An SSH-based toolkit for User-based Network Services

Joyita Sikder
Univ. of Illinois at Chicago

Manigandan Radhakrishnan
VMware

Jon A. Solworth
Univ. of Illinois at Chicago

Abstract

Network authentication, even when using libraries intended to simplify the task, is inordinately difficult. Separate libraries are used for cryptography, network authentication protocols, accessing stored authentication information, and verifying the identity of remote entities. In addition, service used must be authorized. Finally, privilege separation is needed to separate security sensitive, highly privileged operations from the remainder of the application.

These tasks consume thousands of lines of application source code (not counting the security libraries on which they rely), and require much specialized security knowledge from the application programmer and system administrator.

In this paper we present a simple toolkit called `sshUbns` which encapsulates all these tasks in an easy-to-use tool. We modified SSH to add in `sshUbns` (in addition to SSH's other modes) and implemented a new super-server called `unetd`. It reduces to a negligible level the amount of application server security code needed. This toolkit makes it easier to create secure networking code, reduces security specific knowledge needed by application programmers, and makes it easier for system administrators to protect and analyze their systems.

1 Introduction

Network service user authentication seems to be a simple procedure: The user provides either a password or some cryptographic proof of her identity to the remote service. The service verifies the user's identity, and authentication is complete.

In practice, however the task is far more complex:

- Passwords, if used, must be of sufficient diversity to prevent dictionary attacks. Since attackers today have access to large botnets, password attacks consisting of millions of guesses are easily possible, even if a host is blacklisted after a few tries. On the other hand, if cryptography is used it must be implemented correctly to prevent side channel attacks (thus exposing secret keys) and to ensure sufficient randomness of keys (preventing brute force attacks).
- Authentication must be mutual so that the user knows that she is talking to the legitimate service. This is typically done cryptographically, for example with RSA [20].
- To maintain authentication after the initial authentication protocol, cryptography is used to prevent undetected packet modification (and prevent viewing) in transit. Symmetric cryptography, such as AES [12], is used to provide these protections.
- If the service is not anonymous, it is necessary to authorize users. The user must be allowed to perform the service and the service's permissions must be tailored to those of the user.

The complexity is not limited to cryptographic algorithms and network protocols. In addition, a complex software stack is used. For example, Generic Security Services (GSS-API) transmits authentication tokens between client and server [17]; Network Services Switch (NSS) accesses the stored authentication information; and Pluggable Authentication Modules (PAM) actually authenticates the user [22]. Failures in the use and configuration of this software can violate authentication and authorization requirements. Ensuring that these tasks are properly done in traditional schemes requires examining each service's code and verifying that security services are properly used.

Finally, traditional mechanisms are implemented with libraries which share the address space of the application. When application logic and authentication sit in the

same address space, there is a danger that failures in application logic (e.g., buffer overflow) can cause authentication to fail—for example, by bypassing authentication all together. Moreover, these applications often need superuser privileges to bind to restricted ports or to change the user ID on whose behalf the service runs. Without careful partitioning, there is substantial code which runs with excess privileges. If this code is successfully attacked, these excess privileges increase the damage that the attacker can do.

To prevent these authentication failures, *privilege separation* is used [18, 7]. Privilege separation partitions logic over multiple processes so that most code runs with reduced privileges. Security sensitive code is isolated in a separate process with administrative privileges; the remaining parts of the application can then be run without administrative privileges. Using privilege separation, a highly privileged isolated process performs operations as a proxy for the application. This requires partitioning of the application and inter-process communication.

We consider here the most demanding of these problems, *User-Based Network Services (UBNS)* in which the service process operates with user-specific privileges, thus using the *Operating System (OS)* to restrict service accesses. UBNS services use OS access controls to limit the accesses that a service is allowed to do (by running user-specific parts of that service under the user's ID), and thus to isolate users from one another. Services such as mail, calendaring, distributed file systems, ftp, and source code revision control systems can be implemented as UBNS. Examples of UBNS services include `dovecot` for IMAP/POP3 mail delivery [1] and `zimbra` for calendaring [2]. Although such services can and have been built without UBNS, they require increased application-level authorization and pose greater dangers due to more application-level vulnerabilities [7].

