Two-Factor Authentication Basics for Linux

Pat Barron
(pat@lectroid.com)
Western PA Linux Users Group

Some Basic Security Terminology

- Two of the most common things we discuss related to security are
  - Authentication – proving who you are
  - Authorization – deciding what you are allowed to do
- This presentation only discusses mechanisms of authentication. Authorization decisions are made by human beings (i.e., somene taking action to put your userid in the 'sudoers' file, etc.)

Proving your identity to a computer-based service

- This is a challenge, since computers aren't sentient, so they can't get to know you, and recognize you the way other humans can.
- The most traditional way for computer-based services to identify users is via a password.
  - A password is effectively a shared secret between you and the computer, and (in theory) no other party.
  - If you claim to be X, and you demonstrate that you know X's password, then the system concludes that you must, in fact, be X.

Passwords aren't perfect (in some cases, they're not even “good”...)

- Demonstrating that you know X's password doesn't prove that you are X.
- Demonstrating that you know X's password only proves that you are someone who knows X's password....
- Passwords get written down, stolen, intercepted in transit, phished, and all kinds of other bad things.
- If someone gets your password, and that is all that is used to authenticate you, they can masquerade as you as they please, until you change your password.

Improving the situation

- We can improve the reliability of authentication by testing additional “authentication factors”.
- The more points of data we have to make the case for your identity, the more confidence we have in the authentication.
  - Testing only one authentication factor is referred to as “single-factor authentication” - testing more than one before deciding that the subject's identity is sufficiently proved is “multi-factor authentication”.

Multi-Factor Authentication

- An authentication “factor” is some characteristic that we can attempt to test, that is (theoretically) unique to a specific party.
  - Knowledge factors – something you know (password, PIN, security question, etc.)
  - Possession factors – something you have (credit cards, keys, PKI certificates, authentication tokens, etc.)
  - Inheritance factors – something you are (fingerprints, retina pattern, voiceprint, etc.)
- The usefulness of any of these factors for authentication depends on the idea that it would be impractical for anyone other than the individual we wish to identify to successfully pass a test of that factor.
  - We already know this isn't always the case - though some factors are harder to “hijack” than others...

An example of “real world” single-factor authentication.

- Consider the process of opening your locked front door
  - The locked door is intended to keep out anyone who is not allowed in.
  - You give keys to anyone who is allowed in. This is an authorization mechanism.
  - The key is an authentication mechanism – though it doesn't identify an individual, it simply identifies “a person who is allowed in”.
  - As a consequence, anyone who can get hold of a key, by whatever means, can masquerade as “a person who is allowed in” until such time as the lock is changed – the lock doesn't know any better...

A more familiar example

- You use a userid and password to log in to your Linux system.
- You make a long, complicated password, because you're told that is more “secure”. But it's also hard to remember, so you write it down on a notepad next to your computer...
- Anyone who happens to see the notepad can now masquerade as you and access resources on your computer in your name, until such time as you change your password.

A “real world” example of multi-factor authentication

- Withdrawing cash at the ATM
  - You go to the machine
  - You put in your card
  - You punch in your PIN number
  - You withdraw some money
- Note that we need the card, <u>and</u> the PIN. Neither one, by itself, is sufficient.
- We are testing two authentication factors – possession of the card (something you have), and knowledge of the PIN (something you know).
- Anyone who wants to masquerade as you to access your account must get possession of the card, <u>and</u> knowledge of the PIN, or they will not succeed.

Why is multi-factor authentication helpful?

- Testing multiple factors strengthens the case to “believe” the authentication.
  - You know your bank card PIN, but someone may look over your shoulder and watch you key it in.
  - You have your bank card, but if you lose it and someone else finds it, now that person has it instead of you.
  - These are things which are known to happen in the real world – so if someone knows your PIN, or has your card, should someone really believe that they're you?
  - The chance of someone stealing your PIN, <u>and</u> getting possession of your card, is much less likely – still possible, but sufficiently less likely that we are satisfied with this.

Accidentally “breaking” multi-factor authentication

- In systems that use two factors – specifically, “something you know”, and “something you have”, naive users may break the system by storing the two factors together – such as, writing one's bank card PIN on the card itself.
- This effectively changes the system to single-factor – in this example, now physical possession of the bank card is enough to successfully authenticate.
- So, don't do this....

