CITI Technical Report 02-2

Preventing Privilege Escalation

Niels Provos
provos@citi.umich.edu

Abstract

Many operating system services require special privileges to execute their tasks. A programming error in a privileged service may open the door to system compromise in form of unauthorized acquisition of privileges. In the worst case, a remote attacker may obtain superuser privileges. In this paper, we discuss the methodology and design of privilege separation, a generic approach that lets parts of an application run without special privileges. Programming errors occurring in these now unprivileged parts of the application can no longer be abused to gain unauthorized privileges. Privilege separation is orthogonal to capability or role-based security systems and may be used to enhance the security of such systems even further.

As a concrete example, the concept of privilege separation has been implemented in OpenSSH. We illustrate how separation of privileges reduces the amount of OpenSSH code that is executed with privileges. Privilege separation would have prevented past security vulnerabilities in OpenSSH including those that were unknown at the time of its implementation.

August 5, 2002

Center for Information Technology Integration
University of Michigan
535 West William Street
Ann Arbor, MI 48103-4943

Preventing Privilege Escalation

Niels Provos
Center for Information Technology Integration
University of Michigan

1 Introduction

Services running on computers connected to the Internet present a target for attackers to compromise their security. This can lead to unauthorized access of sensitive data or resources.

Services that require special privileges for their operation are critically sensitive. A programming error here may allow an attacker to obtain and abuse the special privileges.

The degree of the escalation depends on which privileges the attacker is authorized to hold and which privileges can be obtained in a successful attack. For example, a programming error that permits a user to gain extra privilege after successful authentication limits the degree of escalation because the user is already authorized to hold some privileges. On the other hand, a remote attacker gaining superuser privileges without any authentication presents a more severe escalation.

For services that are part of the critical Internet infrastructure is it particularly important to protect against programming errors. Sometimes these services need to retain special privilege for the lifetime of a session. For example, in SSH, the SSH daemon needs to know the private host key during re-keying to authenticate the key exchange. The daemon also needs to open new pseudo-terminals when the SSH client so requests. These operations require durable privileges as they can be requested at any time during the lifetime of a SSH connection. In current SSH implementations, therefore, an exploitable programming error allows an attacker to obtain superuser privileges.

Several approaches to help prevent security problems related to programming errors have been proposed. Among them are type-safe languages [18] and operating system mechanisms like protection domains [9]. However, these solutions do not apply to many existing applications as they are written in C to run on a generic Unix operating systems.

Instead, this paper discusses the methodology and design of *privilege separation*, a generic approach to limit the scope of programming bugs. The basic principle of privilege separation is to reduce the amount of code that runs with special privileges without affecting or limiting the functionality of the service. This reduces the opportunity for bugs in code that is executed with privileges. Ideally, the only consequence of an error in a privilege separated service is denial of service to the attacker himself.

Privilege separation also facilitates source code audits by reducing the amount of code that needs to be inspected initially. While all source code requires auditing, the size of code that is most critical to security decreases.

Privilege separation is instantiated by spawning unprivileged children from a privileged parent. To execute privileged operations, the unprivileged child requests a privileged operation from the privileged parent.

The principle of separating privileges applies to any privileged service on a Unix-like operating system. In this paper, we use OpenSSH as an example of a service whose privileges can be separated. We show that bugs in OpenSSH that led to system compromise are completely contained by privilege separation. Privilege separation requires small changes to existing code and incurs no noticeable performance penalty.

The rest of the paper is organized as follows. In Section 2, we discuss the principle of least privilege. We introduce the concept of privilege separation in Section 3 and describe a generic implementation for Unix operating system platforms. We explain the implementation of privilege separation in OpenSSH in Section 4. In Section 5, we discuss how privilege separation improves security in OpenSSH. We analyze its performance impact in Section 6. Section 7 describes related work. Finally, we conclude in Section 8.

2 Least Privilege

We refer to a *privilege* as a security attribute that is required for certain operations. Privileges are not unique and may be held by multiple entities.

