Implementing Zero–Knowledge Authentication in OpenSSH

Andreas Gaupmann Christian Schausberger Ulrich Zehl
Jürgen Ecker

{andreas.gaupmann,christian.schausberger,ulrich.zehl,juergen.ecker}@fh-hagenberg.at

Abstract: The SSH protocol provides a secure way of accessing remote systems. Besides traditional password-based authentication mechanisms, it currently only includes public-key authentication as a cryptographic method. This paper introduces an additional method based on a zero–knowledge proof of knowledge and the necessary additions to the SSH protocol. Additionally, we present a proof-of-concept implementation of our zero–knowledge scheme for OpenSSH, an open source SSH implementation.

1 Introduction

SSH is excessively used for remote administration. Beside traditional password-based login mechanisms, SSH also offers public key cryptography challenge-response methods to authenticate users. These schemes do have one disadvantage, however. The messages sent during the authentication process contain information about the user's secret. Although this information is not easily accessible by an attacker, some information might nevertheless be leaked. Using a zero–knowledge scheme, one can be sure that no information whatsoever is leaked about the user's secret. Therefore, zero–knowledge schemes are a perfect fit when it comes to user authentication. OpenSSH[1] is one of the most popular SSH implementations today. Additionally, it is distributed with an open source license, so the code may be changed and published freely under the same terms.

The cryptographic concept of zero–knowledge proofs of knowledge was first introduced by Goldwasser, Micali and Rackoff in [GMR85]. For a comprehensive introduction to zero–knowledge, see [Gol02] or [Gol01, Chapter 4]. A recent user authentication protocol with the properties of a zero–knowledge identification protocol is the SRP protocol [Wu05]. A patch for enabling SRP in OpenSSH also exists. Its goal is to provide a secure means for password authentication. Authentication based on the Diffie–Hellman problem is used in the protocol after a private key was derived from the password using a hash function. Nevertheless, weak passwords make dictionary and brute force attacks feasible regardless of how secure the authentication process is carried out. Therefore, a zero–knowledge protocol (ZK protocol) without this disadvantage is desirable. The zero–knowledge protocol we implement is due to Ohta and Okamoto [OO88]. It is a modification and generalization of the well-known Fiat-Shamir scheme [FS86] (U.S. Patent 4,748,668) with one crucial advantage: it is, to the best of our knowledge, not encumbered by patents. Another popular zero–knowledge identification scheme by Guillou and Quisquater [GQ88] is also

[1] http://www.openssh.com

patented as U.S. Patent 5,140,634. Although the patents mostly apply to usage with smart cards or other computationally bounded devices, we wanted to use a scheme that can be freely implemented into OpenSSH.

2 The Ohta-Okamoto zero–knowledge authentication protocol

Ohta and Okatomo present a sequential version in some detail and discuss the security of a parallel scheme in [OO88], but they do not present the parallel protocol's actions. Therefore, we will do this in the following section.

A prover P chooses an RSA-like modulus n (the product of two secret large primes) and k random integers $S_1, \ldots, S_k \in \mathbb{Z}_n$ as his private identity keys, and a small integer L. His private key is $((S_1, \ldots, S_k), n, L)$. P now computes $I_j = {S_j}^L (\text{mod } n)$ for all $1 \leq j \leq k$, and publishes $((I_1, \ldots, I_k), n, L)$ as his public identity key.

The proving party P wants to prove its purported identity to the verifying party V. To this end, the following protocol is carried out over t rounds until V is convinced of the correctness.

1. P chooses a random $R \in \mathbb{Z}_n$ and sends the witness $X = R^L (\text{mod } n)$ to V.
2. V picks a challenge vector $(e_1, \ldots, e_k) \in (\mathbb{Z}_L)^k$ at random and sends it to P.
3. P computes $Y = R \cdot \prod_{j=1}^{k} {S_j}^{e_j} \pmod{n}$ and sends Y as his response.
4. V accepts the round if and only if the following equation holds.

$$Y^L = X \cdot \prod_{j=1}^{k} {I_j}^{e_j} \pmod{n}$$

After t successful rounds, V accepts the proof. The probability that a cheating prover can successfully fool an honest verifier is only L^{-kt}.

We will work in a remote attacker scenario where no assumptions about his power—beyond being polynomially bounded—are reasonable. Therefore, the probability that a cheater can successfully fool an honest verifier should be at most 2^{-80}. Choosing $L = 4$, $k = 10$ for the number of identity keys, and $t = 4$ as the number of rounds yields the desired probability. We propose to use at least 2048 bits for the length of the modulus n to have a margin of security for future developments in factorization and cryptanalysis. In one authentication pass, each party needs to perform roughly 40 RSA-like operations (exponentiation modulo n) with very small exponents ($\leq L$). As $L = 4$, this does only have a negligible impact on authentication time in the implementation.

3 Adjustments to the OpenSSH 4.0p1 code base

The format of the extension is a patch that can be applied to the OpenSSH portable source code before compilation. The zero–knowledge protocol applies only to authentication and leaves other parts of the SSH protocol untouched, as due to the nature of zero–knowledge proofs of knowledge, they cannot be used for key distribution.