Implementing multi-factor authentication on computer systems

- A common way to implement multi-factor authentication is to add an authentication token (“something you have”) to the traditional password (“something you know”)
  - Hardware tokens – dedicated devices (usually very small) that are used to interact with the authentication process.
    - RSA “SecureID” tokens are a very popular form of this
  - Software tokens – an app that runs either on your own computer, or a handheld computer (e.g., a smartphone) that can interact with the authentication process.

How tokens typically work

- A secret is shared between the token, and the computer system.
- When authenticating to the computer system, the token is used in some way during the authentication process.
- If the token interaction succeeds (i.e., if it seems that you have a token that knows the secret shared with the system), the system assumes that whoever is authenticating must be in possession of the token.

Advantages of hardware tokens

- Typically, they are simpler.
- Typically, they are made to be tamper-resistant – it is very hard to take them apart to extract the shared secret (which would allow you to clone the token). Attempting to disassemble a hardware token typically destroys the secret.

Disadvantages of hardware tokens

- You may not know what is going on inside them, and may not be able to design software to be compatible with them – not necessarily open-source friendly.
  - You may get locked into a single vendor.
- They typically have a limited lifespan, after which they must be replaced.

Advantages of software tokens

- Smartphones are commonplace – why carry an additional device as a token, when you always have your smartphone with you all the time?
- Software token apps are available that conform to open standards.

Disadvantages of software tokens

- If someone else gets possession of your phone, they may be able to extract the shared secret from your software token, and thus clone your token.
  - This is a similar problem to the “writing your PIN on the ATM card” problem, but is mitigated by using the token as a second factor – still using a traditional password as well.

Software authentication tokens on Linux

- We can implement two-factor authentication on Linux by using “oath-toolkit”, which is available on a number of Linux distributions.
- OATH is an open authentication architecture promoted by The Initiative for Open Authentication.
  - Not to be confused with OAuth, which is also an authentication technology, but is different than what we are talking about here.

What oath-toolkit does

- oath-toolkit implements software token support (both client and server) based on the HOTP (HMAC-based One Time Password) and TOTP (Time-based One Time Password) standards, which are published as RFCs.
- The “oathtool” utility is a command-line based testing tool that can also in itself be used as a software token to interact with other systems (though it's very awkward to use it this way).
- The client side of oath-toolkit consists of a PAM module that can be plugged into the usual PAM security infrastructure.

HMAC-based One Time Password (HOTP)

- Generates single-use passwords on demand, based on a secret shared with between the user and the system, and a moving sequence identifier.
- Each time you authenticate, the sequence number changes.
- One problem – if you use the HOTP token to generate a one-time password at a time when you're not actually authenticating to the computer, it can throw HOTP out of sync, and you may have to reset the token and the server's idea of your token's state.

Time-based One Time Password (TOTP)

- This is effectively the same as HOTP, except the sequence identifier used is the current time.
  - This is similar to what SecurID tokens do, where they constantly display a token code that changes periodically, and when you authenticate, you provide whatever token code is displayed at that moment. TOTP works in this way as well.
- This eliminates the problem of the HOTP token getting thrown out of sequence sync.
- However, it introduces a new requirement that the computer system and the software token be kept in close time sync.

What you need to use oath-toolkit

- You need the appropriate Linux packages installed on the system you're using OATH to protect
  - Fedora: “yum install oathtool pam_oath”
  - Debian: “apt-get install oathtool libpam-oath”
- You need to plug the pam_oath.so module into your PAM configuration (see live demo).
- You probably need a software token app for your smartphone (I happen to use an app called “mOTP” on Android, but Red Hat provides a free Android app called “FreeOTP”, as does Google Authenticator, or you can use any other app that supports HOTP and/or TOTP).

Caveats about oath-toolkit to keep in mind

- The biggest problem with oath-toolkit is that it is very poorly documented, particularly the PAM module.
  - There is no man page for the PAM module, the “documentation” exists only in a README file installed under /usr/share/doc/pam_oath on Fedora (may be elsewhere on other distros). The format of the “shared secret” file is documented only in the source code itself – and it's not really “documented”, you just need to read the source and see what it is looking for....
- Beware that you may need to disable SELinux (or switch it to “permissive” mode) in order to make the PAM module work – I had to do this on Fedora 21. Should be fixable for someone with time and energy to analyze the SELinux logs and fix the policy.

Live Demo Now

Backup materials

- Links to all of the materials I referenced will be added to the version of this presentation that is uploaded to the WPLUG wiki.