The motivation for this effort is the principle of least privilege: every program and every user should operate using the least amount of privileges necessary to complete the job [16]. Applying the principle to application design limits unintended damage resulting from programming errors. Linden [11] suggests three approaches to application design that help prevent unanticipated consequences from such errors: defensive programming, language enforced protection, and protection mechanisms supported by the operating system.

The latter two approaches are not applicable to many Unix-like operating systems because they are developed in the C language which lacks type-safety or other protection enforcement. Though some systems have started to support non-executable stack pages which prevent many stack overflows from being exploitable, this mechanism is not available for most Unix platforms.

Furthermore, the Unix security model is very coarse. Process privileges are organized in a flat tree. At the root of the tree is the superuser and its leaves are the users of the system. The superuser has access to every process, whereas users may not access processes of other users. Privileges that are related to file system access have finer granularity because the system grants access based on the identity of the user and his group memberships. In general, privileged operations are executed via system calls in the Unix kernel, which differentiates mainly between the superuser and everyone else.

This leaves defensive programming, which attempts to prevent errors by checking the integrity of parameters and data structures at implementation, compile or run time. For example, defensive programming prevents buffer overflows by checking that the buffer is large enough to hold the data that is being copied into it. Improved library interfaces like *strlcpy* and *strlcat* help programmers avoid buffer overflows [13].

Nonetheless, for complex applications it is still inevitable that programming errors remain. Furthermore, even the most carefully written application can be affected by third-party libraries and modules that have not been developed with the same stringency. The likelihood of bugs is high, and an attacker will try to use those bugs to gain unauthorized privileges. Even if the principle of least privilege has been followed, an attacker may still gain those privileges that are necessary for the application to operate.

3 Privilege Separation

This section presents an approach called *privilege separation* which cleaves an application into privileged and unprivileged parts. Its philosophy is similar to the decomposition found in micro-kernels or in Unix command line tools. Privilege separation is orthogonal to other protection mechanisms that an operating system might support, *e.g.*, capabilities or protection domains. We describe an implementation of privilege separation that does not require special support from the operating system kernel and as such may be implemented on almost any Unix-like operating system.

The goal of privilege separation is to reduce the amount of code that runs with special privileges. We achieve this by splitting an application into two parts. One part that runs with privileges and the other that runs without them. We call the privileged part the *monitor* and the unprivileged part the *slave*. The slave has to ask the monitor to perform any operation that requires privileges. Before serving a request from the slave, the monitor first validates it. If the request is currently permitted, the monitor executes it and communicates the results back to the slave.

In order to separate the privileges in a service, it is necessary to identify the operations that require them. The number of such operations is usually small compared to the operations that can be executed without special privileges. Assuming a uniform distribution of programming errors, privilege separation reduces the number of programming errors that occur in a privileged code path. Furthermore, source code auditing efforts can be directed towards code that is executed with privileges which can further reduce the number of programming errors remaining in it.

Although errors in the unprivileged code path can not result in any immediate privilege escalation, it might still be possible to abuse them for other attacks like resource starvation. Such denial of service attacks are beyond the scope of this paper.

In the following, we explain the Unix mechanisms that allow us to implement a privilege separated service. Processes are protection domains in a Unix system. That means that one process can not access data in another process. To achieve privilege separation, we create two entities: a privileged parent process that acts as the monitor and an unprivileged child process that acts as the slave. The privileged parent can be modeled by a finite-state machine (FSM) that monitors the progress of the unprivileged child. The parent accepts requests from the child for actions that require privileges. The set of actions that are permitted changes over time and depends on the current state of the FSM. If the number of actions that require privileges is small, most of the application code is executed by the unprivileged child.

A privilege separated service can be in one of two different phases:

- Pre-Authentication Phase: A user has contacted a system service but is not yet authenticated. In this case, the unprivileged child has no process privileges and no rights to access the file system.

- Post-Authentication Phase: The user has successfully authenticated to the system. The child has the privileges of the user including file system access, but does not hold any other special privileges. However, special privileges are still required to create new pseudo-terminals or to perform other privileged operations. For those operations, the child has to request an action from the privileged parent.

The unprivileged child is created by changing its user identification (UID) and group identification (GID) to otherwise unused IDs. This is achieved by first starting a privileged monitor process. It forks a slave process. The first action that the slave performs is to change its UID and GID. As a result, it loses its process privileges. To prevent access to the file system, the child changes the root of its file system to an empty directory in which it is not allowed to create any files.

