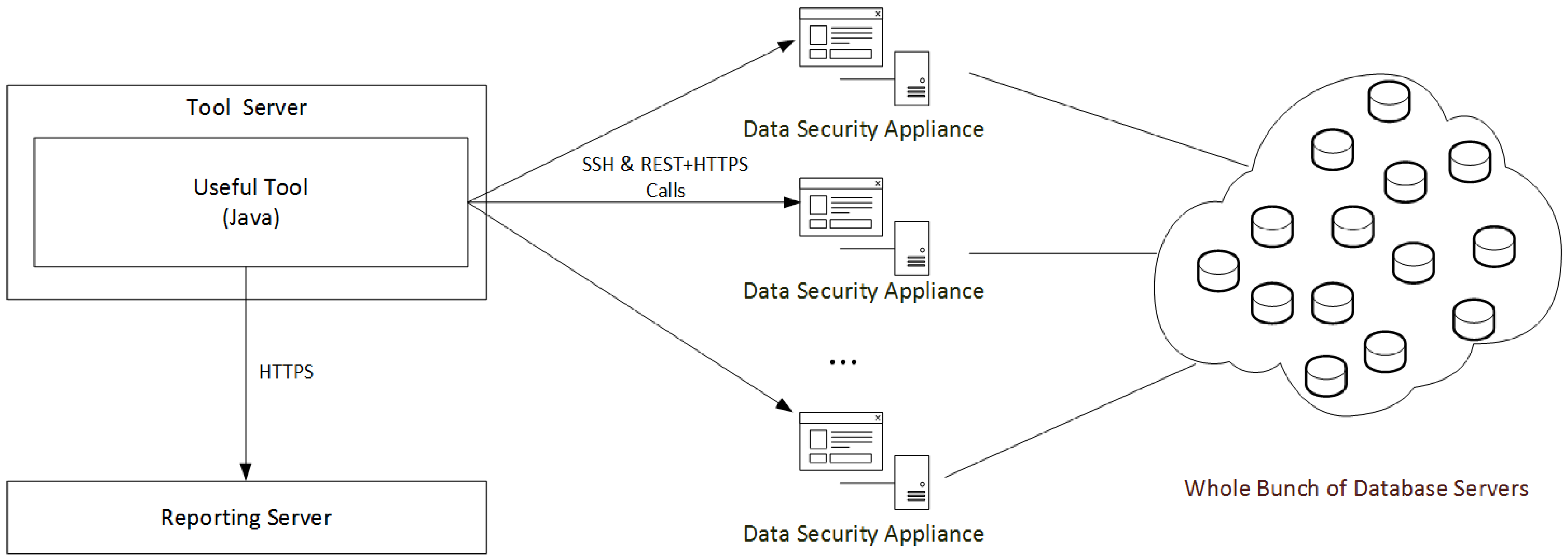
2-Step Verification is: ON

Protecting your account since Aug 9, 2012.

