Authentication on Untrusted Remote Hosts with Public-key Sudo

Matthew Burnside, Mack Lu, and Angelos D. Keromytis – Columbia University

ABSTRACT

Two common tools in Linux- and UNIX-based environments are SSH for secure communications and sudo for performing administrative tasks. These are independent programs with substantially different purposes, but they are often used in conjunction. In this paper, we describe a weakness in their interaction and present our solution, public-key sudo.

Public-key sudo[1] is an extension to the sudo authentication mechanism which allows for public key authentication using the SSH public key framework. We describe our implementation of a BSD SSH authentication module and the SSH modifications required to use this module.

Introduction

In today's age of large, distributed networks, trusted remote machines are rare. That is, untrusted machines are the common case, but through business or other requirements, users and administrators find themselves required to connect to such machines, regardless. These may be machines maintained by disreputable system administrators, machines which are believed to have suffered compromises, or simply machines for which the user suspects there is a high probability of future compromise. It is desirable not to provide sensitive information to such machines.

Two tools which have become the norm in such environments are SSH [15] and sudo [11]. These are indpendent programs with substantially different purposes, but they are often used in conjunction. In this paper, we describe a weakness in their interaction, and then present public-key sudo to solve it.

SSH is a tool used for secure communication between computer systems. It protects the end user by, among other things, providing confidentiality of the user's data on the wire and authentication of the end-host to the user. It also provides a framework for authenticating the user to the end-host. In its default configuration, SSH requires a password on the server, but it can also be configured to use public keys.

To configure SSH for public keys, the user generates one or more key pairs (DSA or RSA) and distributes the public key(s) to the desired servers. The private keys are encrypted with passwords and stored on the local host. During the login process, the SSH client prompts the user for the password to his private key and uses it to decrypt the key. It then uses the decrypted key to generate a signature which is sent to the server. If the server can verify the signature, it allows the user to log in.

SSH provides an additional tool SSH-agent which assists in managing the private keys and further assists in multi-hop sessions. After it is started, the user registers each of his private keys and the SSH-agent takes responsibility for each. The agent prompts the user for the private-key passwords, then loads the keys into protected memory. The agent also creates a UNIX-domain socket on the local host, and stores its location in a well-known environment variable. The socket can be thought of as a tunnel directly to the agent, and all communication with the agent is through this socket. The SSH client then uses the tunnel to query the agent for authentication material during the login process.

SSH also provides an option to forward this socket to subsequent hosts. This allows SSH connections started multiple SSH hops away from the client's local host to connect back to the SSH-agent on the local host and authenticate using the material it manages.

One of the great strengths of SSH with public-key authentication is that a user can log in to an untrusted host without providing any sensitive data. The user provides only his public key and a signature. Even if the remote host is compromised, the user's authentication material is safe. Compare this to password authentication; if the remote SSH daemon has been compromised, an adversary can obtain the plaintext password. Any tool on the remote host which uses password authentication is susceptible to such an attack. By default, sudo is one of those tools.

The UNIX sudo command is designed to allow users to run commands as other users. It has a rich configuration language which allows the system administrator to delegate authority to execute commands as root (or other users) to particular users. It also provides detailed auditing records. The most common case is to use sudo for administrative purposes.

[1]This work was partially supported by NSF Grants CNS-04-26623 and CNS-07-14647. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the U.S. Government.

In this common case, the user instructs sudo to execute a command, and sudo checks a configuration file to verify that the user has permission to run the command as root. It then authenticates the user via password. If the password is correct, sudo then executes the command as the root user, while logging the details. In some environments [12, 5], sudo has supplanted the need for a root user entirely.

The sudo authentication mechanism can be configured to require the user's password (by default), the root password, or no password. The sudo package also includes extra authentication modules for Kerberos, Secureware, and SecurID, among others. The extra authentication modules operate in addition to the password requirements. That is, depending on the configuration, a user may have to enter his Kerberos password, and the root password. There is no module for public key-based authentication. Heavyweight schemes like Kerberos and SecurID require substantial investments in time and/or money, so the best options for small- to medium-sized networks are lightweight schemes such as password authentication or S/KEY [3].

