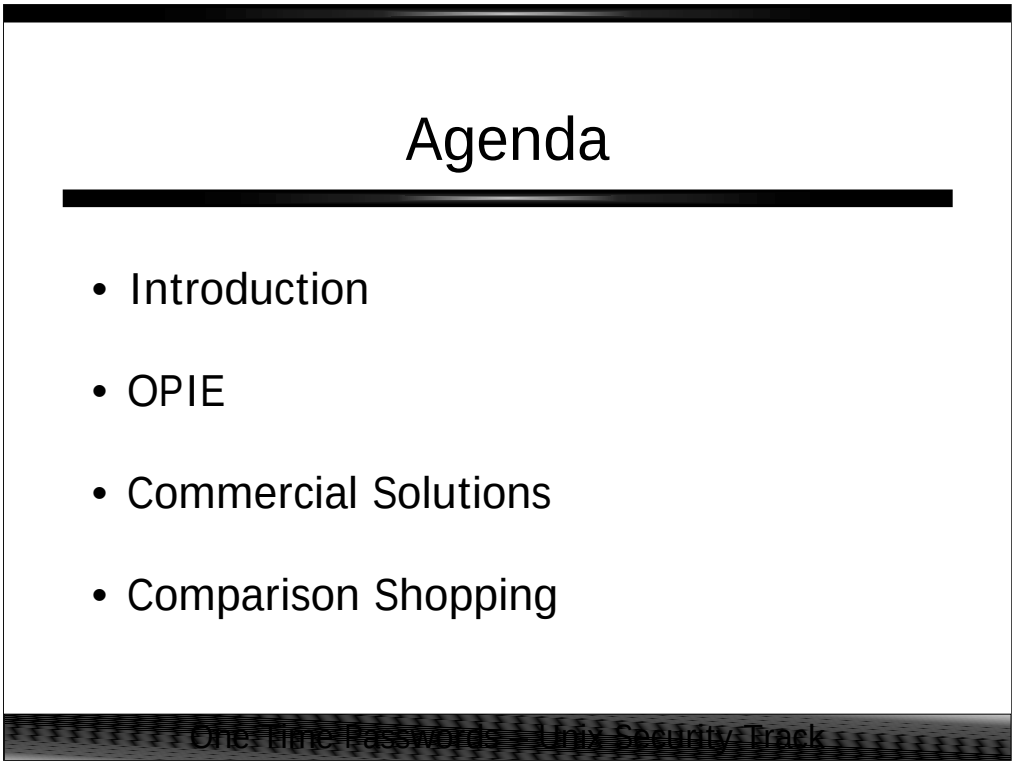
One-Time Passwords

Hal Pomeranz
Deer Run Associates

Hal Pomeranz * Founder/CEO * *hal@deer-run.com*
Deer Run Associates * PO Box 20370 * Oakland, CA 94620-0370
+1 510-339-7740 (voice) * +1 510-339-3941 (fax)
http://www.deer-run.com/

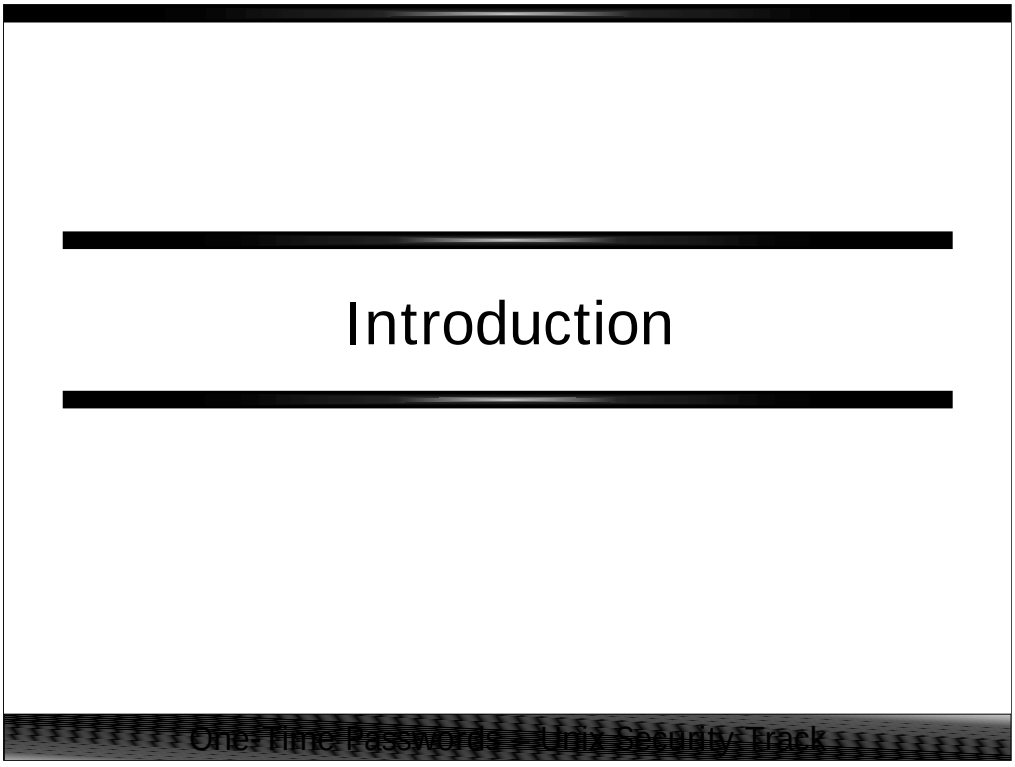


Introduction explains the need for one-time passwords and covers important terms and concepts.

OPIE covers how to install and configure the freely-available OPIE one-time password system. This is designed to give a feel for how one-time password systems look under normal operations.

Commercial Solutions describes how proprietary one-time password solutions differ from the OPIE system and describes issues particular to proprietary implementations.

Comparison Shopping is a brief overview of issues to consider when choosing a one-time password system for your organization.



Introduction explains the need for one-time passwords and covers important terms and concepts.

Problems With Unix Passwords

- Standard Unix passwords deficient:
  - can be captured by a packet sniffer or "shoulder surfing" and replayed
  - crypted passwords often available in `/etc/passwd` or `/etc/shadow`
  - 8 character maximum length is too limited given existing technology

Every time you log in over the network using `telnet`/`rlogin`/etc., the characters that make up your password are being sent over a broadcast medium in clear text. Anybody on any network along the route that your packets travel may intercept this information and steal your password. Also, your fingers are visible as you type in the password on the keyboard. If you use X Windows then somebody could have theoretically taken over your display and be grabbing all of your keystrokes. In any event, since Unix requires you to use the same password over and over again, you are vulnerable.

Many BSD-derived systems keep the encrypted version of your password in a world-readable file. With sufficient computational resources (made simpler by the eight character limit on Unix passwords), a brute force attack can quickly provide your password to a system cracker. Since the vast majority of users choose terrible passwords and don't change them frequently, the task of our theoretical cracker is made simpler.

One-Time Passwords

- A *One Time Password* (OTP) system guarantees a new password on every connection:
  - User establishes secret on remote server
  - User has software or hardware "calculator"
  - At login, server sends challenge string
  - Response calculated w/ challenge+secret
  - Result (password) copied into login window
  - Server verifies response

In a OTP system, each user still has a re-usable secret password. However, this password is used in combination with a special calculator (either a hardware device or a software program) that implements a (sometimes proprietary) cryptographic hash function.

When a user connects to a remote server, the server sends a challenge string to the user instead of the normal `Password:` prompt. The user copies this challenge into their calculator and then types in their secret password. The calculator hashes this information together and computes a response. The user types (or cut'n'pastes) the response back to the remote machine. The remote machine knows the user's secret, the challenge it issued, and the hash function, so it can also compute what the user's response should be– thus it compares its result with what the user sent back to verify the user.