UBNS is so demanding to implement, that often less secure mechanisms are used instead. For example, using traditional techniques `dovecot` requires 24,628 lines to support IMAP. Of that, over 9,307 lines of code are used for user authentication alone, some 37% of the total code base. In addition, to support privilege separation 4 different process types are used. Using new OS mechanisms, `netAuth` implemented UBNS functionality with only 5 lines (vs. 9,307 in the original `dovecot`) of application code [19]! In addition, the application code was simplified using a single process type (vs. 4 in the original), since privilege separation was provided by the implementation of the authentication. However, `netAuth` required OS kernel modifications and IPSec, and hence the code produced is not widely used.

Here, we describe a toolkit, `sshUbns` which provides almost the same functionality without OS kernel modifications. The `sshUbns` toolkit is built on top of *Secure Shell (SSH)* [26]. Unlike library-based approaches, `sshUbns` is implemented in two separate services, a modified SSH and `unetd`. It uses SSH's strong cryptographic authentication and cryptographic protection of communications over the network; it adds end-to-end security for networked applications. It provides strong protections needed for UBNS and yet is very simple to use. This simplicity is in three separate forms: (a) it is easier for system administrators to set and analyze protections; (b) there is less code for application programmers to write; and (c) higher level abstractions require less security expertise from the application programmer. Hence, the programmer and system administrator's task is simplified since the tool implements authentication, encryption and authorization.

Moreover, `sshUbns` is implemented using privilege separation. Like the kernel-based `netAuth`, `sshUbns` provides strong protections with a minimalistic programming interface. It allows system administrators to easily control who can use a service and to easily launch services, since these protections are provided in a service-independent way by the toolkit. Because it provides a simple toolkit for these important services, a system administrator's job of securing their system is vastly simplified. The `sshUbns` toolkit also supports the easier-to-implement class of services that are restricted to certain users but do not differentiate between authorized users, and hence may run as a pseudo user. However, we'll focus here on the support for UBNS.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 describes SSH's port forwarding mechanism, which is the starting point for constructing `sshUbns`. Section 4 describes the `sshUbns` architecture. Section 5 measures the effectiveness of the implementation. Section 6 describes implementation alternatives and finally we conclude.

2 Related work

UBNS and privilege separation are two complementary ways to partition a service into multiple processes. Privilege separation is used to split an application into root and non-root processes. Both UBNS and privilege separation are design strategies to maximize the value of least privilege [21]. Retrofitting privilege separation is not difficult since root privileges are a super set of ordinary user privileges, and there exists both libraries [15] and compiler techniques [9] to do it. UBNS is more invasive as the privileges of different users overlap, and hence the protection of files and users which own processes must be carefully considered at the start of design.

SSH is a widely used UBNS service [26, 18], but is ill-suited to implement UBNS-based network services because of the way network services are built. In the net-

work case, the listening process exists before the connection is made and must know at connect time which user is associated with the service. SSH's port forwarding performs user authentication at the service host—but not at the service—and hence, to the service, the users of a host are undifferentiated. As a result, traditional UBNS services use authentication mechanisms such as SSL or passwords and OS mechanisms such as `setuid` which are awkward to program and may not be secure.

Alternatively, SSH allows a remote executable to be invoked, but that remote executable is not connected to a network service. Similarly, hg-login [3], as used in Mercurial, performs remote authentication using SSH, but `exec`s a new program rather than connect to a running network service.

In contrast, `sshUbns` both authenticates and authorizes the user, so that the service runs *only* with the permission of the user. Unlike SSH, `sshUbns` provides end-to-end security from client to service. The `stunnel` tool could have been used as an alternative to an SSH-based implementation—it provides similar protections to SSH port forwarding; the primary reason we chose SSH is because it uses a fixed port which is already allowed by our firewall rules.

