

Protecting Systems with One Time Passwords

Scott Nolin

Steve Barnett

22 March 2017

Contents

1. Introduction
2. Multifactor Authentication and OTP – Scott and Steve
3. OTP on a single server – Steve
4. Beyond a single server – Scott
5. Conclusion

Introductions

Scott Nolin - Head of SSEC Technical Computing Group.

Steve Barnett – IceCube

Why are we here?

- Increase security for critical systems
- This project especially helps with the problem of a compromised password
- This is about improving security, making things better.
 - **Don't think that 'This is secure, they're done'**. Even within this subset of topics, we're continuing to evolve and have plenty of gaps.
- We encourage you to share information if you do similar projects, and we'll all benefit.
- Note: the systems and examples we discuss are all centos/redhat linux

I asked myself a scary question.

What is the easiest and quickest way to compromise every linux host at SSEC?



Answer: Compromise My Password

Mine, or any sysadmin with full permissions. Thinking about trojans and keyloggers made me consider all the great ways my password could be compromised. Or an ssh key passphrase, same difference - if the Trojan is on the machine where you type the passphrase, your key is owned too.

- We have evolved to use configuration management, host our own rpm repositories, and many other things to improve systems management.
- That's great but it also **increased our attack surface**.
- I want to protect these systems better.

What if **your** password is compromised?

Multifactor Authentication and OTP

General information and background

Multifactor Authentication and OTP

One time passwords (OTP) are a convenient and commonly used authentication factor which can help provide multifactor authentication. They can help if a password is compromised.

- An OTP should be resistant to replay attacks – seeing a password should not allow the next to be deduced.
- An OTP is a ‘password’ – but the goal is it’s impossible to reproduce without a particular device (phone, keyfob, etc.) The device that generates the OTP is something you ‘have’.
- Systems can use text messages as an OTP. In this case, the cell service provider is part of your chain of trust.

Acronyms

- MFA : Multi-Factor Authentication – system that requires more than one category of authentication – typically knowledge, possession, and inherence (biometric)
- 2FA : Two-Factor Authentication – using two different categories of authentication
- OTP : One-Time Password – password only valid for one use. A device that generates an OTP can be used as a **possession** factor.
- OATH : Initiative for Open Authentication – Industry group for standard authentication. Produced HOTP, TOTP, OCRA authentication methods, token specs, etc.
- HOTP – HMAC Based OTP Algorithm. Default specification is event based token
- TOTP – Time-based One-time Password Algorithm. Also HMAC based. Token is valid for a time window – this is HOTP with a timestamp instead of event counter.
- OAUTH – Open Standard for Authentication (NOT OATH) – allows access to websites without passwords via ‘secure delegated access’ for third parties.
- U2F (FIDO U2f) – Newer (than OATH) open authentication standard, mostly works with chrome browser, there is a pam module available.

HOTP and TOTP Overview

- Both use HMAC algorithms to generate the One Time Passwords.
- HMAC algorithms TL;DR: hash a secret key and a message. Can verify the integrity and authenticity of a message.
- To generate One Time passwords, you generate a message, and hash it with a secret key.
- The secret key should be nice and long-ish and randomly generated
- In HOTP, the message is just a counter (integer) that increments at each use.
- In TOTP the HOTP message (counter) is replaced with the current date/time.

Takeaways

- Does not require communication between token, protected server or any third party server.
- Requires shared state between protected host and the token.
- In the simple case, requires both sides to share the secret key. You get all the problems of shared key management.
- For TOTP, your clocks will need to be fairly close together. Acceptable time skew is implementation dependent.
- Of course, compromise either the server or the token and the secret key becomes known.

SSH Keys: We're not friends any more.

- Our methods rely on the PAM stack
- SSH Keys **SKIP IT.**
- We've had to work around this issue for things like account expiration (I'll provide the ugly details if you like), but for 2fa/OTP protection, there's no good workaround.
- **Disable in /etc/sshd_config**
 - PubKeyAuthentication no
- **AuthenticationMethods** sshd feature
 - Does not work for me: It's global, all or nothing. I wanted, but cannot have:
 - OTP and (ssh_key or password)
 - I welcome any insights.