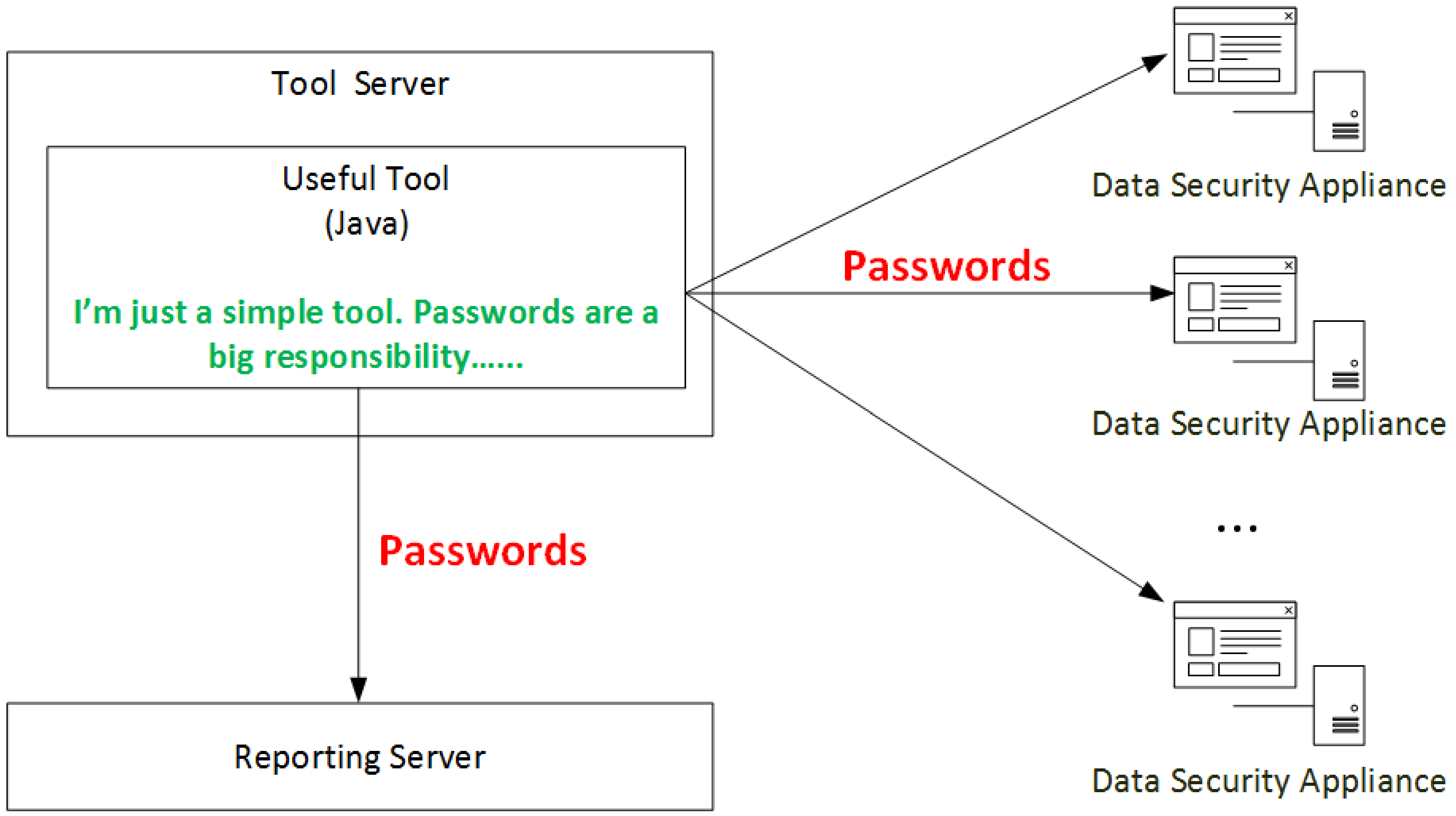
Turn off

A Tool – A service, not a human

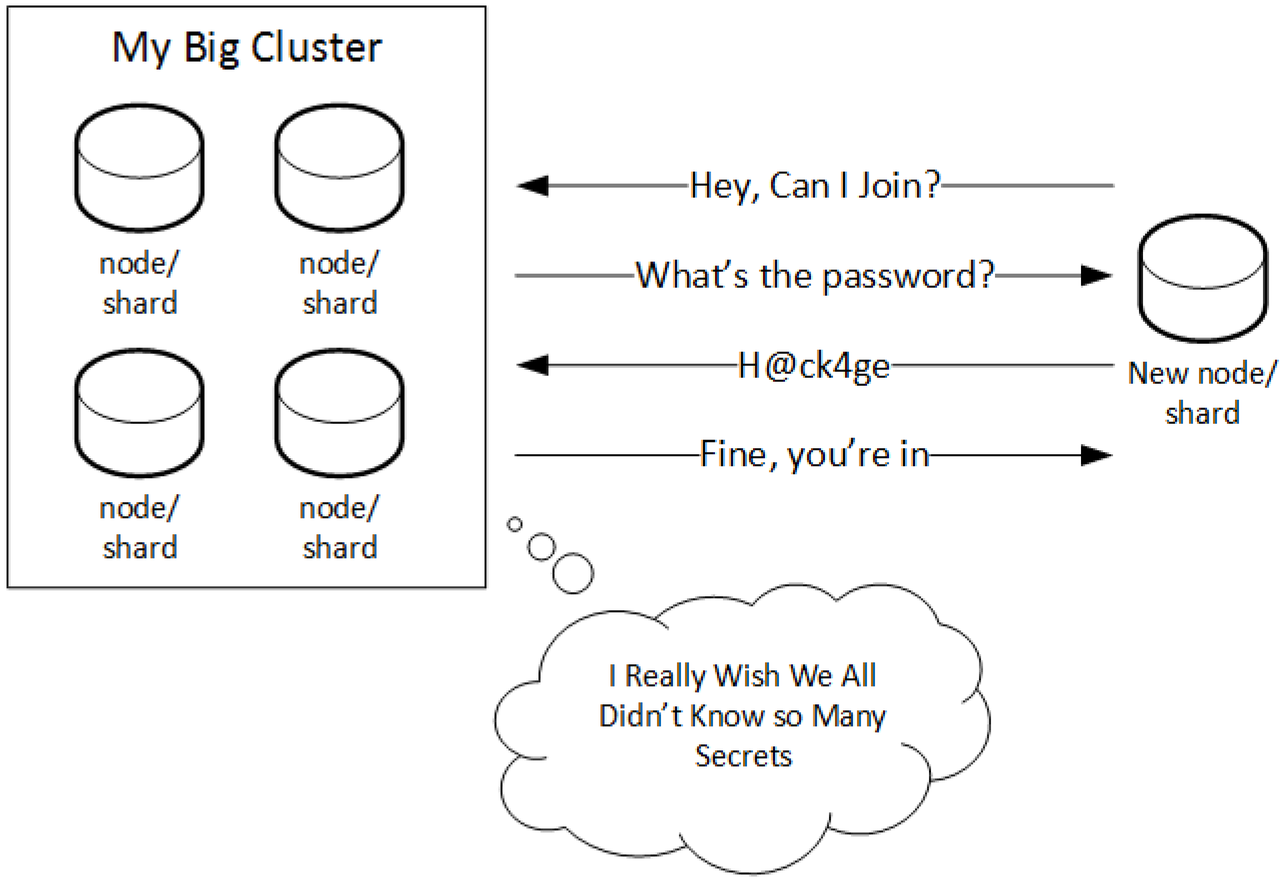
- If you have lots of appliances, you want consolidated views of appliance health
- Appliances are specialized – Sometimes generic stuff (SNMP Traps/System Resources Monitoring) is not enough
- Build a Tool!