To enable slave requests to the monitor, we use interprocess communication (IPC). There are many different ways to allow communication between processes: pipes, shared memory, etc. In our case, we establish a socket between the two processes using the *socketpair* system call. The file descriptor is inherited by the forked child.

A slave may request different types of privileged operations from the monitor. We classify them depending on the result the slave expects to achieve: *Information*, *Capabilities*, or *Change of Identity*.

A child issues an informational request if acquiring the information requires privileges. The request starts with a 32-bit length field followed by an 8-bit number that determines the request type. In general, the monitor checks every request to see if it is allowed. It may also cache the request and result. In the pre-authentication phase, challenge-response authentication can be handled via informational requests. For example, the child first requests a challenge from the privileged monitor. After receiving the challenge, the child presents it to the user and requests authentication from the monitor by presenting the response to it. In this case, the monitor remembers the challenge that it created and verifies that the response matches. The result is either successful or unsuccessful authentication. In the case of OpenSSH, most privileged operations can be implemented with informational requests.

Ordinarily, the only capability available to a process in a Unix operating systems is a file descriptor. When a slave requests a capability, it expects to receive a file descriptor from the privileged monitor that it could not obtain itself. A good example of this is a service that provides a pseudo-terminal to an authenticated user. Creating a pseudo-terminal involves opening a device owned by the superuser and changing its ownership to the authenticated user, which requires special privileges.

```
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_len = CMSG_LEN(sizeof(int));
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmsg) = fd;
```

Figure 1: A file descriptor can be sent to another process by a special control message.

Modern Unix operating systems provide a mechanism called *file descriptor passing*. File descriptor passing allows one process to give access to an open file to another process [17]. This is achieved by sending a control message containing the file descriptor to the other process; see Figure 1. When the message is received, the operating system creates a matching file descriptor in the file table of the receiving process that permits access to the sender's file. We implement a capability request by passing a file descriptor over the socket used for the informational requests. The capability request is an informational request in which the slave expects the monitor to answer with a control message containing the passed file descriptor.

The change of identity request is the most difficult to implement. The request is usually issued when a service changes from the pre-authentication to the post-authentication phase. After authentication, the service wants to obtain the privileges of the authenticated user. Unix operating systems provide no mechanism to change the user identity a process is associated with unless the process has superuser privileges. However, in our case, the process that wants to change its identity does not have such privileges.

One way to effect a change of identity is to terminate the slave process and ask the monitor to create a new process that can then change its UID and GID to the desired identities. By terminating the child process all the state that has been created during its life time is lost. Normally a meaningful continuation of the session is not possible without retaining the state of the slave process. We solve this problem by exporting all state of the unprivileged child process back to the monitor.

Exporting all state is not easy. For global structures, we use XDR [12] like data marshaling which allows us to send all data contained in a structure to the monitor. The data is unpacked by the newly forked child process. This prevents any data corruption in the exported data to affect the privileged monitor in any way.

For structures that are allocated dynamically, *e.g.*, via *malloc*, data export is more difficult. We solve this problem by providing memory allocation from shared memory. As a result, data stored in dynamically allocated memory is also available in the address space of the privileged monitor. Figure 2 shows the interface to shared memory allocator.

```
mm_master_t *mm_create(mm_master_t *, size_t);
void mm_destroy(mm_master_t *);
void *mm_malloc(mm_master_t *, size_t);
void mm_free(mm_master_t *, void *);
void mm_share_sync(mm_master_t **, mm_master_t **);
```

Figure 2: These functions represent the interface for shared memory allocation. Using them allows the export of data from a child process to its parent.

The two functions *mm_create* and *mm_share_sync* are responsible for permitting a complete export of dynamically allocated memory. The mm_create function creates a shared address space of the specified size. There are several ways to implement shared memory, we use anonymous memory maps. The returned value is a pointer to a *mm_master* structure that keeps track of allocated memory. It is used as parameter in subsequent calls to *mm_malloc* and *mm_free*. Every call to those two functions may result in allocation of additional memory for state that keeps track of free or allocated memory in the shared address space. Usually, that memory is allocated with libc's malloc function. However, the first argument to the mm_create function may be a pointer to another shared address space. In that case, the memory manager allocates space for additional state from the passed shared address space.