SSH Keys: We're still into you.

- Disabling all pubkey authentication works, but is harsh.
- Perhaps you want to enable it for some users/hosts – for example, a system protected by 2fa that has a root key:

Match user root Address 10.25.0.1

PermitRootLogin yes

PubKeyAuthentication yes



OTP on a Single Server



This use case is a good starting point.

- Test and get used to the login flow.
- If you have a relatively small number of users, don't need self-service token setup, and are ok with **SSH only** access it could be a complete solution.
 - SSH tunnels means “ssh only access” isn't as tough as it sounds. I (Scott) used this for my testing phase, I forced myself to only access our systems via a host set up like this.

The private OTP key is stored on the linux server itself. While theoretically with a time based key you could use the same private key on multiple servers, that is a bad idea. And multiple event-based keys will simply break – they will get out of sync.

Capital cost is \$0.00

The Single Server Case: Steve

NOTE: All our systems are Centos or Redhat

- First Use Case: protect the systems that allow access to the detector.
 - Relatively small group of people need access (~75)
 - Scattered around the world
 - Need to work on detector subsystems
- Test setup: Single VM, Google Authenticator



Single Server Setup - Software

1. Install Packages Oathtool, pam_oath – provided by EPEL (6,7)
 1. `yum install pam_oath oathtool`
 2. For Centos 6, I had a flaky path/package that needed this symlink:
 3. `ln -s /usr/lib64/security/pam_oath.so /lib64/security/pam_oath.so`
2. Create a credential file for your users and private keys
 1. `/etc/users.oath` – (or whatever name you want)
3. Configure pam files

Single Server: Pam Setting

- Linux PAM is very flexible, and adding OTP as a requirement works well.
- We protect sshd and sudo, this example is for sshd, but sudo is the same
- To the top of **/etc/pam.d/sshd** add

```
auth      required          pam_oath.so      window=20      usersfile=/etc/users.oath
```

- Only want a group called “otpusers” to do this method?

```
auth [success=1 default=ignore] pam_succeed_if.so user notingroup otpusers  
auth      required          pam_oath.so      window=20      usersfile=/etc/users.oath
```

- For debugging, use ‘requisite’ instead of ‘required’ – this makes login fail immediately and not go to the next step.
- The next step should be the password based login of your choice, or what the “normal” routine is for you.

Sshd configuration

- Configure **/etc/ssh/sshd_config** to contain:
ChallengeResponseAuthentication yes
UsePAM yes
- Secure in ways appropriate for your environment – discuss...
 - Disable SSH keys for all or some
 - Disable root login
 - Disable root login with keys
- Restart sshd

Single Server Setup: credential file

- I use /etc/users.oath – any file is fine.
- Protect the file with permissions (root, 0600)
- The file may contain comment lines starting with '#'
- Syntax is single line per entry
- Users can have multiple keys, and it will check each - so you can use more than one device.
- We use yubikeys (OATH mode) and Google authenticator
- Syntax:
 - `[HOTP,HOTP/T] username - key`
 - HOTP = yubikey or other event type HOTP
 - HOTP/T30 = google authenticator or other time based, 30 second window (= TOTP?)
- You only do a manual entry for a new key. On each subsequent usage, the file is updated with numbers and dates, that takes care of itself.
- Example:
 - `HOTP scottn - <reallylonghexkey>`