Our Problem: Authentication when you ***aren't*** human



Another Example – Peer Authentication



Security Refactoring

- A smell: An uneasy feeling you get that makes you want to change something

- You Know Bad code smells (Code Refactoring)
  - Mega Methods/Classes
  - Feature Envy
  - Copy/Pasted Code
  - …..

- Bad Authentication Smells?
  - Having to trust Clients
  - Passwords in Configuration Files (or clear text anywhere)
  - Relying on another secret, possibly in code, to encrypt passwords

The main question to investigate

- If an application’s server is compromised, can you avoid the potential of the servers that it connects to being compromised as well?

- Analogy: Fire doors – Instead of the entire building burning down, fire doors contain the fire’s spread by blocking off areas on fire

Tomcat – JDBC, JNDI, Configuration Files

- Quick Intro:
  - **JDBC**: A way to access a database in Java
  - **JNDI**: A way to retrieve a named resource in Java
  - **JDBC + JNDI**: Separates the definition of the database connections (and connection pools) from the code – JDBC data source retrieved using JNDI
  - **Tomcat**: An Application Server – It runs Server-Side Java code and Serves Up Web Pages

Defining a postgres database in Tomcat to be Used by an Application

In a configuration file somewhere:

```
<Context>

<Resource name="jdbc/postgres" auth="Container“
type="javax.sql.DataSource" driverClassName="org.postgresql.Driver“
url="jdbc:postgresql://10.10.9.28:5432/mydb" username="myuser"
password="mypasswd" maxActive="20" maxIdle="10“ maxWait="-1"/>

</Context>
```

Problems with passwords in configuration files

- The security of your database server (or any other system you are connecting to) is now only as good as your application server’s security
- God forbid you do have account compromise – how easy is it to change account passwords? What do you do when an Application Owner leaves the company?
- Owners of the application server can log into the database with an application credential (service account) – Obscuring their identity
- Quite often administrative tools obscure these seemingly obvious facts. *“The system took care of it, I’m sure it’s okay”*

Can we do any better…

- I know! Let’s Encrypt it!
    - Store the password:
        - encrypt(password, ***key***)
    - Retrieve the password:
        - decrypt(password, ***key***)
    - Where do we put ***key***…………….
    - ***key*** becomes the new problem. See:
        - http://wiki.apache.org/tomcat/FAQ/Password
- Wait… I heard people hash (as opposed to encrypting) passwords? Doesn’t that work here?
    - What makes HASHes great is that they are one way functions
    - What makes HASHes inappropriate for this use case is that they are one way functions

Password Authentication

- In short:
  - Storing a password on the peer/client is not great (whether you can avoid this at all – we'll discuss later)
  - Storing the encrypted password on a peer/client is only marginally better
    - Arguably it might be worse because you can look someone in the eye (say, a security auditor) and say "The passwords aren't stored in clear text"
- Where I've seen it:
  - Very widespread, especially when connecting to databases