Since the challenge string varies each time, the same OTP response will never be valid twice. Even if the OTP response were stolen in transit, it does no good to an attacker.

OTPs Are Good, But...

- Attacker can see both challenge *and* response– brute force attack is possible
- Secret can also be discovered easily by "shoulder surfing"

We need a way to prevent access even if the attacker knows our secret...

However, since an attacker listening on the wire sees both the challenge string and the encrypted hash the user sends back, they can try combining the challenge with random password strings until they get an encrypted hash which matches the one the user transmitted. At this point, they will know the user's secret.

The good news is that OTP secrets are typically longer than 8 characters and the hash algorithms used by OTP systems are computationally more difficult than the standard Unix password hash, so it takes a lot longer to brute-force OTP secrets. OTP algorithms also generally force users to change secrets regularly.

Two-Factor Authentication

- Increases security by combining
  - "Something you know" (your secret)
  - "Something you have" (physical device)
- Physical device is encoded with a unique identifying key or algorithm
- Most commercial OTP systems employ two-factor authentication

Two-factor authentication means that the user must present two proofs that they are who they claim to be. In commercial OTP systems the two factors are usually:

- A serialized or uniquely keyed hardware device
- A secret password or PIN that the user has memorized

Even if an attacker manages to somehow capture a user's password, they still have to steal the user's personal token in order to successfully impersonate that user. Similarly, if the user loses their hardware token, it's useless to an attacker that doesn't know the user's password. User's must be trained to report lost or stolen tokens immediately, however. The good news is that since users will be unable to log in without their token, reporting tends to be very good.

Types of Two-Factor Devices

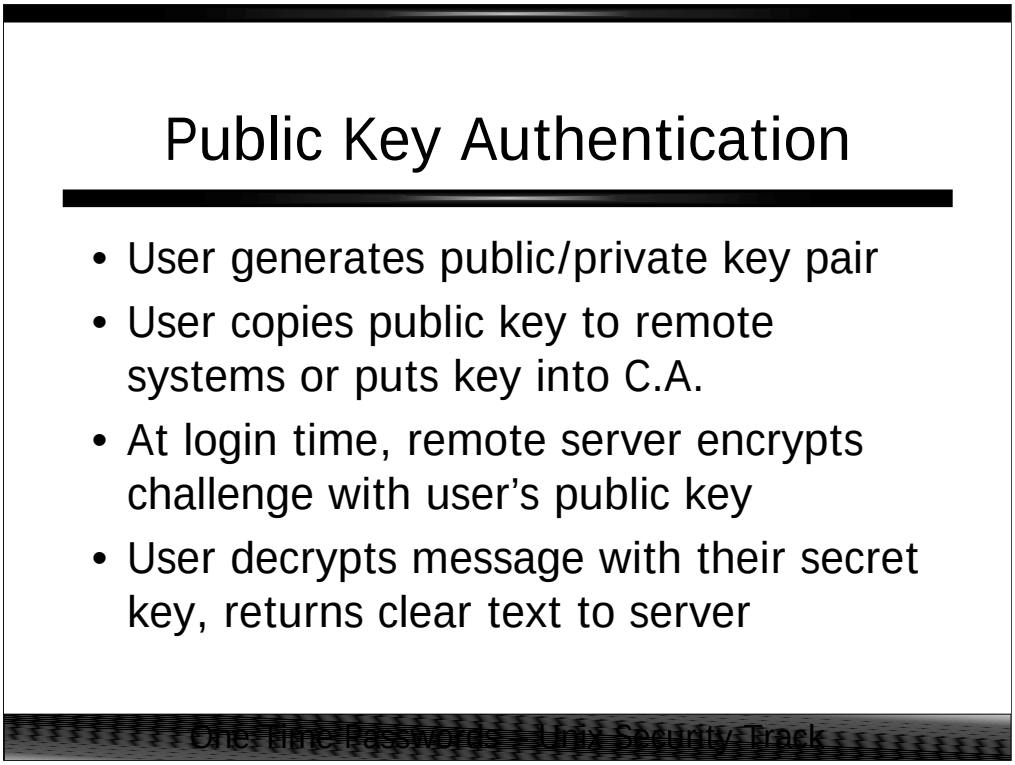
- *Challenge/Response*
  - Device is used to calculate responses
  - User's secret entered to compute response
- *Synchronous*
  - Continuously produces new passwords
  - User's secret "unlocks" token or is combined with password from token

Generally speaking there are two types of hardware tokens supplied by the various commercial OTP system vendors.

Challenge/response tokens function just like the standard OTP system described earlier, except that the unique identifier or key encoded in the token is used to compute the correct responses. Thus only the somebody who knows the user's secret and has the token with the correct unique identifier associated with that user can compute the correct responses.

Synchronous tokens continuously generate new, unique passwords (typically every 30-60 seconds). The token and the remote machine that the user is attempting to access are kept in synch through some proprietary mechanism (usually involving a trusted third-party "security server" machine). When the user wishes to log in, they must provide the "current" password from their token. The user also has a secret which is also known to the remote machine– sometimes the secret is used to "unlock" the hardware token or permute the "current" password value from the token, and sometimes the secret is merely entered by the user along with the password from the token. Obviously, if the user types their secret over the network then it is potentially vulnerable to sniffer attacks, but the attacker would still have to steal the user's token.

Users generally hate challenge/response tokens (too much typing) and prefer synchronous tokens.



Public-key cryptography can be used to do OTP authentication, and this has been generating a lot of consumer and vendor interest over the last several years.

Each user generates a public/private key pair (messages encrypted with the public key can only be decrypted with the private key and vice versa). When the user wishes to access a machine, that machine must be able to somehow determine the user's public key– usually this is accomplished by the user simply copying their public key to a file on the remote machine, but could theoretically be accomplished using a trusted third-party certificate authority.

In any event, when the user attempts to log in, the remote machine encrypts a message with the user's public key. The user receives the encrypted message and uses their private key (known only to them) to decrypt the message. When the remote machine receives the clear text response, it knows the user on the other end of the connection possesses the correct secret key and therefore assumes the user is who they claim to be.

RSAAuthentication in SSH is an example of public-key-based authentication.

How to Store Secret Key?

- File on disk
  - Must be encrypted to protect secret
  - Can still be stolen and cracked off-line
  - Not very portable
- Smart cards
  - Portable and tamper-resistant
  - Need to deploy readers everywhere

The problem with public-key-based authentication is that it is only as secure as the user's private key– anybody who steals that key can impersonate the user until that user "revokes" their old key pair and establishes a new one (potentially a very laborious process).

One option is to encrypt the secret key and store it in a file on disk. Of course the encrypted file could be stolen and cracked offline. Also, the file doesn't do the user much good if the user is traveling and can't access the machine that their private key is stored on. Some users put their private keys on floppy disks (or other removable media) for this reason. However, be aware that DOS formatted floppies have no access control protection, so as soon as that floppy is mounted, any user on the machine can steal the user's (encrypted) secret key file– many users format their floppies as UFS file systems for this reason (though this means they can't be read on Windows machines).

Smart cards are a portable way of storing and retrieving keys. Generally smart cards are tamper resistant– some even going so far as to destroy themselves if somebody appears to be accessing them incorrectly. The major difficulty with smart cards is that they require special card reader hardware to be deployed every place your users *might* want to log in from (i.e., every computer in the world)– this tends to limit the likelihood of wide-scale smart card deployment in the near future.