The OKWS web server [16], built on top of the Asbestos OS [11] does a per-user demultiplex, so that each web server process is owned by a single user—it is another example of a UBNS. However, this facility is provided at the HTTP level via cookies, while the technique presented here is application (and application protocol) independent.

Kerberos [23] performs encryption using private key cryptography. Microsoft Windows' primary authentication mechanism is Kerberos. Kerberos works well in the enterprise, when the user it authenticates is part of the enterprise, but works less well in widely distributed systems. The problem in this setting is that the clients must be "kerberized versions". Kerberos does not directly support UBNS. Moreover, implementing Kerberos for an application is more complex, and less modular than `sshUbns`. Kerberos does have an advantage over our scheme in that it has a key distribution mechanism while SSH does not.

Distributed authentication consist of two components: a mechanism to authenticate the remote user and a means to change the ownership of a process. Traditionally, UNIX performs user authentication in a (user space) process and then sets the User ID by calling `setuid`. The process doing `setuid` needs to run as the superuser (administrative mode in Windows) [24]. To reduce the dangers of exploits using such highly privileged processes, Compartmented Mode Workstations divided root privileges into about 30 separate capabilities [6], including a SETUID capability. These capabilities were also adopted by the POSIX 1e draft standard [5], which was widely implemented, including in Linux.

Plan9's OS kernel uses a fine grained one-time-use capability [10], which allows a process owned by user U to change its owner to U'. It works with `factotum`, a user space process which actually performs the cryptography for the application. The `sshUbns` toolkit unlike Plan9 uses only generic POSIX mechanisms, and thus does not require kernel modifications.

Distributed Firewalls [14] (based on Keynote [8]) in contrast to SSH, implements per user authorization for services by adding it to the OS kernel implementation of `connect` and `accept` APIs. While Distributed Firewalls sit in front of the service, and thus are not integrated with the service, Virtual Private Services are integrated and thus can provide UBNS services [13], but unlike `sshUbns`, this relies upon kernel modifications.

3 SSH port forwarding

The closest service to `sshUbns` is SSH port forwarding. Using SSH, a command, executed by the user on the client

```
ssh -L 3000:localhost:25 example.com
```

results in the local port (3000) being tunneled to host `example.com` at port 25. The command is successful if `sshd` is running on `example.com`; the user has an account there; and port 25 is bound.

Now a process on the client can reach the service at port 25 at `example.com` by accessing port 3000 on the client. The connection between hosts is authenticated and cryptographically protected. The protection is coarse grained, since any user on the client may connect to port 3000—even those without accounts on `example.com`. Moreover, the service at port 25 (smtp) does not know which user is sending to it, although firewall rules can ensure that the port is only reachable from within `example.com`.

The above example assumes that the user name on the client is the same as on the server. If instead, the user's name at `example.com` is say, `dave`, then the SSH command would be:

```
ssh -L 3000:localhost:25 \
        dave@example.com
```

Although we shall assume the names match in the following text neither SSH nor `sshUbns` require this.

Figure 1 shows the traditional SSH port forwarding. (For simplicity, we leave out the server-based root-owned SSH processes which are used to establish the SSH connection). SSH authenticates and encrypts the traffic between client and server hosts. SSH ensures that the endpoints of the SSH tunnel are owned by the same user (U_2

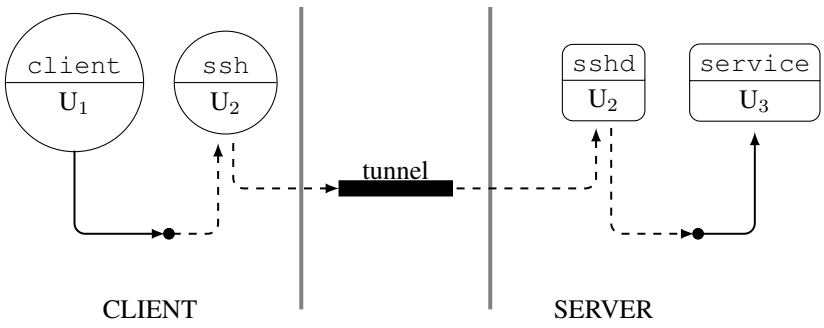

Figure 1: Client host to Service path using traditional SSH tunneling. Processes are indicated by circles or rounded edge rectangles. Above the interior line is the name of the executable, below the line is the user who owns the process.