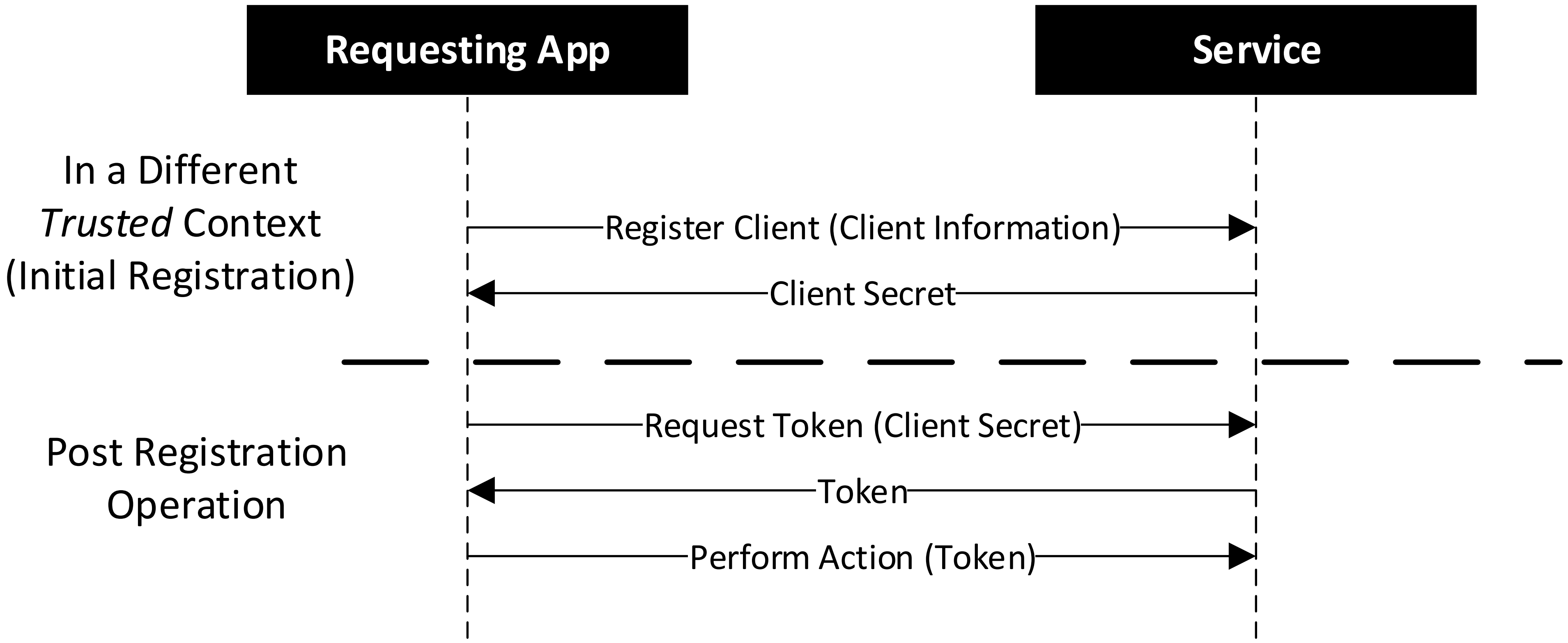
OAuth and Tokens

- You might run across a web service that says you must use something called an OAuth token to authenticate

- OAuth was originally designed to allow web services to have limited access to other web services
  - Eg: Facebook logging into gmail to help a user find friends based on their contact list

- Becoming a more generic web service authorization framework

Very simplified OAuth token request flow



Requesting App
Service
In a Different *Trusted* Context (Initial Registration)
Register Client (Client Information)
Client Secret
Post Registration Operation
Request Token (Client Secret)
Token
Perform Action (Token)

OAuth

- *Tokens* – Temporary and give you access to a resource for the time you need it

- *client_secret* – Something you use to request tokens. This is a password that you have to store somewhere to use it

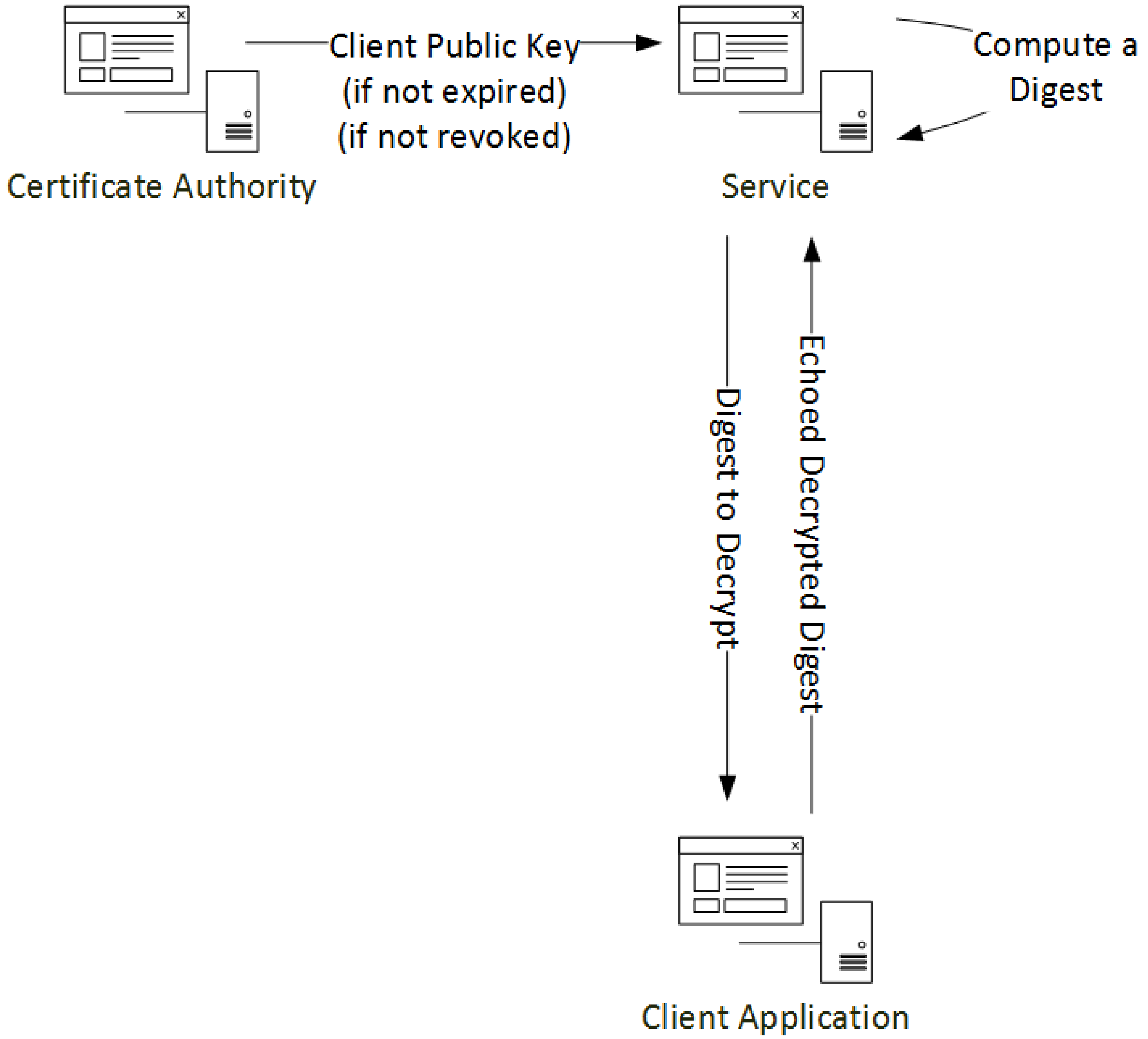
- OAuth is a great tool to help set up external authentication and limited access to the resources in your service – but it does not solve our problem

```
{
   "client_id":“app1",
   "client_secret":"3ac89782-ce55-4f24-b795-b6c76ecc4045",
   "grant_types":"password",
   "scope":"read,write",
   "redirect_uri":“……"
}
```