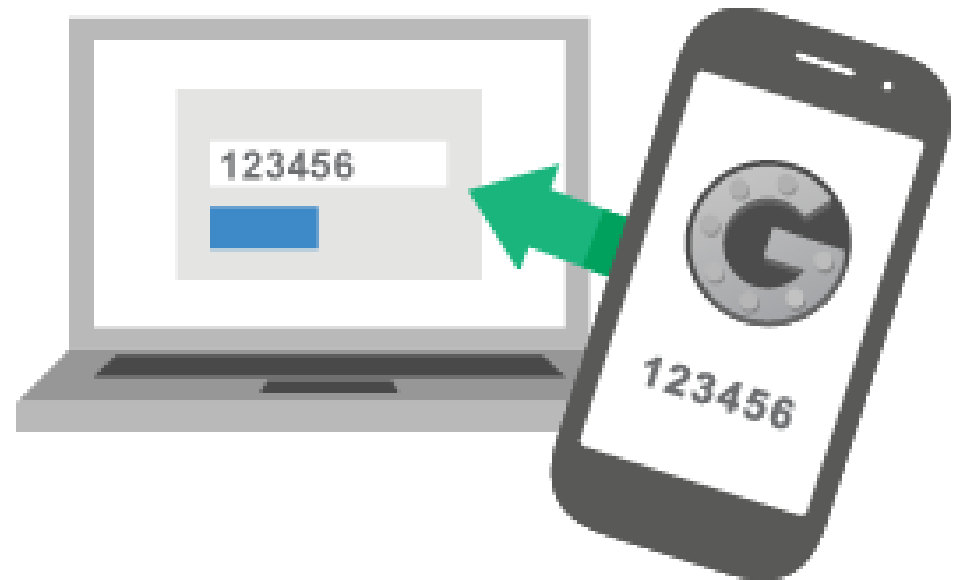
Single Server: Set Up Google Auth Private Keys

- Google Auth: google's secret is base32 - need to use hex in users.oath
- Generate a 20 character random hex

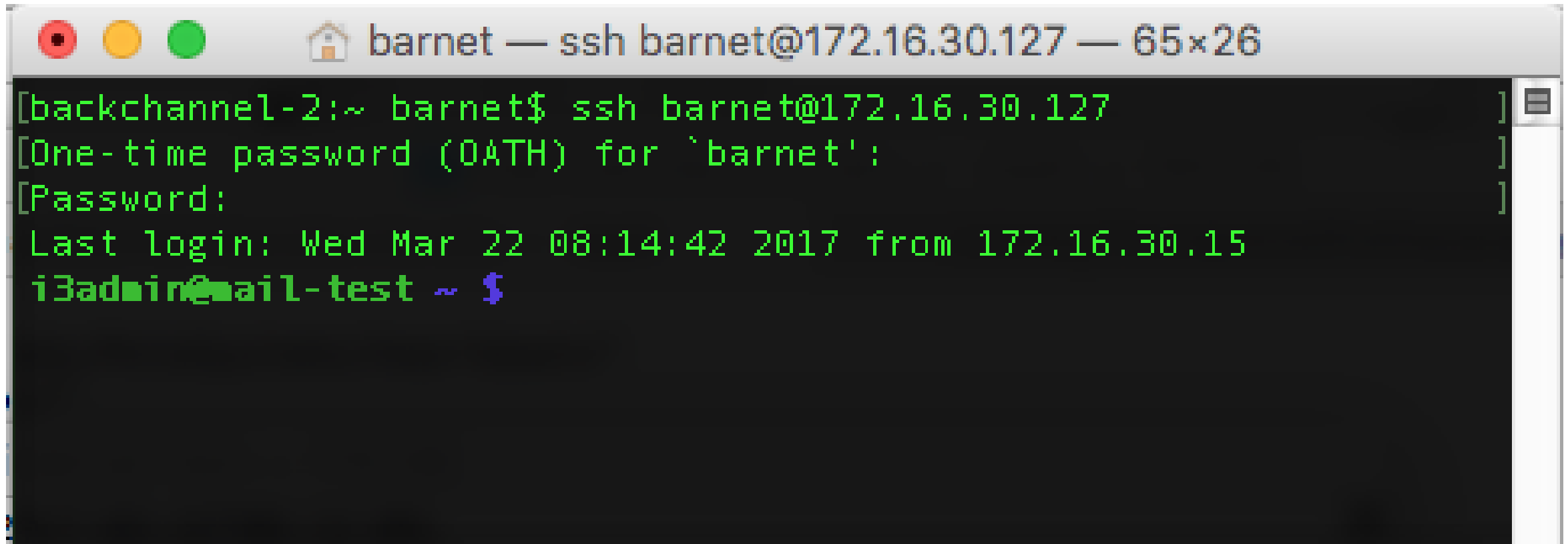
```
head -c 1024 /dev/urandom | openssl sha1 | tail -c 21  
oathtool --totp -v (hex from above)
```