Available OTP Solutions

- Freely available (software only):
  S/Key (`ftp://thumper.bellcore.com/pub/nmh`)
  OPIE (`http://www.inner.net/opie`)
- Commercial Solutions:
  SecurID (RSA) CryptoCard (CryptoCard)
  Defender (Axent) ActivCard (ActivCard)
  SafeWord (Secure Computing Corporation)

The first freely available OTP scheme was S/Key, developed at Bellcore. Bellcore started work on a commercial version of S/Key and trademarked the name, and so freeware OTP development is continued under the name OPIE (One-time Passwords In Everything).

There are several other vendors besides the five major players listed above. Most vendors offer both hardware and software token solutions, and are working on support for “electronic-wallet” type smart card systems. Note that there has been a great deal of consolidation in the security industry, so Secure Computing Corp is marketing technology originally developed by Enigma Logic, and Axent bought the SNK technology from the company which bought it from Digital Pathways. RSA bought Security Dynamics to get the SecurID technology.

Issue #1: Integration Nightmare

- Most OTP systems include support for `login`, `ftp`, and `su`
- Many other apps and services that *you* get to integrate with your OTP system:
  - `rlogin`, SSH
  - XDM, `xlock`, POP/IMAP server
  - Windows auth services, Kerberos?
  - Apache? ERP Apps? ...?
- Huge (ongoing) devel/admin effort...

Whether you go with a freely available solution or a commercial product, you end up having to replace any software which does regular password authentication with new software which supports your token solution. This turns out to be a huge number of applications from a wide variety of different sources (proprietary, Open Source, and in-house development). Your OTP solution will typically come pre-packaged with support for certain basic apps plus a set of programming libraries so you can replace the standard authentication routines in the program with special OTP routines– this is typically a non-trivial operation (particularly for challenge/response tokens). Then you get to maintain all of this locally modified software across future revisions of the software itself, the OTP vendor's software, and the underlying operating system.

Note that most commercial products developed as dial-up security solutions for terminal servers and other network devices-- not all of them support host authentication software in the first place!

Note also that SecurID has significant market dominance. The upside of this is that if software you're buying supports any OTP system at all, it's going to support SecurID first (and then *maybe* somebody else). The downside is that SecurID is 4x-5x more expensive than their competitors and SecurID tokens are the only tokens on the market which expire (after 2 years or 4 years, depending on which you buy).

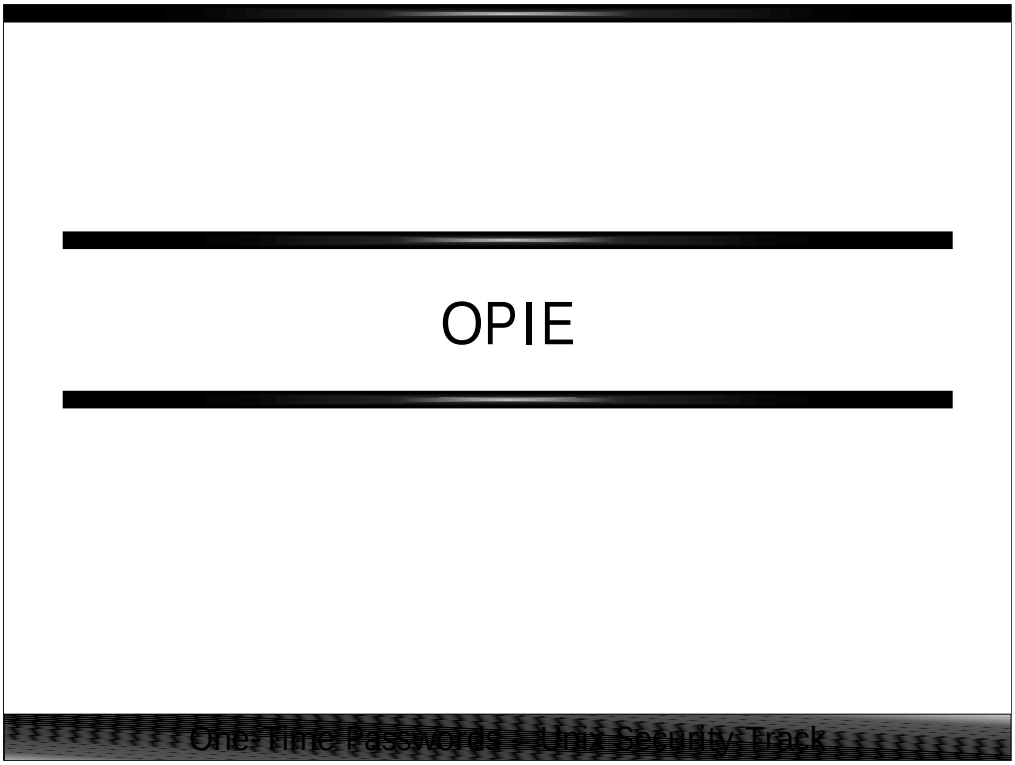
Issue #2: User Resistance

- The average user finds OTP systems difficult (even synchronous tokens)
- Lots of hand-holding required
- Many sites choose to deploy OTP at the perimeter (dial and VPN access) only

One-time passwords are just "different enough" that most users have difficulty accepting them. Also, the use of one-time passwords generally precludes the use of automatic login scripts– security analysts often see this as a feature but users typically disagree.

In any event, many sites often deploy OTP at the "edges" of their network– typically only on their dial-in pools and Internet/VPN access points. This seems to be a reasonable compromise, but can present problems when you are trying to grant a business partner or vendor temporary access to your network. Do you give that vender their own OTP login (and token?) with a limited lifetime, or do you force that vendor to contact one of your employees who will help that vendor get logged in through the employee's account (typically by the employee telling the vendor the correct OTP over the phone at login time)?

One of the advantages to public-key based systems is that they can be integrated in a fairly transparent fashion into most applications (c.f. RSAAuthentication in SSH). Users only have to remember the reusable password that unlocks their secret key and all of the challenge/response communication is handled by the application.



OPIE covers how to install and configure the freely-available OPIE one-time password system. This is designed to give a feel for how one-time password systems look under normal operations.

S/Key vs. OPIE...

- S/Key (developed at Bellcore) was originally Open Source
- In 1994 Bellcore decides to produce a commercial product based on S/Key
- Open Source development continues at NRL under the name “OPIE”
- Craig Metz leaves NRL, OPIE development stalls in 1998
- OpenBSD Project produces highly-modified version of S/Key for their OS, 1999

In the early 1990s, the free S/Key OTP implementation (based on the MD4 cryptographic hash algorithm) was created by researchers at Bellcore. It was fairly widely adopted, but in 1994 Bellcore decided to try creating a commercial product based on S/Key and free development ceased.

Since the Navy Research Labs were using S/Key heavily at the time, their developers continued work on S/Key but had to change the name to OPIE (One-time Passwords in Everything) to avoid infringing on Bellcore's trademark. One of the major improvements in OPIE was the inclusion of support for the MD5 hash algorithm (MD4 support was maintained for backwards compatibility). However, by the late 1990s all of the principal OPIE developers were no longer doing active development and things have basically stalled.

Recently, the OpenBSD developers have taken the sources from the last free Bellcore release and made substantial changes– including simultaneous support for the MD4, MD5, SHA1, and RMD-160 hash algorithms. Unfortunately, this software is only available in OpenBSD, though there is a port to other platforms underway at `http://www.sparc.spb.su/solaris/skey/`

... and the IETF

- S/Key documented as informational RFC 1760 (1995)
- Standards-track OTP system based on S/Key introduced as RFC 1938 (1996)
- RFC 2289 obsoletes RFC 1938 (1998)

We still have no widely-adopted, portable Open Source OTP system!