S/KEY is a one-time password scheme which requires the end-user to maintain a hard-copy list of passwords, or to run a one-time password generator locally for each authentication attempt. Sudo requires re-authentication every five minutes, by default, so S/KEY is feasible for only the most dedicated. As a result, password authentication is the most common technique.

In this paper, we propose an extension to the sudo authentication mechanism to allow for public key authentication with the sudo engine. We describe our implementation of an BSD authentication module that uses SSH_AUTH_SOCK to authenticate incoming clients. We then describe the source code and configuration changes required to provide public-key authentication with sudo.

Related Work

The sudo [11] architecture was designed by Bob Coggeshall and Cliff Spencer at SUNY/Buffalo in the early 1980s for a VAX-11/750 running 4.1BSD. Control of sudo has passed through many hands in the intervening years. It is currently maintained by Todd Miller.

The SSH [15] protocol was designed in 1995 by Tatu Ylonen at Helsinki University of Technology. There are now several competing implementations of the original protocol and its derivatives, including the implementation released by SSH Communications Security (founded by Tatu Ylonen), and OpenSSH, developed by the OpenBSD project. Our work is based on the OpenSSH implementation.

The BSD Authentication framework [1] is an authentication framework used by some UNIX-like operating systems including OpenBSD and BSD/OS. It is similar in spirit to the Pluggable Authentication Modules (PAM) [10] found in Linux and Solaris. Neither provides a mechanism for interaction with the SSH authentication system.

Other related works include Kerberos [7] and LDAP [4] which provide unified network authentication mechanisms. LDAP also provides the Proxied Authorization Control [13]. In larger networks, RADIUS [8] and its successor DIAMETER [2] provide authentication, authorization and accounting protocols. They allow communication with a policy server to make policy-based decisions. These latter protocols are typically used for user administration in roaming and dial-up situations.

In [6], Napier describes a security flaw in the default sudo configuration wherein an attacker can obtain victim's sudo privileges without the victim's password. Sudo caches the user's password so that a user only has to enter his password every five minutes. That cache is valid for all TTYs on which the user is logged in. If the attacker can run an arbitrary process as the victim, then the attacker will have the same root privileges. Napier recommends turning off password caching entirely, but recognizes that this would encourage administrators to use a root shell to avoid retyping their passwords – a problem when it comes to auditing. Sudo with public-key authentication will allow the system administrator to turn off password caching, and does not require frequent re-typing of the administrator's passwords.

Sudo and SSH

Our implementation platform is the OpenBSD 4.2 operating system. This operating system ships with OpenSSH 4.7 and sudo 1.6.9p4, with OpenBSD customizations. In this section we will describe some relevant details from the BSD Authentication framework, sudo, SSH and the ssh-agent, as they exist in OpenBSD 4.2.

BSD Authentication Framework

OpenBSD uses the BSD Authentication framework, sometimes called bsd_auth, to present a uniform API to the various authentication modules. bsd_auth maintains a collection of modules in /usr/libexec/auth/. Each module performs a particular style of authentication, including password, Kerberos, S/KEY, and RADIUS, among others. A user program interacts with the authentication framework by making calls on the bsd_auth API. The API then executes the modules as separate processes in order to limit interactions between the child and parent processes, under the principal of least privilege [9].

The BSD Authentication framework is configured through the file /etc/login.conf. This file allows the user to add new authentication styles, and to assign styles to specific users or programs.

Sudo

The sudo command takes as a command line argument the command the user desires to have executed another user (typically root). It then searches for

the user's login name in the configuration file /etc/sudoers, and verifies that the user has the correct permissions. Beyond this point, the sudo implementation in OpenBSD diverges slightly from the general sudo release.