in the Figure). However, because it is based on network ports—which don't perform any authentication—neither `client` to `ssh` network connection nor the `sshd` to `server` connection is authenticated. Thus traditional SSH is coarse grained, it is insufficient for UBNS such as mail, calendaring, etc. Moreover, since the ultimate user is unknown, logging effectiveness is very limited. Thus we turn to the architecture of `sshUbns`.

4 Architecture

In contrast to traditional SSH port forwarding, shown in Figure 1, `sshUbns` maintains the same user from client application to service, as shown in Figure 2. (Although the user is the same, the user name and user ID may be different on client and server, as per the previous section). It is this end-to-end property which ensures that the user is the same along the entire path which distinguishes `sshUbns` from SSH port forwarding.

The architecture we have implemented consists of three components:

SSH modifications which adds a UBNS mode to client and server sides,

unetd is a simple super-server which supports UBNS, and

server modifications which provide UBNS code to applications.

Of these, by design the server modifications are by far the smallest, since it minimizes the cost of porting servers to `sshUbns`. All the other code is independent of specific services.

4.1 SSH modifications

We have modified SSH to create a UBNS tunnel. This was done by modifying the port forwarding mode of SSH. The first step is to invoke SSH in UBNS mode from the client:

```
ssh -u -L 3000:localhost:25 \
        example.com
```

It is the "`-u`" which invokes `sshUbns`. (Alternatively, `autossh`—which automatically restarts SSH if there is a connection failure—can be used to make the connection robust even when the IP address changes).

We modified both the client side (`ssh`) and the server side (`sshd`) of SSH. On the client side, the `ssh` process which connects to the local port must be running *and* must be owned by the same user as the client process. This prevents other users on the client system (who don't have accounts on the server) from piggybacking on a legitimate user's port forwarding to the server system. Thus `sshUbns` is significantly safer than vanilla SSH port forwarding.

On the server side, we have written a `sshUbns` mode for `sshd` (based on its port forwarding mode) which interfaces with the service and runs on behalf of the remote user. It gets the port number of the user service process using a per service directory which is part of `unetd` (details are given in the next section).

For simplicity, we describe `sshd` as a process which runs on behalf of a user. Actually, to provide privilege separation `sshd` consists of two types of processes; one type which runs as root and the other as the user. However, only the user-owned process communicates with `unetd` and the UBNS.

TCP/IP are used everywhere except for a Unix domain socket between `sshUbns` components on the server side. Since the Unix domain socket is created in the file system, permissions can be (and in `sshUbns` are) set to ensure that the same user who creates the socket opens the existing socket. Unix domain sockets are not available on Windows computers, and in such a case it is possible to use TCP/IP sockets. However, where Unix domain sockets are available they are preferred.

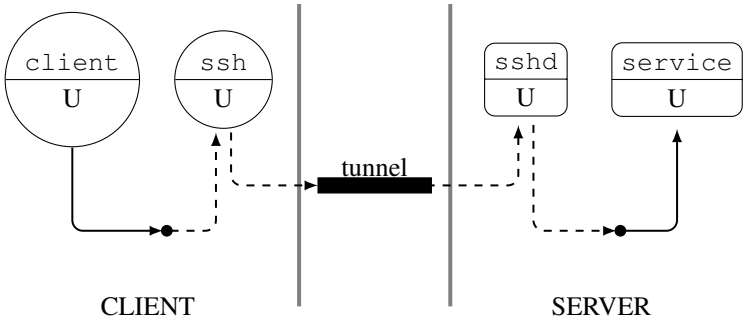

Figure 2: Client to host service path using `sshUbns`. The `client`, `ssh`, `sshd`, and `service` all run under the *same* user.