Throughout this period, an OTP system based on S/Key has been moving through the standards track process. Other developers have produced interoperable reference implementations in order to meet the IETF guidelines, but none of them have been widely embraced or adopted.

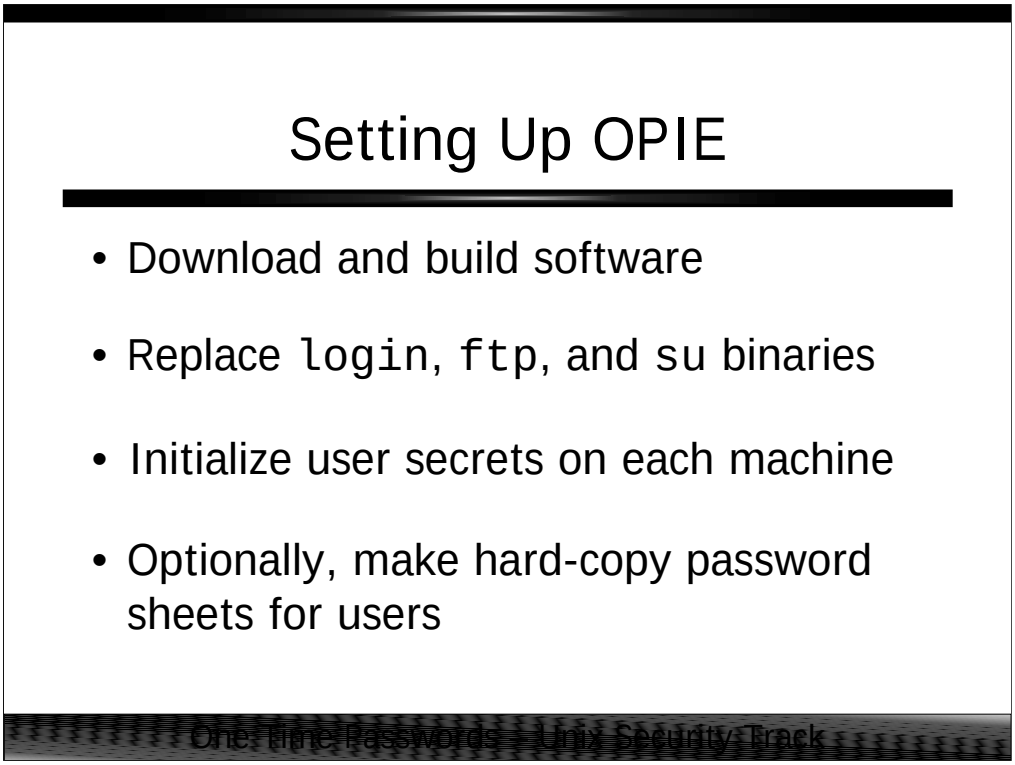
The upshot is that after nearly 10 years of effort, there isn't a single freely available OTP system that has been widely ported or adopted (even by applications in the Open Source community).

How It Works

- Per machine `opiekeys` file stores user secrets, random seed, counter
- At login, user is challenged with counter+random seed values
- User enters challenge and their secret password into OPIE calculator
- User cut'n'pastes response into login window
- Counter in `opiekeys` file decremented

The OPIE system utilizes a per-machine `/etc/opiekeys` file which stores an encrypted form of the user's secret, plus a random seed value and a decrementing counter value (usually initialized to 100 or 500). Each time the user logs in, the machine presents a challenge string containing the current counter value plus the random seed. This challenge is copied by the user into a software calculator (calculators for Win32, Mac, and other platforms are available at the OPIE Web site) and the user then enters their OPIE secret into the calculator. The result is copied back into the login window. The remote machine decrements the user's counter and gives up a login shell.

One big problem is that the user must not allow their counter to drop to zero or they'll be unable to log in to change their secret and re-initialize the counter. Also, there are obvious issues with having to maintain distinct `opiekeys` files on all of your machines. Users also tend to use the same password across all of their accounts, so the user's OPIE secret is sometimes easy to compromise.



To use OPIE at your site, you must first download the source code and build binaries for every platform in use at your site. This can be a fair amount of effort for very heterogeneous sites. The binaries then have to be copied to all systems, overwriting the vendor supplied binaries. Remember that vendor patches may overwrite the OPIE binaries, so take care when applying vendor patch clusters.

As soon as the new binaries are installed (or perhaps before), the administrator needs to set up initial OPIE secrets for all users who need to log into the machine. As a backwards-compatibility mechanism, OPIE can be built to read an `/etc/opieaccess` file which lists those users who must use OPIE (anybody not listed in the file uses normal passwords). This is a transition mechanism only and something of security hole and is not meant as a long-term solution.

Since OPIE secrets can be generated in advance, some sites choose to print out hard-copy lists of OPIE responses for their users. This means the users don't have to mess around with an OPIE calculator. Using a small font (6 point fonts are about the smallest that are still readable), it's possible to get about 100 responses on a sheet of paper that can be folded in half and stuffed into a wallet. If this paper is lost or stolen, however, the user needs to report in and have their secret changed (and a new password sheet FedExed or faxed over to them).

OPIE Installation -- Files

- On server
  `opielogin`, `opieftpd`, `opiesu`
  `opiekeys`
  `/etc/opiekeys`

- On local client
  `opiekey`

OPIE uses the GNU `autconf` mechanism, so building the package on most systems is easy.

The `opielogin`, `opieftpd`, and `opiesu` programs are drop-in replacements for their standard OS counterparts (even going so far as to implement certain vendor-specific hacks, lick support for `/etc/default/login` under Solaris). `opielogin` is smart enough to know if you are logging in on the system console and accept a standard Unix re-usable password instead of an OPIE response.

`/etc/opiekeys` is the OPIE equivalent of the `/etc/shadow` (**not** `/etc/passwd`) file and the `opiekeys` program is what users run to update their entries in the `opiekeys` file. `opiekeys` has a mechanism to allow users to securely set passwords even over a public network connection.

The `opiekey` program takes and OPIE challenge on the command line, prompts for a secret word, and returns the appropriate OPIE response. `opiekey` will not allow itself to be executed if the secret word will have to travel over a network connection.

As mentioned earlier Mac, Windows, and PalmOS OPIE calculators are also available from the OPIE Web site.

OPIE Install -- Server

```
# ls
opieftpd    opielogin   opiekeys  opiesu
# mv -f opielogin /bin/login
# chmod 4111 /bin/login
# cp opiesu /sbin/su
# mv opiesu /bin/su
# chmod 4111 /sbin/su /bin/su
# mv opieftpd /usr/sbin/in.ftpd
# chmod 100 /usr/sbin/in.ftpd
# chown root /bin/login /bin/su /sbin/su
   /usr/sbin/in.ftpd
# chgrp sys /bin/login /bin/su /sbin/su
   /usr/sbin/in.ftpd
#
```


You should know that if you `make install` from the OPIE source directory the install process will automatically overwrite the vendor-supplied OS binaries (though it makes a copy as `<binary>.opie.old`). Installing manually is generally safer.

Before executing any of the above commands, make sure you have a **root shell** on the console of the system you are modifying. Don't give up this root shell until you are able to verify that OPIE is working properly or you could end up locking yourself out of the system.