The general sudo release is configured to support multiple authentication styles, including passwords, Kerberos, and SecurID. The code for implementing each of these styles is included with the sudo release. In OpenBSD, these are bypassed and the operating system's BSD Authentication framework is called instead. To effect this, the sudo authentication modules have been replaced with a single module bsdauth which, in turn, provides all interaction with the BSD Authentication framework. Thus, on OpenBSD, the authentication styles presented by sudo are those supported by the BSD Authentication framework.

Thus, the complete authentication process with sudo is as follows. The bsdauth module uses the BSD Authentication framework API to authenticate the user. The API uses the definitions in login.conf to determine the particular authentication type, and then loads the appropriate module from /usr/libexec/auth. The module is loaded, it performs the authentication process, and then completes with success or failure. If successful, sudo executes the given command in a cleansed environment. That is, all environment variables except LOGNAME, SHELL, USER, and USERNAME are removed.

SSH and the SSH Agent

The SSH-agent is used to facilitate the use of public keys with SSH. We will walk through a sample SSH connection, from an SSH client to an SSH server, with SSH-agent enabled on the client, to illustrate their interaction in the authentication process. Before the authentication process can begin, the keys must be generated using the ssh-keygen utility and the public key must be stored on the server – appended to the user's .ssh/authorized_keys file.

The user then starts the SSH-agent, and verifies that the SSH_AUTH_SOCK environment variable has been exported to the environment that will use the agent. The user adds his keys to the agent using the ssh-add command. The agent prompts the user for the password to each key and then loads them into memory for future use. Once the key has been added to the agent, the client is ready to initiate an SSH connection.

The SSH protocol architecture consists of a transport layer protocol, a user authentication protocol, and a connection protocol. The transport layer protocol provides encryption, integrity protection and server authentication. The transport layer is negotiated first. When it is complete, the user authentication protocol begins, as described in [14]. The client requests public key authentication:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name in US-ASCII
string    "publickey"
boolean   FALSE
string    public key algorithm name
string    public key blob
```

Where the 'public key blob' may contain certificates. If the public key matches a public key stored on the server, the server accepts the request:

```
byte      SSH_MSG_USERAUTH_PK_OK
string    public key algorithm
string    public key blob
```

The client uses the SSH_AUTH_SOCK tunnel to obtain from the agent a signature over the following data:

```
string    session identifier
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm
string    public key for authentication
```

And returns it to the server:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key for authentication
string    signature
```

If the server is able to verify the signature, the authentication has succeeded.

The agent forwarding option on SSH maintains the SSH_AUTH_SOCK variable at each hop in a multi-stage SSH connection. This means that when a client connects to server h_1, the SSH_AUTH_SOCK environment variable is re-created there, tunneled back to the SSH-agent on the client machine. When the user then initiates a connection from h_1 to h_2, the SSH process uses the same SSH-agent connection, with the same authentication process described above, connecting through the SSH_AUTH_SOCK on h_1. We take advantage of the fact that this connection exists and use it to leverage the authentication process to create public-key sudo.

Implementation

Our implementation consists of the addition of a new authentication style login_pubkey to the BSD Authentication framework, and re-configuration of sudo to make use of the new style. We link the login_pubkey module to libssh during compilation, which allows our module to manipulate SSH keys, request signatures, and call other functions used by the SSH client and SSH server.

We modify the login.conf file to add login_pubkey to the BSD Authentication framework authentication

styles. We also modify the sudoers file, using the env_keep directive, to preserve the SSH_AUTH_SOCK environment variable for the authentication module.

When an authentication module is invoked by the BSD Authentication API, it receive a number of arguments from the calling process. These arguments provide the necessary information for authentication to occur. Parameters include the name of the user being authenticated, the authentication service being requested, and several other options, including details on whether the authentication service being requested is a challenge or a response. Since the challenge/response portion of the SSH authentication are handled outside of sudo, the challenge service is ignored by login_pubkey; all work is performed during the response service.