Authentication with OAuth

- In short:
  - A useful standard with features like tokens that help with certain security issues
  - Because of the use of the *client_secret*, it is effectively password authentication
- Where I’ve seen it:
  - Administrative tooling that exposes REST+HTTP APIs

Certificates



- Note this is a little different than what some people might be used to because we are authenticating a ***client***, not a server, but the idea is the same

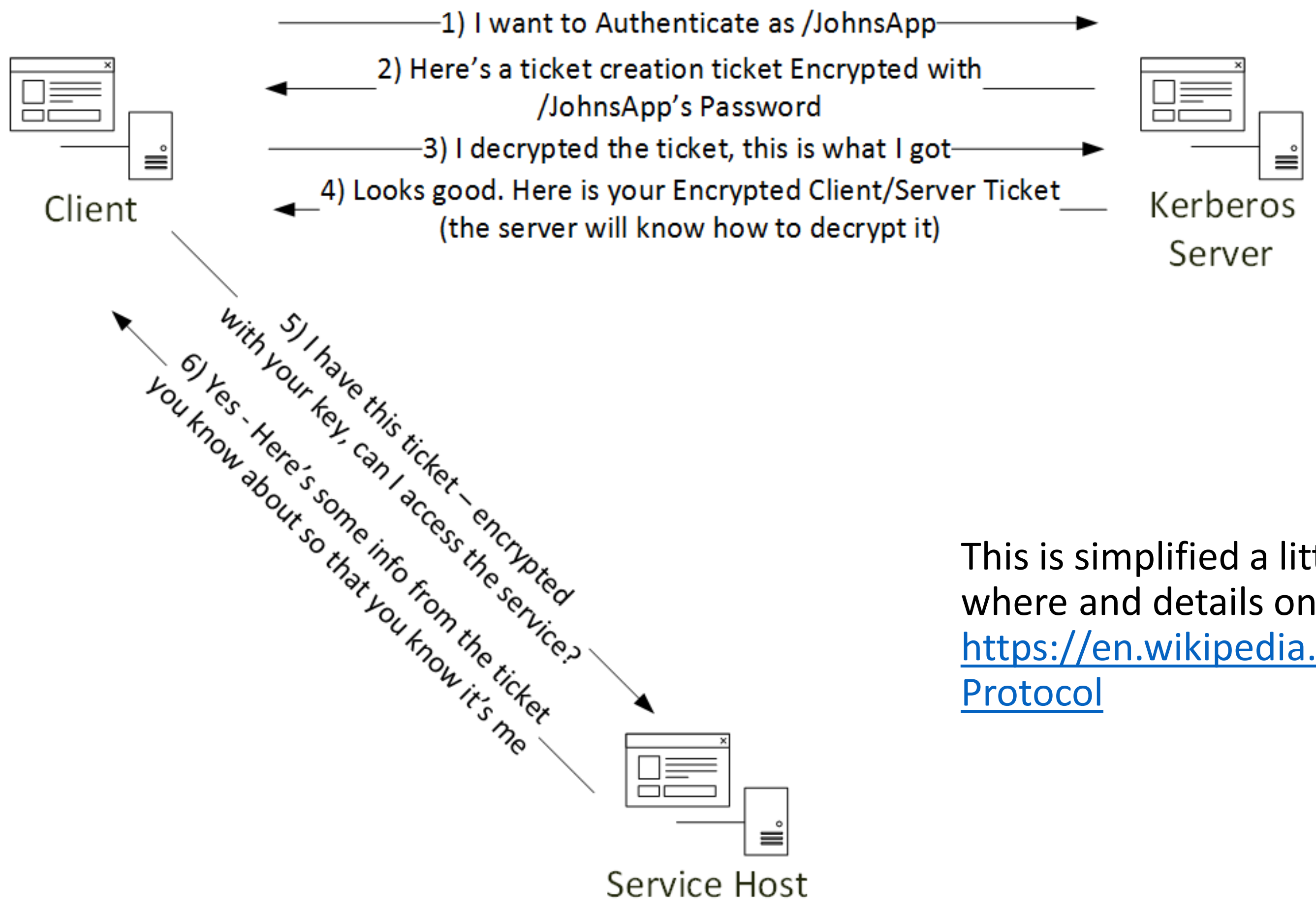
- Important distinction from passwords:
    - Uses asymmetric cryptography
    - No secret data needs to be transmitted off of the Client for it to work