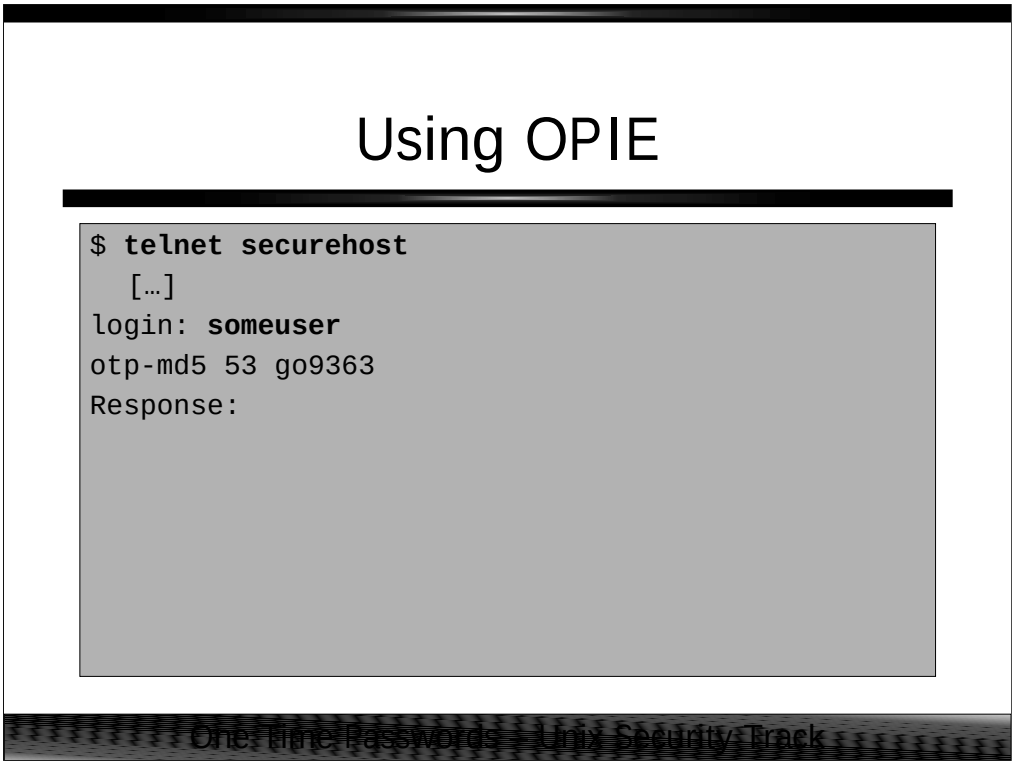
Since we are overwriting the standard OS versions of the above commands, you may first wish to make a copy of the vendor-provided versions before proceeding further. If everything goes well, make sure you **delete** the vendor versions.

OPIE Install -- Server (cont.)

```
# touch /etc/opiekeys
# chmod 600 /etc/opiekeys
# mv opiekeys /bin
# chmod 4111 /bin/opiekeys
# chown root /etc/opiekeys /bin/opiekeys
# chgrp sys /etc/opiekeys /bin/opiekeys
# /bin/opiekeys -c -n 99 root
Adding root:
   […]
Enter new secret pass phrase: (not echoed)
Again new secret pass phrase: (not echoed)
   […]
#
```

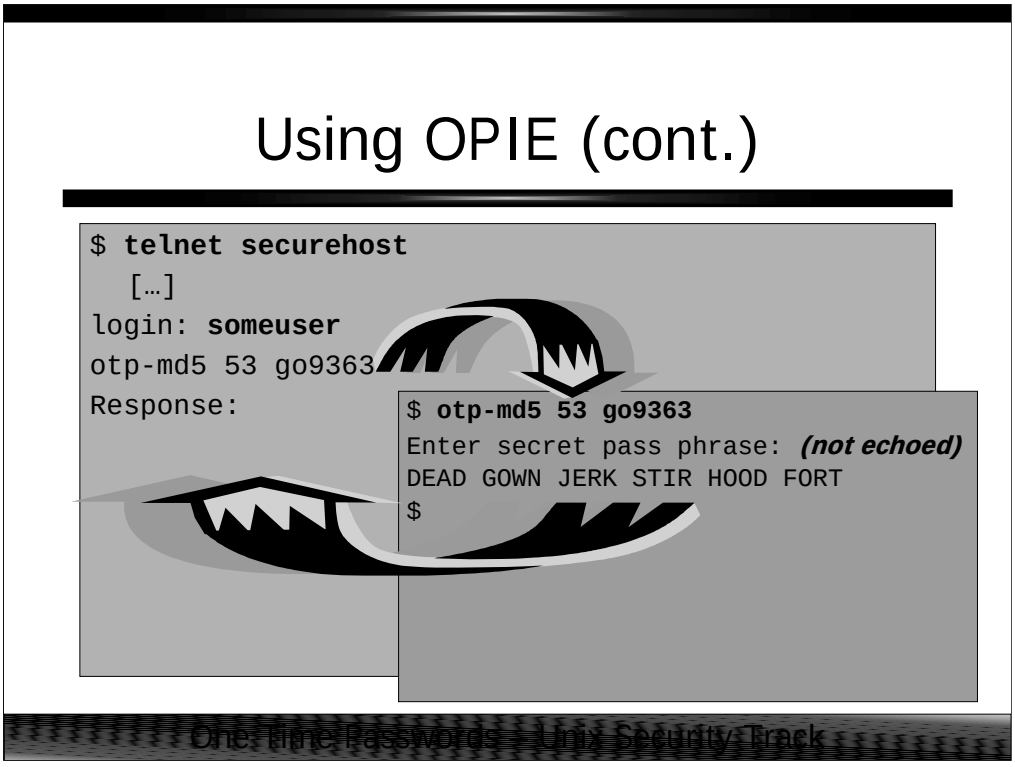

Note that we are making `/etc/opiekeys` mode 600 since it contains encrypted OPIE secrets. Again, think of `/etc/opiekeys` as you would `/etc/shadow`.

Having created the `opiekeys` file, we add an entry for root: this will be used by `opiesu` only, since you shouldn't be allowing root logins. The `-c` option means that we are on the system console and it is safe to type the secret word in directly (`opiekeys` verifies this before allowing you to proceed); the `-n` option specifies how many logins are allowed before the secret needs to be reset. OPIE does allow you to re-use the same secret from one time to the next when you reinitialize using `opiekeys`.



Now let's go through an example of logging in with OPIE to get a feel for how the system works. For purposes of this example we assume an account and OPIE secret have been created for account `someuser`.

The first step is to establish a `telnet` or other network connection to the remote server. After the normal `telnet` dialog and `login:` prompt, the OPIE challenge appears…



The user cut'n'pastes the entire challenge string from the `telnet` window into another window. `otp-md5` is actually a symbolic link to the `opiekey` program and tells OPIE to use the MD5 algorithm to compute the response.