When login_pubkey receives a service request of type *response*, this indicates that the parent authentication process is awaiting an authentication decision. The module loads the SSH_AUTH_SOCK environment variable and opens the socket which tunnels to the SSH-agent, using the SSH client function ssh_get_authentication_connection(). The module then queries the agent for key details using the ssh_get_first_identity() and ssh_get_next_identity() functions.

With the private key details in hand, login_pubkey then searches the user's authorized_keys file for a corresponding key by calling the SSH server function user_key_allowed(). It requests the agent signature by calling ssh_agent_sign, and verifies it by calling key_verify(). If it succeeds, then user has successfully authenticated. If any of the functions above fail, the process is considered a failed authentication.

This process is identical to the authentication process performed in SSH, with login_pubkey serving as both SSH client and SSH server with respect to the SSH-agent. At process completion, a standard SSH public key authentication has taken place, and no sensitive information has been revealed on the server. Furthermore, with agent forwarding, the SSH_AUTH_SOCK is recreated, tunneling back to the original agent, on each subsequent hop in a multi-hop session, so this process remains unchanged even in multi-hop sessions.

Example Session

In this section, we will walk through an example session using sudo with public key authentication. The user starts his SSH-agent, and checks to make sure that the SSH_AUTH_SOCK environment variable has been created. This variable contains the filename of the UNIX domain socket connected to the agent.

```
castor% ssh-agent
castor% echo $SSH_AUTH_SOCK
/tmp/ssh-aHYaC22922/agent.22922
```

The user then adds his private key to the agent. The agent now manages the private key, and it is now possible to make queries through SSH_AUTH_SOCK against this key. We also make use of the -c option which is discussed in the following section.

```
castor% ssh-add -c .ssh/id_rsa
Identity added: .ssh/id_rsa
```

Next, the user connects to the remote server making sure to enable agent forwarding. After connecting, the user checks to make certain that the SSH_AUTH_SOCK has been forwarded.

```
castor% ssh -A pollux
pollux% echo $SSH_AUTH_SOCK
/tmp/ssh-kOfZrk1118/agent.1118
```

It is now possible to use sudo with public key authentication:

```
pollux% ./sudo -a pubkey /bin/ls
file.1  file.2  hello.txt
pollux%
```

From the server pollux, connect to another server clytemnestra, again with agent forwarding.

```
pollux% ssh -1 -A clytemnestra
clytemnestra% echo $SSH_AUTH_SOCK
/tmp/ssh-jjneu20310/agent.20310
```

Sudo on clytemnestra uses the local SSH_AUTH_SOCK which tunnels through pollux to the agent on castor and authenticates the user.

```
clytemnestra% ./sudo -a pubkey /bin/ls -a
this_is_an_empty_file  hello.txt
clytemnestra%
```

As long as the SSH_AUTH_SOCK is forwarded, the number of intervening hops does not affect the authentication mechanism.

Discussion

The adversary we consider is one who has a root compromise on the remote host. He may have inserted, among other things, malicious replacements for the SSH and Sudo executables and the authentication modules. Our goal is to prevent the adversary from obtaining any sensitive authentication materiel (such as a password).

From this adversary, SudoPK can be viewed as an API on top of SSH agent forwarding. It does not provide any additional functionality, it simply provides easier access to the agent. Hence, we must focus on attacks on the agent itself.

An adversary who has access to a user's SSH_AAUTH_SOCK can use it as an oracle to generate signatures for connecting to any host for which that user has permission. SSH-agent provides protection against this attack by allowing the user to add keys with the -c option. This option requires the local-host identity be confirmed (via password) before every signature. In this fashion, the end-user will be notified of unauthorized authentication attempts using the SSH-agent.

Even with the -c option in place, the adversary may attempt to race for the SSH_AUTH_SOCK during a

valid attempt. If the adversary wins, the end-user will receive the -c password prompt as expected. To prevent this attack, which is an attack on agent forwarding, regardless of whether SudoPK is in place, we modify the SSH agent to present the user with the data to be signed and requesting confirmation before generating a signature.