The source code is divided into three parts: server, client and additional utilities like key generator or agents for holding keys. Server, client and key generator were modified for handling the chosen ZK protocol and its keys. Two functions, each named `userauth_zk`, were implemented for processing and answering requests, one for the server (`auth2-zk.c`) and the other for the client (`sshconnect2.c`). Only SSHv2 was extended.

The response of the client (Step 3 of the protocol) is received and checked for its correctness by calling functions of the module that implement the calculations according to the Ohta–Okamoto zero–knowledge protocol (`ohta-okamoto.c`). Sending a packet can be achieved by using the interface for the Binary Packet Protocol provided by OpenSSH. The nifty details of assembling the packet that really goes out on the wire are hidden in comfortable high–level functions.

In contrast to the server, where every authentication method is implemented in its own implementation file, at the client side all methods are joined in a single file `sshconnect2.c`. This file contains an array of authentication methods and one or more functions specifically for handling a certain method. The array has to be extended by one element and at least two new functions have to be introduced: The function `userauth_zk` sends the initial zero–knowledge user authentication request and generates responses to the challenges issued by the server. The function `input_userauth_zk_ok` answers to a `SSH_MSG_USERAUTH_ZK_OK` received from the server.

The extension of OpenSSH with a zero–knowledge user authentication has led to the introduction of a new option `ZeroKnowledgeAuthentication` in `ssh_config` and `sshd_config`. Alternatively, the user may provide

```
$ ssh user@host -o ZeroKnowledgeAuthentication
```

on the command line. By default zero–knowledge authentication will be used as one of the preferred user authentication methods (the others being “public key” and “keyboard–interactive”). Zero–knowledge authentication can be disabled explicitly by passing `no` to the option shown above.

Keys are generated with a separate program named `ssh-keygen`. It enables the user to generate RSA1, RSA, DSA and `OO_ZK` keys and to modify their optional attributes. The generated keys are stored in identity files in a `.ssh` subdirectory of the user’s home directory. OpenSSH uses the OpenSSL–API functions for PEM encoding to store RSA and DSA keys in files. This is easily done, because the RSA and DSA data types are also taken from OpenSSL which provides routines for PEM encoding and decoding exactly for those key (data) types. In our implementation ZK keys are written to disk using a custom binary file format. The functions for RSA1 keys were taken as a template and adapted to handle `OO_ZK` keys. The identity files are accessed by the client as soon as zero–knowledge user authentication or any method that uses public keys is started. The private key is needed for calculating the witness or signature, respectively. These operations are conducted in the method–specific functions implemented in the `sshconnect2.c` file. Beside key generation `ssh-keygen` also offers options for key manipulation like changing comments for RSA1–style keys and changing the passphrase used for encrypting the private key. This functionality is not implemented for `OO_ZK` keys, yet.

4 Security issues and additional information

Privilege separation is an optional mode of running for the server. It increases security by reducing the runtime in which the server possesses user or even root privileges. These extra privileges are only needed for binding to port 22 or reading keys. The rest of the code can be executed without them. Currently the protocol implementation does not work when privilege separation is enabled. The reason for this is yet unknown.

In [DNS98], Dwork, Naor, and Sahai consider problems with zero–knowledge protocols when the prover is involved in (polynomially) many proof protocols at the same time. For our application this scenario is a rather unusual one, so that we do not see a need to consider this setting. The same is true for the notion of resettable zero–knowledge in [CGGM99]. In this scenario the attacker is able to reset the verifier, forcing him into the exact same state he had before starting the protocol.

The patches for the OpenSSH source code, a more detailed documentation of the project, and additional information about zero–knowledge can be obtained at `http://zk-ssh.cms.ac`.

References

[CGGM99] Ran Canetti, Oded Goldreich, Shafi Goldwasser, and Silvio Micali. Resettable Zero-Knowledge. Technical Report TR99-042, Electronic Colloquium on Computational Complexity (ECCC), 1999.

[DNS98] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent Zero-Knowledge. In Jeffrey Vitter, editor, *Proceedings of STOC '98*, pages 409–418. ACM Press, 1998.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology –* CRYPTO *1986*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In Robert Sedgewick, editor, *Proceedings of STOC '85*, pages 291–304. ACM Press, 1985.

[Gol90] Shafi Goldwasser, editor. *Advances in Cryptology –* CRYPTO *'88, Santa Barbara, California, USA, 1988, Proceedings*, volume 403 of *Lecture Notes in Computer Science*. Springer, 1990.

[Gol01] Oded Goldreich. *Foundations of Cryptography, Volume 1: Basic Tools*. Cambridge University Press, 2001.

[Gol02] Oded Goldreich. Zero-Knowledge Twenty Years After Its Invention. Technical Report TR02-063, Electronic Colloquium on Computational Complexity (ECCC), 2002.

[GQ88] Louis C. Guillou and Jean-Jacques Quisquater. A "Paradoxical" Identity-Based Signature Scheme Resulting From Zero-Knowledge. In Goldwasser [Gol90], pages 216–231.

[OO88] Kazuo Ohta and Tatsuaki Okamoto. A Modification of the Fiat-Shamir Scheme. In Goldwasser [Gol90], pages 232–243.

[Wu05] Thomas Wu. The Secure Remote Password protocol. URL, The Stanford SRP Authentication Project, August 2005. *http://srp.stanford.edu/index.html*.