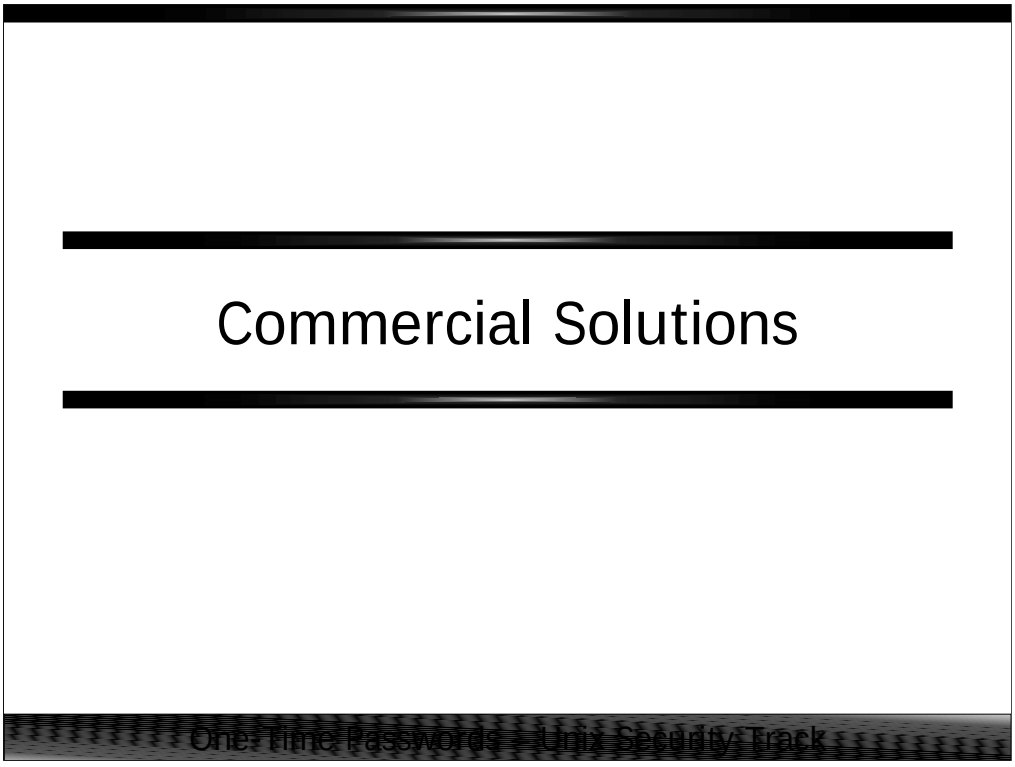
`otp-md5` (aka `opiekey`) prompts the user for their OPIE secret and emits a six word response. These six words actually correspond to a string of hexadecimal digits, but the words are easier to type and remember than a string of random hex codes. The user again cut'n'pastes this response back into the original login window.

One of the significant advantages to the Mac and Windows OPIE calculators is that they automatically manipulate the cut'n'paste buffer for the user, and logins are very easy.

Using OPIE (cont.)

```
$ telnet securehost
  […]
login: someuser
otp-md5 53 go9363
Response: DEAD GOWN JERK STIR HOOD FORT
Last login: Sun May 14 10:18:06 2000 from …
Sun Microsystems Inc.  SunOS 5.6  Generic …
$
```


Assuming the user provides the correct six word response, the normal login process continues and the user ends up with a shell prompt. The `opielogin` program also decrements the user's counter in the `opiekeys` file (and will print a warning message if the user's counter is 5 or less).



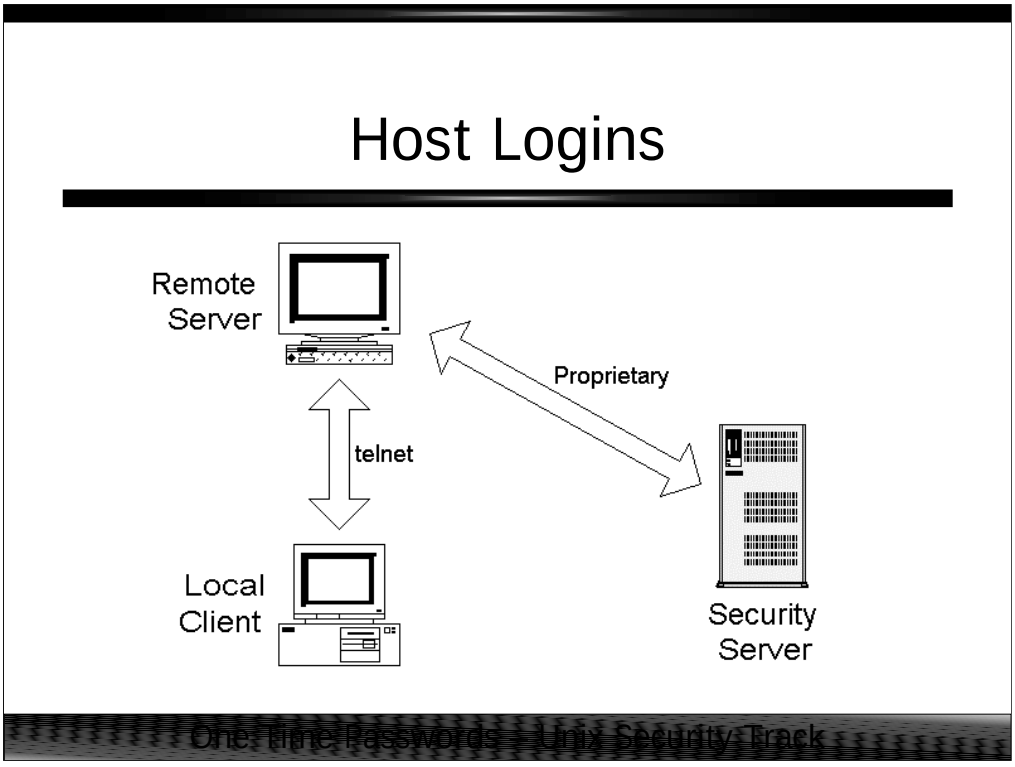
Commercial Solutions describes how proprietary one-time password solutions differ from the OPIE system and describes issues particular to proprietary implementations.

Architecture

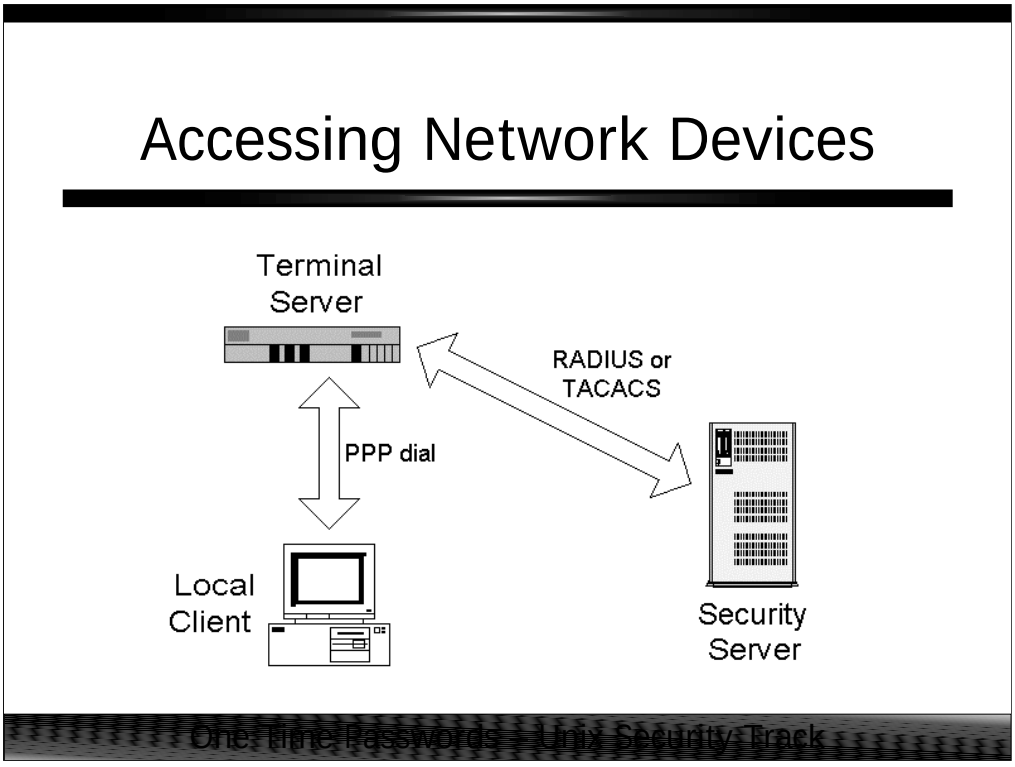
- Commercial solutions generally employ an extra “security server” device
- The user/token database is maintained on the security server
- At login, remote server verifies user’s credentials with security server
- Obviously, security server is a significant potential failure point

All of the commercially available solutions require deployment of a special "security server" machine which holds a database containing user and token information. When a user wants to log into a given machine, that machine must contact the security server to verify the user's credentials.