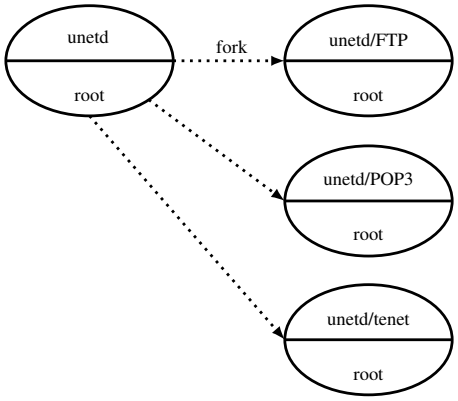

Figure 3: unetd and the service processes it spawns

For TCP/IP connections there is no standard method for user authentication and hence application-level protocols such as SSL are often used. However, when both ends of a TCP/IP connection are on the same host, it is possible to use OS calls to authenticate unmodified TCP/IP traffic. Although the method is non-standard across OSs, each of the major operating systems (Windows, Linux, Mac OSX) can determine the process and owner of the process which is at the other end of a local TCP/IP connection. For example, this information is available using `lsof` in UNIX-based systems or `openports` in Windows-based systems.

4.2 Unetd

We have written a daemon, `unetd` (for user-based network daemon) that launches UBNSs and authorizes users. Unetd is modeled after other super-servers such as `inetd` and `tcpd`. The configuration for `unetd` stored in `/etc/unetd/unetd.conf` contains lines of the form:

*port group * args*

The *port* (or service) specifies the desired service; the *group* specifies those users who are authorized for that service. The "*" is optional and means concurrent server, in which one process is spawned for each user connection. Without the "*" the server process for each user is sequential, meaning at any time there is at most one process per user. The *args* are the arguments with which `unetd/`*service* starts up (that is, `execs`) the *service*. Thus, our mechanism is sufficiently expressive to implement the primary different server types. We could also implement preforked servers, but believe `unetd` is sufficiently flexible without it.

Unetd runs as root, and creates a process per service. As a running example, we'll use POP3 as a service. For POP3, the created per-service `unetd` process is called `unetd/POP3` which listens to the port specified on its service configuration line. The service `unetd/POP3` does not contain any POP3-specific code, its purpose is to authenticate the user and direct the connection to the appropriate user-owned POP3 server.

It also checks that the user is authorized to use the service. When a POP3 `sshUbns` request arrives, `sshd` connects to the `unetd/POP3` and requests the port number of the POP3 process which is specific to that user. Finally, the POP3 process performs the user specific request, relying on the OS's access controls to ensure the accesses are appropriately authorized.

Figure 3 shows a `unetd` process which creates three different service processes, including `unetd/POP3`. All of the processes here are generic; the actual service (and the vast bulk of the code) is performed by user-based services that do not run with administrative permissions. Each service process is created to listen to a single inquiry port from `sshd` and launch the appropriate user based service.

Figure 4 shows the complete tree of processes created by `unetd`, including the service specific component. Each arrow indicates a process was forked. As can be clearly seen `unetd` is a UBNS and all server specific

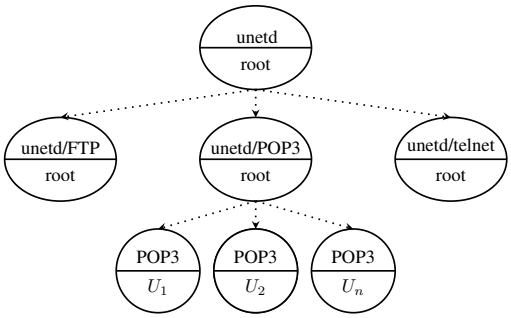

Figure 4: Three levels of processes created by `unetd`. For each process, a process identifier is shown on top, and the user ID on behalf of which the process runs (either root or ordinary users U_1, U_2, or U_n are shown).