Certificates

- In Short
  - Has some distinct advantages (and some disadvantages) over password authentication
  - The private key (to sign the data) is kept on the client. If you get the private key you can authenticate as that client – The private key replaces the password here so things still smell
- Where I’ve Seen It
  - Data Encryption Software – The agents performing the encryption authenticate themselves with certificates in order to get encryption keys from a key server

Kerberos

- Kerberos lets you delegate authentication to a third party, similar to certificates – but it's ticket based


1) I want to Authenticate as /JohnsApp
2) Here's a ticket creation ticket Encrypted with /JohnsApp's Password
3) I decrypted the ticket, this is what I got
4) Looks good. Here is your Encrypted Client/Server Ticket (the server will know how to decrypt it)
Client
Kerberos Server
5) I have this ticket – encrypted with your key, can I access the service?
6) Yes - Here's some info from the ticket you know about so that you know it's me
Service Host

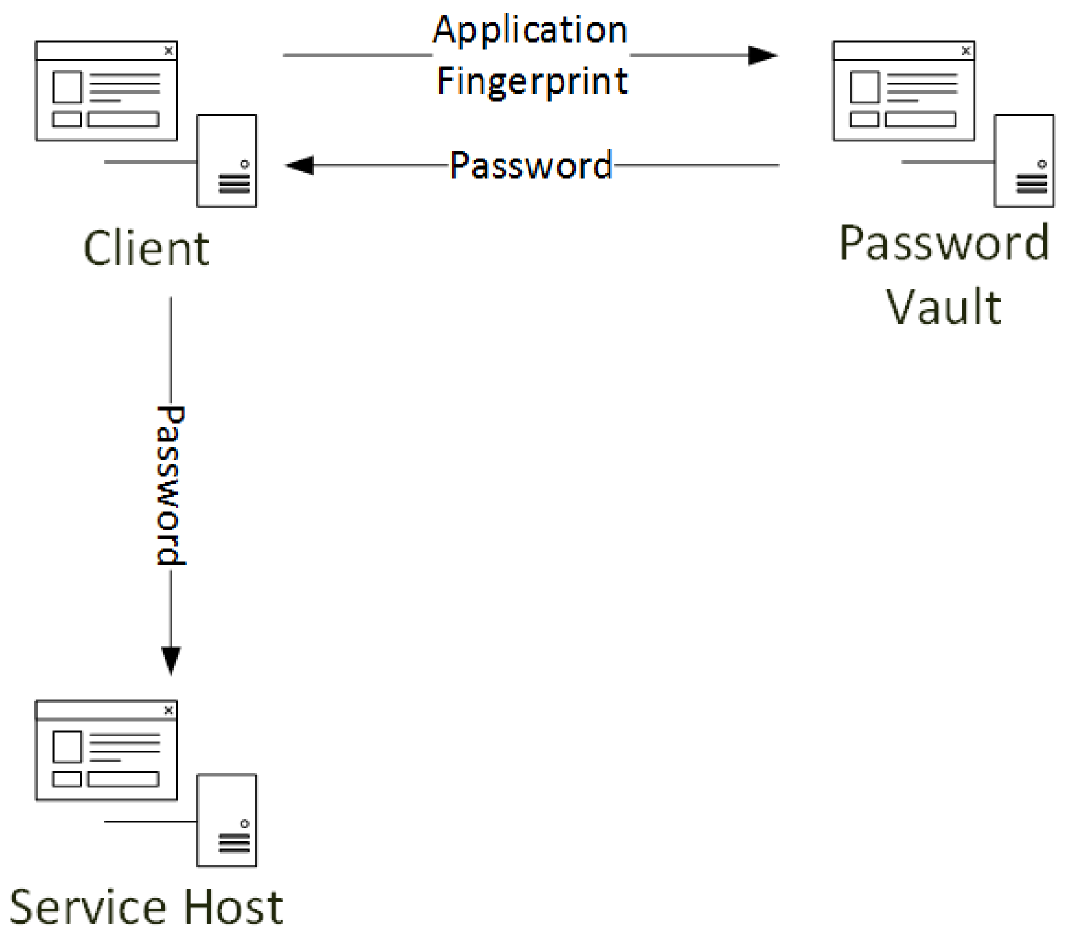
This is simplified a little. For what keys are used where and details on how it works see: https://en.wikipedia.org/wiki/Kerberos_(protocol)#Protocol

Kerberos

- In Short
  - Lots of advantages – again the authenticator is a third party
  - Some disadvantages – Not known to be an easy thing to set up
  - Still relies on passwords on the client

- Where I’ve seen it:
  - Hadoop nodes authenticate each other via Kerberos
  - Many Linux/UNIX services are Kerberized and can use Kerberos
  - If you’ve ever authenticated to Active Directory (by logging into Windows, or having a Service log into AD using the services panel) – That’s a Kerberos protocol implementation