- enter the base32 secret in your google auth
- Add this line to users.oath

```
HOTP/T30 scottn - <google key in hex>
```



Try It Out



```
barnet — ssh barnet@172.16.30.127 — 65x26  
[backchannel-2:~ barnet$ ssh barnet@172.16.30.127  
[One-time password (OATH) for `barnet':  
[Password:  
Last login: Wed Mar 22 08:14:42 2017 from 172.16.30.15  
i3admin@mail-test ~ $
```

Single Server: YubiKey private keys

- I have used yubikey 4.
- The default is to use the Yubico servers, we do not do that, we'll use OATH mode and reprogram the key.
- Use the YubiKey personalization tools application (does not need to be done on your server).
 - Program in OATH-HOTP mode
 - If you want to retain Yubico functionality, program in slot 2, otherwise overwrite slot 1. This gives you 2 keys.
 - Select "Yubikey(s) unprotected - Keep it that way" for protection pin (not present in MacOS app)
 - We stack authentication don't use pin protection
 - Uncheck OATH Token Identifier (6 bytes)
 - Select "HOTP Length" of 6 digits
 - Click "Generate" button to generate your private key. It should appear in the GUI.
 - Click "Write Configuration" to write the configuration to the yubikey.
 - You will be prompted to save the configuration. Save to a CSV file.
 - The file contents will look something like this: OATH-HOTP,1/1/70 12::00 AM,2,,,(MANY HEXADECIMAL NUMBERS)1,,0,0,0,6,0,0,0,0,0,0
 - The private key is the sixth field: (MANY HEXADECIMAL NUMBERS)
- Enter private key in your credential file (/etc/users.oath)
 - HOTP scottn - <reallylonghexkey>



Discussion: Single Server Method

Beyond a Single Server: Scott

- The single server method works and is simple, but has some limitations
 - The key generation and entry requires root, no self-service.
 - Key's can not be used on other servers
 - Definitely not event type keys as they'd get out of sync.
 - Time based you maybe could, but you then are spreading your private keys around
 - A single server as the OTP protected 'bastion' doesn't solve the **sudoers** case.
 - It only works with Linux and PAM
 - We want a web proxy feature. Yes, you could tunnel web, but things like sharing URL's between sysadmins becomes an adventure.

So, we moved on to a centralized system at SSEC. However, **we also set up a single server as a backup method for entry**. It has fewer dependencies, is a standalone island. When things go sideways there's a way to get in remotely. The console is the other option – we chose to leave physical access without OTP, but you can close it if required.

The power of Negative Thinking: Ways to make it terrible.