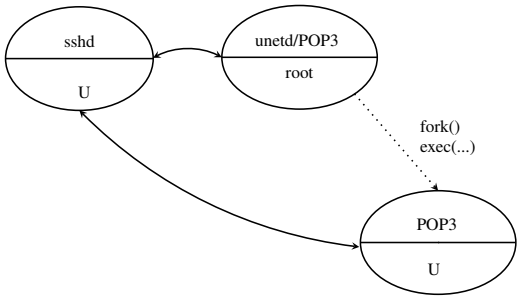

Figure 5: Creating the user-based service

code runs without root privileges.

4.3 Service support

It is trivial to modify a service for UBNS support. The port is opened by the parent process, so the only thing for the service to do is to check that the user of the user-based network service is the same as that of the process at the other end of the TCP/IP or Unix domain socket connection.

This checking is done by replacing the `accept` call with the `acceptUBNS` library call which does both accept and user ID checking. We note that this is the only security-specific call done by the service, the service has no need to deal with cryptography, authentication, user authorization, or privilege separation which are all generic services provided by sshUBNS.

The flow of service invocation on the server is shown in Figure 5. The `sshd` process sends to `unetd/POP3` its TCP port (having previously done a bind) and requests it to send it the port for user U's POP3 service. If none exists, or if POP3 is set up as a concurrent service, then a user-based service is created. Then `unetd/POP3` returns the port number. The `sshd` process then directly connects with POP3 server for U; U's POP3 server then does an `acceptUBNS` which ensures that `sshd` is owned by U thus completing the authentication.

4.4 End-to-end invocation of a user-based service

There is no change to `client` application code. The `client` configuration must specify the local port and local host to connect to `ssh` rather than directly to the service.

We consider the overall flow of a connection. Before this flow begins, we assume that (a) `ssh` in UBNS mode has been invoked on the client (and `sshd` has been started on the server) and (b) the server has invoked `unetd` which has started each UBNS, such as `unetd/POP3`. The overall flow from beginning to end of connection establishment is diagrammed in Figure 6. The trace of a connection is as follows:

1. the client application connects to `ssh` on the client,
2. `ssh` on the client connects to `sshd` on the server,
3. `sshd`
   (a) binds to a TCP/IP port p
   (b) sends p to `unetd/POP3` and asks for U's port address for POP3,
4. If the user is not in the group of users who are authorized to use that service, then `unetd/POP3` sends a failure message to `sshd`. Otherwise
5. If "*" has been specified in the configuration file or if there is no service for that user, then `unetd/POP3` does the following
   (a) a TCP/IP listening socket is created for the process to be forked,
   (b) a service process is `fork`ed and `exec`ed,
   (c) the UID of the resulting process is changed, and
   (d) the service executable is `exec`ed.
6. The `unetd/POP3` process replies back to `sshd` with the port number of the user's service process, and
7. `sshd` connects to the user's service process. which tests that it is coming from port p. Since `sshd` has been bound to port p, the connection must be from the specified user.

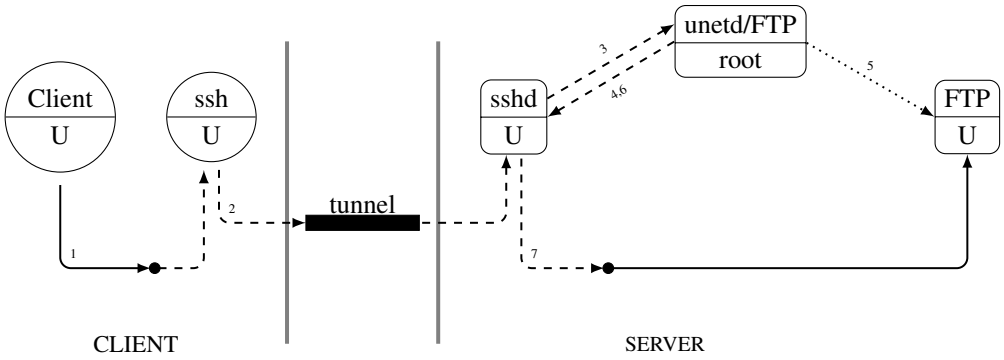

Figure 6: Overall flow

measure	time
real	0.004
user	0.000
sys	0.003

Figure 7: Time for a client response in seconds

measure	10 users
real time	3.390
user time	1.636
sys time	3.488
connections/second	295