Application Identity Management Systems



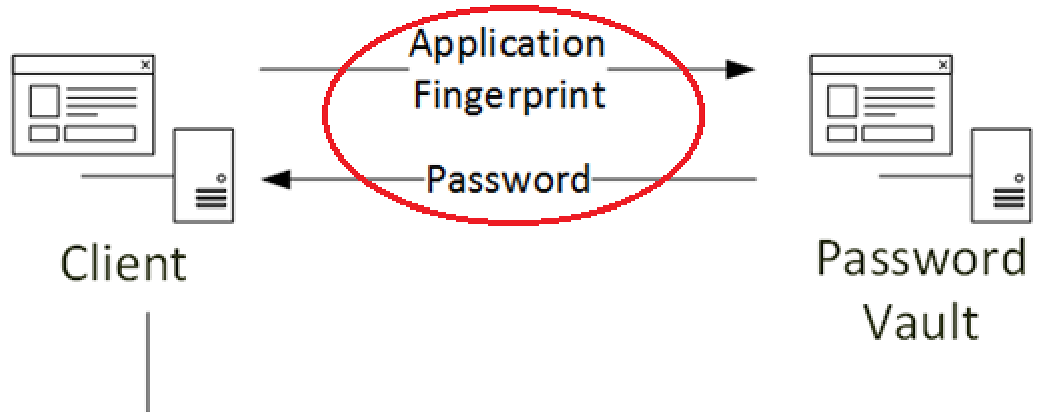
Application Identity Management Systems

- What a fingerprint looks like (example from IBM Privileged Identity Manager)

```
{ "userName":"john.haldeman",
"hostName":"JHaldeman-T440p",
"group":"scripts",
"hostTimezone":"America/New_York",
"creationTime":1427224852736,
"binaryHash":{
        "path":"30C7D9851769AC42639113E26FC6D69C9AF6C623FF19A5D80E807E07742B1A3D",
        "transformation":"CONSTANT"
},
"user":{
        "directory":"C:\\Users\\John.Haldeman\\DownloadDirector\\PIM_2.0_CLIENT_SDK_ML",
        "home":"C:\\Users\\John.Haldeman",
        "language":"en",
        "country":"US"
},"network":[{"ipAddress":"fe80:0:0:0:7c69:cc:70e6:e2a5%eth4",.......}] }
```

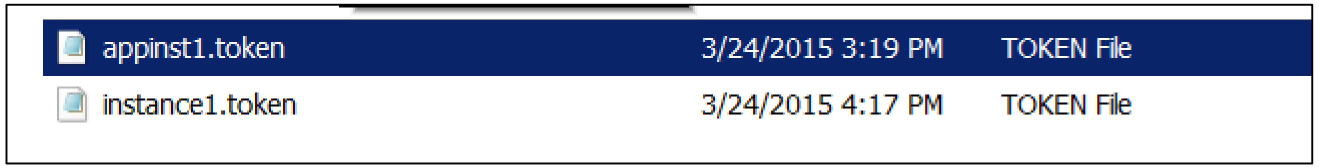
Application Identity Management Systems

- Any passwords?
- What about this interaction:

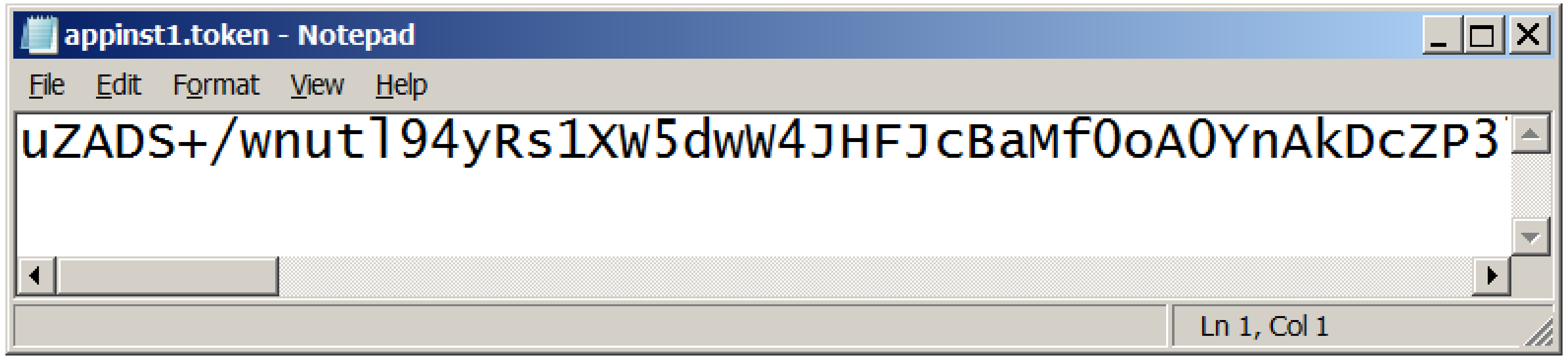


- How does the password vault authenticate the client that uploads the fingerprint……….

On Further Investigation: Token files..........



```
appinst1.token      3/24/2015 3:19 PM    TOKEN File
instance1.token     3/24/2015 4:17 PM    TOKEN File
```



```
uZADS+/wnutl94yRs1XW5dwW4JHFJcBaMf0oA0YnAkDcZP3
```

Conceivably if you knew this password and the fingerprint you can get the password from the vault