Conclusion

The public-key sudo mechanism is an implementation that solves a specific and common problem. However, the concepts used here are generic and can be extended. The notion of public-key authentication through secure tunnels, as implemented in SSH with agent forwarding, is quite powerful. The login_pubkey module is a generic interface to that mechanism and can be used by any application, not just sudo. We believe that this architecture is applicable in other scenarios, including remote attestation. In this scenario, the trusted platform module (TPM) takes the place of the SSH-agent, and queries against the TPM may be made through the corresponding agent tunnel.

Author Biographies

Mack Lu received his B.S. in Computer Engineering from Columbia University. During his undergraduate years he worked at the Network Security Laboratory. He is currently at Google.

Matthew Burnside is a Ph.D. student in the Department of Computer Science at Columbia University. He works for Professor Angelos Keromytis in the Network Security Lab. He received his B.A in Computer Science and M.Eng in Computer Science and Engineering from MIT. His research interests are in network anonymity, trust management, and enterprise-scale policy enforcement.

Angelos Keromytis is an Associate Professor with the Department of Computer Science at Columbia University, and director of the Network Security Laboratory. He received his B.Sc. in Computer Science from the University of Crete, Greece, and his M.Sc. and Ph.D. from the Computer and Information Science (CIS) Department, University of Pennsylvania. He is the author and co-author of more than 130 papers on refereed conferences and journals, and has served on over 70 conference program committees. He is an associate editor of the ACM Transactions on Information and Systems Security (TISSEC). He recently co-authored a book on using graphics cards for security, and is a co-founder of StackSafe Inc. His current research interests revolve around systems and network security, and cryptography.

Bibliography

[1] *BSD Authentication System*, http://www.openbsd.org/cgi-bin/man.cgi?query=bsd_auth .

[2] Calhoun, P., A. Rubens, H. Akhtar, and E. Guttman, ‘‘DIAMETER Base Protocol,’’ *Internet Draft*, Work in progress, Internet Engineering Task Force, December, 1999.

[3] Haller, Neil M., ‘‘The S/KEY One-time Password System,’’ *Proceedings of the Internet Society Symposium on Network and Distributed Systems*, pp. 151-157, 1994.

[4] Harrison, R., ‘‘Lightweight Directory Access Protocol (LDAP): Authentication Methods and Security Mechanisms,’’ *RFC 4513*, June, 2006.

[5] *Mac OS X*, http://www.apple.com/macosx .

[6] Napier, Robert A., ‘‘Secure Automation: Achieving Least Privilege with SSH, Sudo and Setuid,’’ *18th Large Installation System Administration Conference*, pp. 203-212, November, 2004.

[7] Neuman, B. Clifford and Theodore Ts’o, ‘‘Kerberos: An Authentication Service for Computer Networks,’’ *IEEE Communications*, Vol. 32, Num. 9, pp. 33-38, 1994.

[8] Rigney, C., A. Rubens, W. Simpson, and S. Willens, ‘‘Remote Authentication Dial In User Service (RADIUS),’’ *Request for Comments (Proposed Standard) 2138*, Internet Engineering Task Force, April, 1997.

[9] Saltzer, Jerome H. and Michael D. Schroeder, ‘‘The Protection of Information in Computer Systems,’’ *Proceedings of the IEEE*, Vol. 63, Num. 9, pp. 1278-1308, 1975.

[10] Samar, V. and R. Schemers, *Unified Login with Pluggable Authentication Modules (PAM)*.

[11] *Sudo*, http://www.sudo.ws .

[12] *Ubuntu 8.04*, http://www.ubuntu.com .

[13] Weltman, R., ‘‘Lightweight Directory Access Protocol (LDAP) Proxied Authorization Control,’’ *RFC 4370*, February, 2006.

[14] Ylonen, T., ‘‘The Secure Shell (SSH) Authentication Protocol,’’ *RFC 4252*, January, 2006.

[15] Ylonen, T., ‘‘The Secure Shell (SSH) Protocol Architecture,’’ *RFC 4251*, January, 2006.