Figure 8: Bandwidth measure (connections/second) and time for 10 users to each performed 100 POP3 connections

5 Experimental results

We have performed some initial testing of the performance of `sshUbns`. The testing was done on an AMID 4600+ 64x2 Dual Core in 64-bit mode. The software base is OpenSSH 5.1 (patch 1) and we ported `dovecot`'s POP3 server. Testing was done on a gigabit LAN. We used RSA keys.

In Figure 7 the client response time is shown for a remote use of POP3. The client side is much easier to measure than the server side, since we can simply measure the time for a trivial POP3 session, in which the only command is "bye". Total CPU elapse time is .004 seconds on the client side; we expect the server time to be slightly longer as it has an extra connection (to find the port of the UBNS). The first time `sshUbns` runs for a user it must also do a fork-exec of the user-owned service.

We have done some primitive tests to measure bandwidth using 10 users each doing 100 connections, the numbers are shown in Figure 8. The results show 295 connections per second on a dual core, or 147.5 connections per core per second. We have not had yet an opportunity to do any tuning which we expect will significantly increase performance.

We intend to do a number of ports to `sshUbns`, for example of web servers and calendaring systems.

6 Alternatives and future work

We would have liked to use UNIX Sockets throughout. This would have removed the need to do an `acceptUBNS` instead of an `accept` and authorization to connect to the UNIX socket could be done by the UNIX socket mechanism. Unix domain sockets are also considerably faster than using `lsof`. This performance advantage is far more important on the server side, and hence we have assumed it for our server experiments.

Using UNIX Sockets is considerably less attractive on the client side, since (1) client software many not be under the control of the server organization, (2) client software may be proprietary and hence not easily modified, and (3) there may be many different implementations of client software (e.g., many different mail user agents) thus increasing the difficulty of modifying them. The performance issues for the client are small, since each client is expected to use only a relatively small number of `sshUbns` connections.

SSH can be set up to be based on public key only, or to allow a combination of public key and password. Public key authentication is more secure, but requires some method for installing the public keys on the servers.

We could have used the ability to transfer a file descriptor over a UNIX socket to make `unetd/POP3` send the connection transparently to POP3 service for that user. This would allow local (i.e., non-networked) clients

to connect transparently to a UBNS. We will implement this in the next version of our software.

The current implementation requires patching `sshd` and possibly to the application (if we choose to use UNIX sockets for communication). A less invasive approach would be to make these changes as part of some library or wrappers (like TCP Wrapper [25]) that are linked with the program. This imposes difficulties for two reasons: (1) the communication between `sshd` and `unetd/POP3` and between POP3 and `unetd/POP3` has to be done in the library or wrapper and as part of `accept` or `connect` and (2) both `sshd` and the application (POP3) may be required to perform security critical operations before and/or the establishment of a connection that may not be securely performed without patching the application.

It would be interesting to extend this mechanism to applications which don't easily support port redirection (e.g., some web servers). Since the ports are not known in advance, some mechanism would be needed to examine packets without redirection; we are considering using TUN/TAP for this interface [4]. The TUN interface would also make invoking `sshUbns` transparent on the client.

The design of `sshUbns` is intended to be able to run on Windows as well as Unix-based hosts. We have used Unix domain sockets in only one single place, on the server side. To port this code to a Window's server, it would be necessary to use some other form of IPC, for example TCP/IP and to use `openports` for authentication between `sshd` and `unetd/POP3`. Similarly, `openports` could be used on the client side for connection between the application and `ssh`.

We have not made any attempt to make `sshUbns` fast. For large configurations, the cost of doing these operations may be significant, and performance optimization important. This is left for future work.