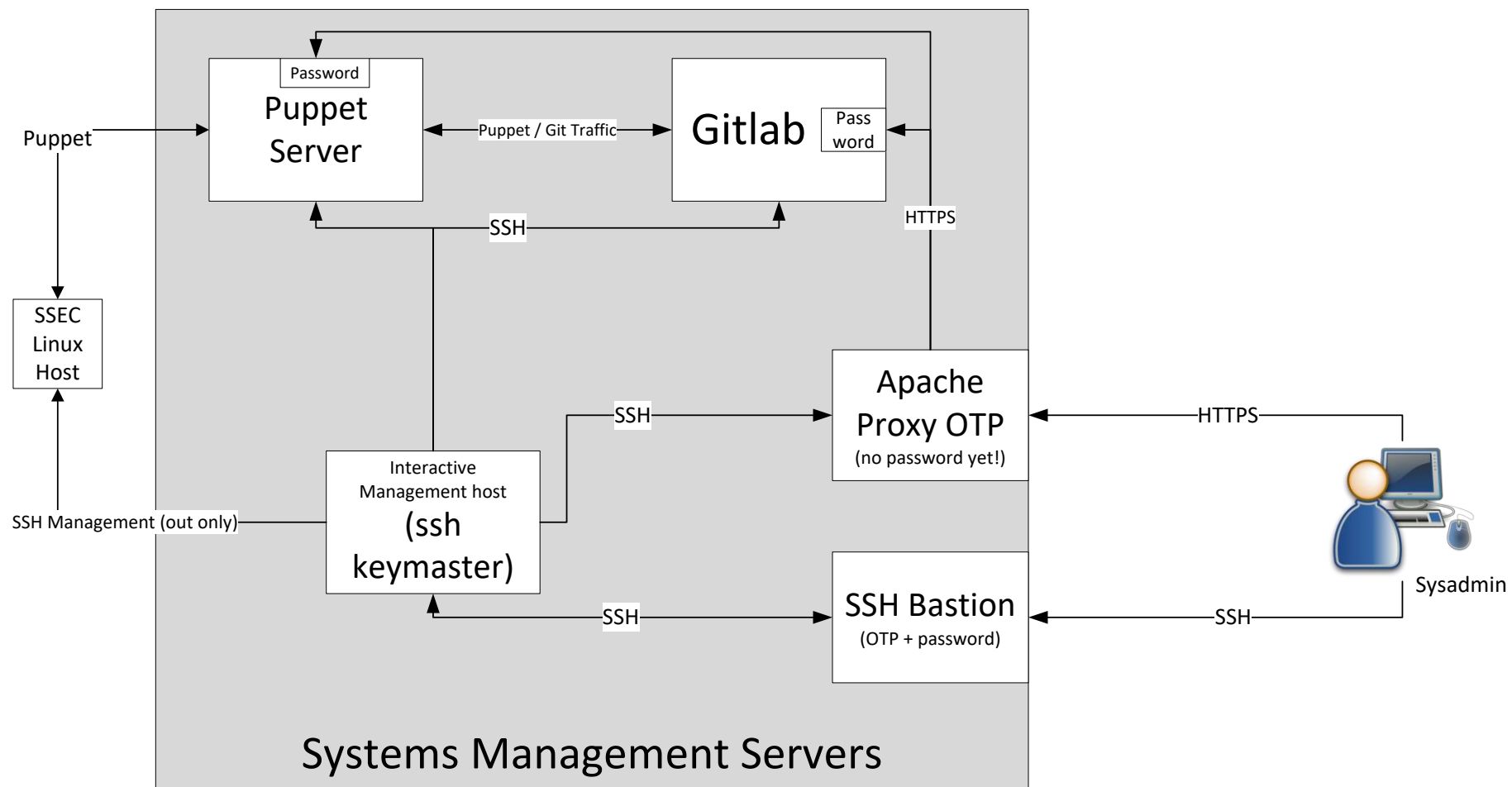
The list of pain is fairly long, and OTP doesn't help them all. But it helps improve many, including:

- get to root ssh key on our administrative bastion host
- Our unix admins group – we have sudo everywhere
 - Put yourself in the unixadmins group on AD, access everywhere
 - get a unixadmin user's password
- Puppet server - can execute everywhere (for example, make a custom fact!)
- RPM repositories - add a bad package and you're in. We re-host these at SSEC
- Gitlab – We use Gitlab for our puppet modules, compromise that.
- Virtual machines – most or all of the servers we use for sysadmin work are virtual. Compromise the hypervisor, and own the VM's. (If you're afraid of a VM, fear the hypervisor too.)

Using OTP to make it better

- Identify all of those critical systems *and* hypervisors they run on.
- Where needed, stand up separate resources – we were using the general SSEC Gitlab server, needed our own.
- Apply local firewall rules to all hosts to segregate, allow access through a bastion host.
 - A separate network would also work
- **Use multifactor authentication (AD password and OTP) on the bastion.**
 - Balance the benefit vs inconvenience
- Add OTP to **sudoers** for all hosts

Linux Admin Systems - Access Diagram



Example Machine Policies

Role	sshd_config Root Login	sshd_config SSH Keys	Iptables allow	OTP
Bastion	Only from ssh keymaster	Only from keymaster	Ssh: anywhere	All accounts, ssh and sudo
Interactive Mgmt “keymaster”	keymaster, bastion	Any (iptables controls)	Ssh: bastion, keymaster	none
Puppet Server	Any (iptables restricts all ssh)	keymaster	Ssh: bastion, keymaster https: web proxy Puppet (8140): All	none
Apache Proxy	keymaster	Keymaster	Ssh: bastion, keymaster	LinOTP apache module
Random Linux Host	Usually keymaster (but sometimes none!)	Usually Keymaster, and certain groups or users	Varies	Sudo (sysadmins, maybe for all, maybe others)