While this is more convenient from an administrator perspective than maintaining `opiekeys` files on every machine, the security server becomes a significant failure point for your network– if the machine is unavailable then nobody can log in! The commercial solutions all support some sort of fail-over or replication scheme, but with varying degrees of usefulness (more on this in the last section).



When a user wishes to log into a remote machine (Unix, NT, whatever), the remote machine typically speaks a proprietary network protocol to the remote security server in order to verify the user's credentials. This proprietary protocol is typically made available as a set of programming libraries which can be integrated into locally-developed or Open Source applications in use at the site.



Commercial OTP solutions also typically support RADIUS (and sometimes TACACS as well) for doing authentication on terminal servers, routers, and other network devices. One of the difficulties with both RADIUS and TACACS is that there are a lot of different "versions" of each protocol with varying degrees of interoperability, so make sure you test before you buy!

Installing Commercial OTP

- Install security server(s)
  - OS Install
  - OTP server software
- Create user accounts
- Distribute tokens
- Install new binaries (`login`, …)

The first phase of deploying a commercial OTP solution is getting the security servers installed and making sure that replication and fail-over are working properly. Generally speaking, you should spend a great deal of time focusing on the security of the security servers themselves– they are an obvious target for people wanting to attack your network. Be sure to limit access to these machines to only those people who actually have the responsibility for maintaining user accounts in your OTP system!

Once the security servers are installed and working, create user accounts and hand out tokens *before* installing or activating the modified system binaries for the OTP system. However, if you hand out tokens too far in advance of your deployment, then they tend to get lost or forgotten.

Establishing initial user secrets varies from token solution to token solution. Sometimes secrets must be encoded into the token at the time they are distributed, other systems allow the user to choose their secret the first time they log in using the token. The latter is generally preferrable (because the admin can always pre-set the token by logging in as the user one time before handing the token over to the user).

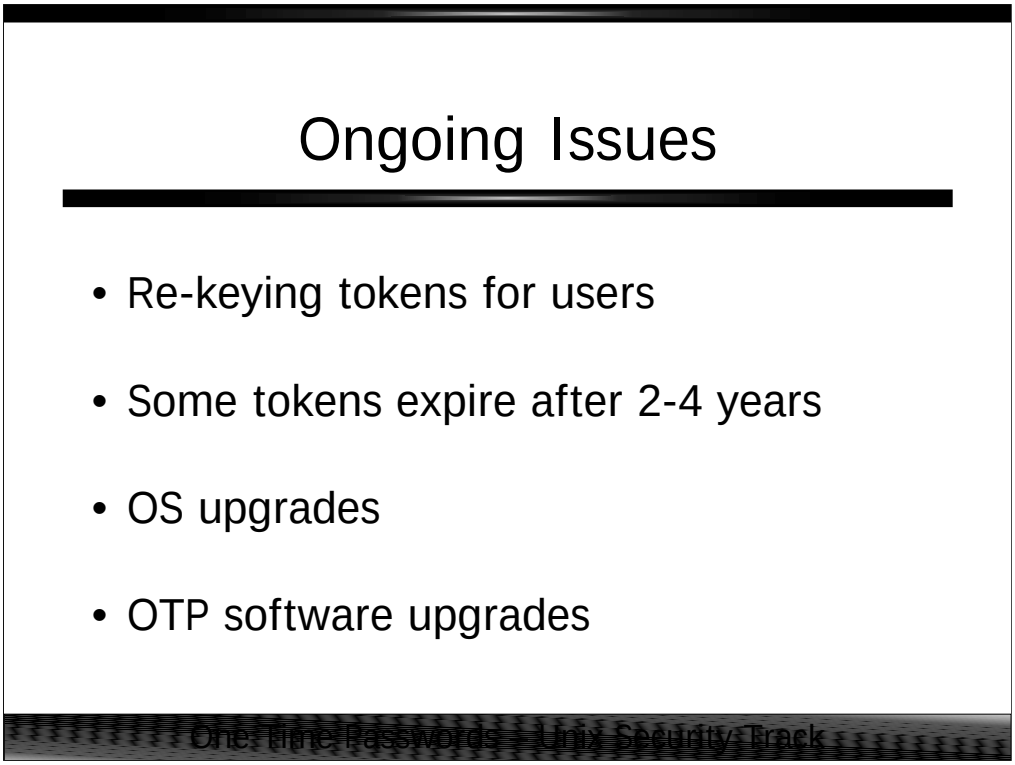
Setting Up Users

- Each user has entry on security server:
  - Username, secret
  - Information about token(s) issued to user
  - What machines can be accessed by user
  - Time of day user may access systems
- User also needs an account in `passwd` file on all systems

User entries on commercial OTP systems are generally reasonably complex. The user has their secret plus one or more tokens which have been assigned to that user (some systems allow multiple tokens, some only one per user). The user may also have a re-usable password which can be used under certain circumstances. There is also a per-user list of machines which that user is allowed to access.

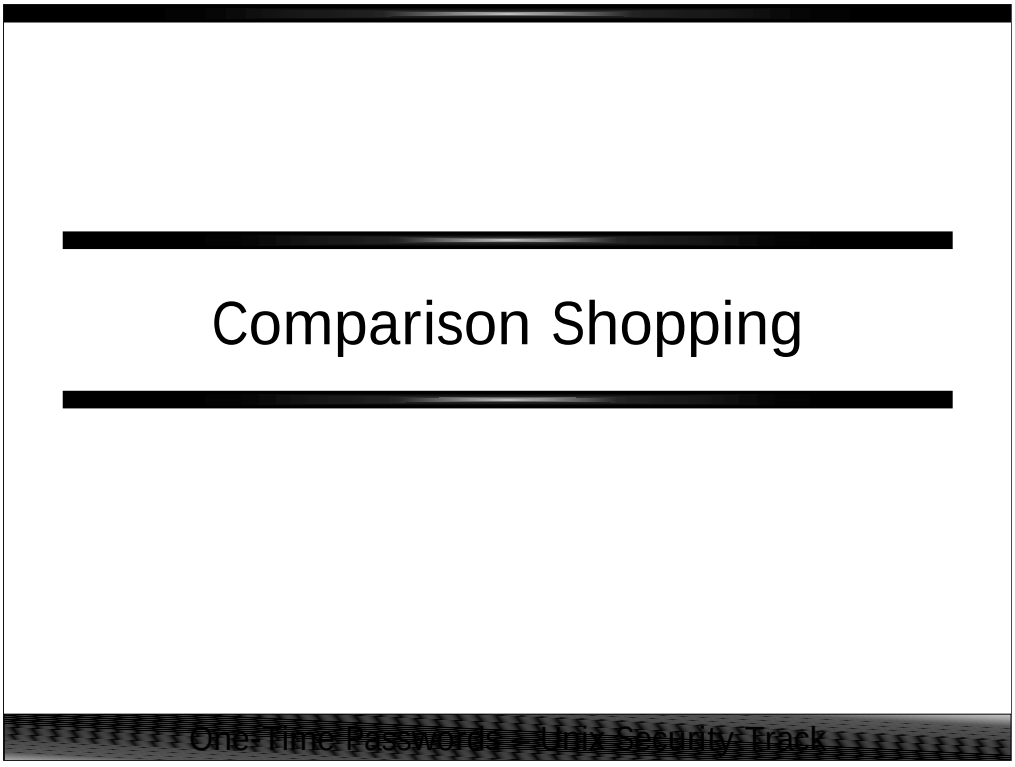
Some systems allow administrators to create "groups"– by assigning users to a group, that user inherits a certain set of default configuration options (typically things like a list of machines which can be accessed, times when access is allowed or not allow, etc.). There may also be per-machine configuration information stored in the database as well.