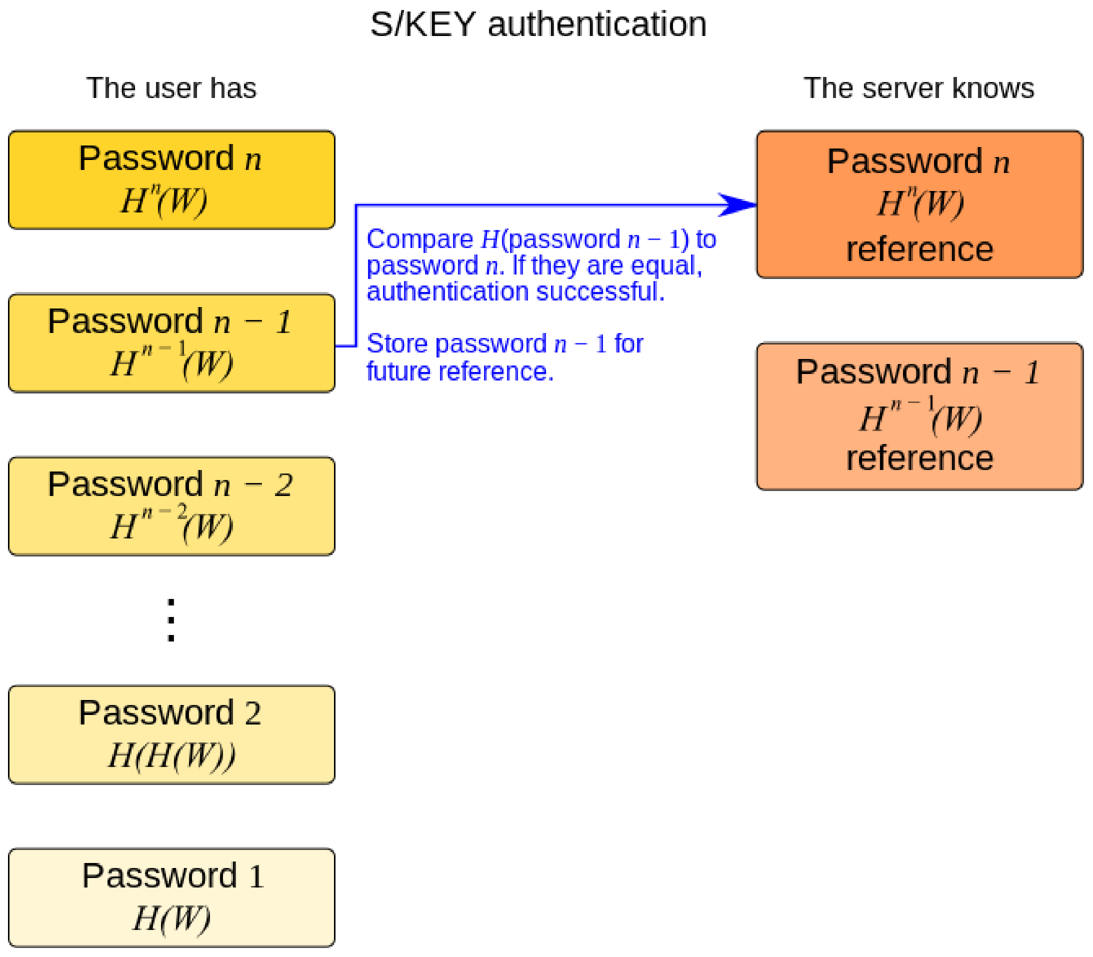
Revisiting 2-Factor Authentication

- Two factor authentication is also sometimes called “One Time Password” use

- Could you create a one-time password system for authenticating applications…

S/KEY (Lamport's scheme)

- Generation Process:
  - A secret key is provided or automatically generated
  - The key is hashed repeatedly (*n* times). Each hash is recorded and kept on the client. The last hash is kept on the server
  - The secret key is discarded
- Authentication process
  - The client sends only the last key in it's key list and then removes the key from the list
  - The server hashes the key received from the client. If it matches the previously authenticated key, authentication is successful


S/KEY authentication
The user has
The server knows
Password n $H^n(W)$
Password $n-1$ $H^{n-1}(W)$
Password $n-2$ $H^{n-2}(W)$
⋮
Password 2 $H(H(W))$
Password 1 $H(W)$
Compare H(password $n-1$) to password n. If they are equal, authentication successful.
Store password $n-1$ for future reference.
Password n $H^n(W)$ reference
Password $n-1$ $H^{n-1}(W)$ reference

https://en.wikipedia.org/wiki/S/KEY

S/KEY

- Clearly secrets are known on the client – the list of passwords
- But...
  - What happens if the attacker steals the password list and authenticates themselves?
  - When you try and authenticate again, you send the wrong key and authentication fails
  - At least you get tipped off…
- Where I’ve seen it:
  - Nowhere (although there are implementations of it and other related schemes for Linux/UNIX PAM)

Conclusions

- There are a lot of authentication mechanisms for applications and each provides some advantages/disadvantages
- We seemingly can't get away from some kind of secret on the client: We can just put in layers of obscurity/obfuscation – Each requires at least some kind of client trust
- In Short: You have to protect your application servers as if they were storing the sensitive data

Which brings us to

1) Understand that once one server in an environment is compromised, it becomes easier to compromise others
2) Things to help prevent/detect compromise:
   - Firewalls – A kind of additional authentication by IP and the network
   - Log Analysis/Activity Monitoring
   - Keeping the servers updated and hardened
   - Striving for application/script least privileges
3) You shouldn’t just use password authentication and keep in mind the other options for sensitive applications – Kerberos still has lots of advantages over passwords (but does require infrastructure)