Authentication Server: LinOTP

- For SSEC I looked for various products, and LinOTP met our requirements. It worked well in testing, and now is in production.
 - This manages the private keys and authenticates them with OTP
 - LDAP (Active Directory) for self-service portal.
 - Self-service portal is where users set up keys – google QR codes and manual entry
 - It supports many types of token, not just google authenticator and YubiKey
 - Configuration and install was difficult on Redhat/Centos 6, considerably improved for 7.
- If you decide to use LinOTP and want to compare setup and usage notes, feel free to contact me. There are too many details to include in this talk.



SSEC - OTP Tokens We Use

- Google Authenticator – TOTP mode. Other applications should work
- Yubikey – reprogrammed to not use Yubico servers, but local via OATH mode. They are programmed as in the single server example.

You can configure both, so if you forget your yubikey it's less of a hassle. Google Authenticator is free, and that can be important. If you have to use your token repeatedly I find the yubikey is much more convenient.



LinOTP Policies

- The Self Service option was confusing to configure, here's what I use now:
 - Scope: self_service
 - Action: webprovisionGOOGLEtime, assign, disable, enable, enrollHMAC, resync,
- Number of Tokens –
 - If allowing self-service this is critical to set. Consider the scenario where my password is compromised. **Evildoer** then logs into self-service, registers their own token with LinOTP and they're in!
 - Action: maxtoken=1
 - User: SSEC: (our ldap resolver for all ssec users)
 - This means by default, users get only one token. Admins can then go and set more on specific users as needed. So admins do have to 'set the table' for users, but not actually enroll tokens. How you do this is up to your workflow and needs.

LinOTP Client: PAM module

- For pam we used the C Pam module: **pam_linotp.so**
 - That must be compiled, but it was not difficult
- There is also a python based module: `pam_linotp.py`
 - Have not used it

LinOTP Client Configuration

- The same SSH caveats from the single server example apply – you likely want to limit pubkey authentication.

- As before, configure `/etc/ssh/sshd_config` to contain:

```
ChallengeResponseAuthentication yes
UsePAM yes
Restart sshd
```

- PAM entries are similar. For example, for sudo, but only users in the “2fa_sudo” group:

```
auth    [success=1 default=ignore]      pam_succeed_if.so      user    notingroup 2fa_sudo
auth    required                        /usr/local/lib/security/pam_linotp.so url=https://mylinotpserver.wisc.edu/validate/simplecheck
```

- Useful flags:
 - debug – be warned, it shows OTP keys in `/var/log/secure`
 - For self-signed keys – `nosslcertverify nosslhostneverify`

LinOTP Apache Module

- Clone from https://github.com/LinOTP/mod_auth_linotp
- This requires apxs and libcrypto. For my build host I had to:
 - yum install httpd-devel
 - yum install openssl-devel
- Installs in /usr/lib64/httpd/modules/mod_auth_linotp.so
- We then use apache reverse-proxy to make this a generic way to add OTP.
 - You get 2 logins, it's not beautiful, but better.

My SSEC Wish List

- Visualization of iptables, sshd_config, and pam settings
 - Auditing our setup is not simple, this increases risk
- Radius to protect other services. This seems well supported for freeradius in LinOTP
- Windows
 - configure Windows NPS to query the Radius server?
- Unified login page for web instead of double login proxy
 - But would need customized for each web application I assume

Conclusion

- OTP for Linux systems is possible, used in production, and helps reduce risks from password compromises.
- Focus on incremental improvements – make it **BETTER**
- Never forget the human part of the system, this is very much about interactive logins
- It's not just about the OTP server.
 - Yes, an OTP server / service is important, but we must also think about things like applications (sshd, apache, console), authentication services (PAM), firewalls, and more.

Discussion