Users will also need an account in the password file on all machines they will be logging into. Generally this account should have an invalid password string (*but not an empty string!*) in the shadow password file– the user should always use token authentication to access the machine and not a re-usable password.



Commercial OTP solutions have all of the same administrative issues as OPIE and then some. The OTP system must be maintained across future versions of your vendor's OS, application revisions, and upgrades of the OTP system itself. Upgrading the security servers is particularly thorny.

Also, users have a tendency to lose, fold, spindle, mutilate, drown, and otherwise dismember their tokens. Also some tokens have a tendency to simply lose synchronization with the security server and need to be re-issued. SecurID tokens expire and need to be re-issued on two or four year cycles (unlimited tokens are apparently now available)– a significant undertaking in a 10,000 person company! Personal experience seems to indicate that one full-time equivalent administrator is required for every 1000 tokens in the field just for user support and to manage "normal" token re-key activities.



Comparison Shopping is a brief overview of issues to consider when choosing a one-time password system for your organization.

RFP Fodder (#1)

- Authentication DB
  - Usually proprietary, also not relational
  - Consider vendor's replication strategy
  - What about performance?
- Admin Interface
  - GUI? Text? API exists?
  - This is where most of your cost occurs

Commercial OTP solutions generally use a proprietary, non-relational database back-end. This makes it difficult to integrate these databases with other databases in your organization (like your HR database for example). Debugging problems with these databases is also extremely difficult, as is performance tuning. Replication strategies vary *widely* from vendor to vendor– some support two-way mirroring while others only support master/slave relationships (make sure you can easily convert a slave into a master in case your master server explodes), some only support a limited number of backup servers, some only allow one server to be in operation at a time and the backup server(s) are warm standbys only (a significant performance issue).

The administrative interface is also critical. If the admin interface is only GUI based (worse, only Windows GUI based) then remote administration is difficult. Also, GUI-only interfaces mean that you can't automate administrative tasks using shell/Perl scripts. If there's only a text-mode interface then it may be difficult to train junior administrators to use the system.

RPF Fodder (#2)

- Tokens
  - Can you replace the battery?
  - Need to be re-licensed periodically?
  - Durable?
  - Keys easy to use?
  - Display easy to read?
  - Software tokens available? On PDAs?
  - Users can have multiple tokens?

You need to clearly understand what sorts of tokens your vendor supports– challenge/response, synchronous, and/or smart cards. You should also find out what sorts of software tokens are available– including for PDAs commonly in use at your site (Palm Pilots or WinCE machines for example).

Some tokens need to be relicensed periodically which is both expensive and administratively time-consuming. Remember, however, that a token with a non-replaceable battery will ultimately expire too.

From an end-user perspective, the token display should be easy to read (and resist scratches) and the keys easy to use even by people with thick fingers. Also, the token should be durable enough to get dumped into a backpack or purse and carried around with a lot of heavy objects (books) piled on top of it. It should also be able to go into a pants pocket and not get destroyed when the user sits down.

You should also determine whether or not a given user can have multiple tokens in use at the same time (software token on their laptop, hardware token when traveling, smart card in the office perhaps).

RFP Fodder (#3)

- Authentication Support
  - RADIUS and TACACS support
  - Host auth support (`login`, FTP, `su`, ...)
  - Application support (Oracle, Web, ...)
- Security Functionality
  - Can you lock out tokens quickly?
  - Deny access by time of day, app, etc.?
  - Can you control PIN length, etc.?

At a minimum, the OTP solution should support RADIUS for communicating with network devices. On the host authentication side, the vendor should provide replacement `login`, `ftpd`, and `su` programs plus an application library which allows you to hack OTP support into your own local apps. You also need to clearly understand from the vendor what other applications they support and what their policy is for staying up to date with new releases.

On the administrative front, you should be able to quickly lock out a lost or stolen token. There should be functionality for user access control based on time of day, host computer, login method, application, etc. Administrators should also have some control over the length and character set allowed in PINs (numeric only versus alphanumeric, etc.).

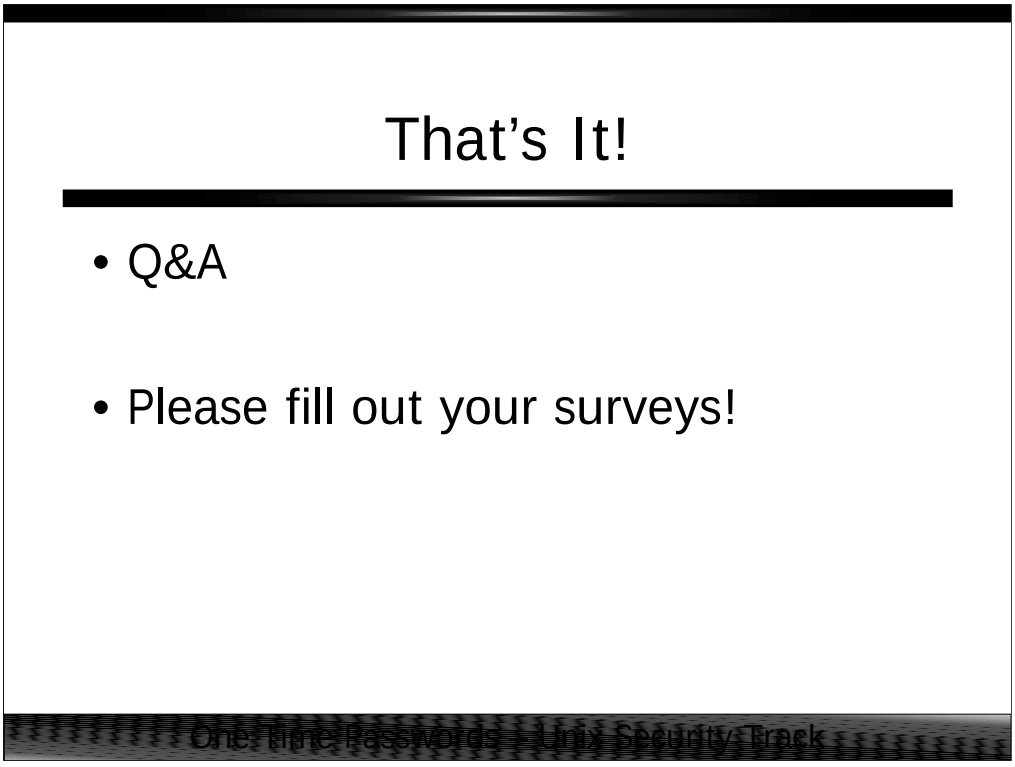
RFP Fodder (#4)

- Vendor Support
  - Check references
  - Set SLAs ahead of time
  - Agree on financial penalties
- Vendor's Plan for the Future...

You *must* get reference sites from your vendor and pay close attention to the reference site(s) experience with vendor support. Remember that if your OTP system is down, then none of your users are going to be able to log in– you need quick turn-around on problems. Try and get your vendor to provide you with a *negative* reference– it can help to understand why people *didn't* choose a particular product (this doesn't necessarily mean you won't choose the product, because your environment or priorities may be different).

Hammer out service level agreements with the vendor *prior* to signing the contract. Financial penalties (rebates on your support contract, etc.) are often a good tool for motivating your vendor to fix your problems quickly.

Also get a sense of where your vendor is headed in terms of technology strategy. Nothing is worse than having to buy a new token system after three years because your vendor is no longer supporting the applications you need!



This space intentionally left blank.