7 Conclusion

Often, it is assumed that security must be traded off against other properties such as usability or code complexity. Sometimes, however, we pay a far higher price for security than is necessary, largely because of the history of incrementally adding security. Comprehensive toolkits—which manage a set of related security issues—can have significantly lower overall complexity than a piecemeal approach while attaining strong security.

We built this tool because we wanted a better way of using and constructing authenticated services. The toolkit, `sshUbns`, is painless to use as it requires only a single line of code in an application to provide authentication, authorization, encryption. It is privilege separated, thus isolating security sensitive operations from the application. Issues of key size, authentication method, and many other issues become irrelevant for the application programmer. However, porting code is more involved because of the large number of lines of code which must be removed from legacy code. We plan to do several more ports.

The `sshUbns` toolkit is particularly attractive for system administrators. First, system administrators are adept at configuring solutions from tools. Second, `sshUbns` is general purpose and thus applicable to a whole range of networked applications. Third, it builds on well known tools and concepts, notably SSH and super-servers. Fourth, it avoids much of the need to individually examine application code and configurations to determine setting, a time consuming and unfortunately error prone process. Fifth, it is consistent across applications, reducing user education and system documentation issues.

Acknowledgements The program committee reviewers provided detailed, extensive, and useful comments. Two of the reviewers served as shepherds for the paper. David Plonka did an amazing and energetic job of providing many notes, giving us suggestions, and keeping us on schedule. William LeFebvre provided useful comments and kept us centered on the most important issues. Thanks to Wenyuan Fei, Prasad Patil, and Michelle Zhou for proofreading.

References

[1] http://www.dovecot.org/.

[2] http://www.zimbra.com/.

[3] http://www.selenic.com/mercurial/wiki/index.cgi/SharedSSH.

[4] vtun.sourceforge.net/tun/.

[5] IEEE/ANSI Draft Std. 1003.1e. Draft Standard for Information Technology–POSIX Part 1: System API: Protection, Audit and Control Interface, 1997.

[6] Jeffrey L. Berger, Jeffrey Picciotto, John P. L. Woodward, and Paul T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, 1990. Special Section on Security and Privacy.

[7] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *First Computer Security Architecture Workshop*, page 1. ACM, 2007. Invited paper.

[8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. RFC 2704: The KeyNote Trust-Management System Version 2, September 1999.

[9] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[10] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in Plan 9. In *Proc. of the USENIX Security Symposium*, pages 3–16, 2002.

[11] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, 2005.

[12] FIPS. *Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, pub-NIST:adr, November 2001.

[13] Sotiris Ioannidis, Steven M. Bellovin, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Virtual private services: Coordinated policy enforcement for distributed applications. *IJNS*, 4(1), January 2007. `http://www1.cs.columbia.edu/~angelos/Papers/2006/ijns.pdf`.

[14] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and Communications Security*, pages 190–199. ACM Press, 2000.

[15] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284. USENIX, 2003.

[16] Maxwell N. Krohn. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track*, pages 185–198, 2004.

[17] John Linn. Generic interface to security services. *Computer Communications*, 17(7):476–482, July 1994.

[18] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242. USENIX, August 2003.

[19] Manigandan Radhakrishnan and Jon A. Solworth. NetAuth: Supporting user-based network services. In *Usenix Security*, pages 227–242, 2008.

[20] Ronald Rivest, Adi Shamir, and L. Adleman. On digital signatures and public key cryptosystems. *Communications of the ACM (CACM)*, 21:120–126, 1978.

[21] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[22] Vipin Samar. Unified login with Pluggable Authentication Modules (PAM). In Clifford Neuman, editor, *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 1–10. ACM Press, 1996.

[23] Jennifer G. Steiner, B. Clifford Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, 1988.

[24] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[25] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the UNIX Security III Symposium*, pages 85–92, Baltimore, MY, USA, September 1992. USENIX Association.

[26] Tatu Ylonen. SSH—secure login connections over the Internet. In *Proc. of the USENIX Security Symposium*, pages 37–42, San Jose, California, 1996.