Figure 3 shows an overview of how allocation in the shared address space proceeds. We create two shared address spaces: *back* and *mm*. The address space *mm* uses *back* to allocate state information. When the child wants to change its identity, it exits and the thread of execution continues in the parent. The parent has access to all the data that was allocated in the child. However, one problem remains. The shared address space *back* uses libc's malloc that allocated memory in the child's address space to keep track of its state. If this information is lost when the child process exits, then subsequent calls to mm_malloc or mm_free fail. To solve the problem, the parent calls the mm_share_sync function which recreates the state information in the

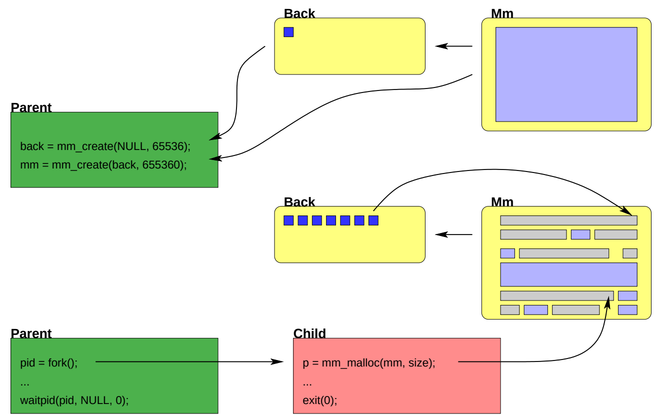

Figure 3: The shared memory allocator is backed by another shared address space. This permits the complete export of state that was allocated dynamically in the child.

shared address space *back*. Afterwards, freeing and allocating memory proceeds without any problems.

We use shared memory and XDR-like data marshaling to export all state from the child to the parent. After the child process exports its state and terminates, the parent creates a new child process. The new process changes to the desired UID and GID and then imports the exported state. This effects a change of identity in the slave that preserves state information.

4 Separating Privileges in OpenSSH

In this section, we explain how to apply the concept of privilege separation to OpenSSH, a free implementation of the SSH protocols. OpenSSH provides secure remote login across the Internet. OpenSSH supports protocol versions one and two; we restrict our explanation of privilege separation to the latter. The procedure is very similar for protocol one.

When the SSH daemon starts, it binds a socket to port 22 and waits for new connections. Every new connection is handled by a forked process. The forked process needs to retain superuser privileges throughout its lifetime to create new pseudo terminals for the user, to authenticate key exchanges when cryptographic keys are replaced with new ones, to clean up pseudo terminals when the SSH session ends, to create a process with the privileges of the authenticated user, etc.

For privilege separation, the forked process acts as the monitor and forks a slave process that drops all its privileges and starts accepting data from the established connection. The monitor now waits for requests from the slave; see Figure 4. Requests that are permitted in the pre-authentication phase are shown in

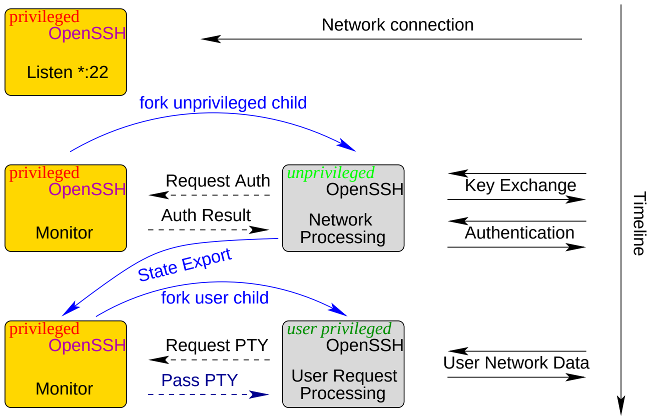

Figure 4: Overview of privilege separation in OpenSSH.

Figure 5. If the child issues a request that is not permitted, the privileged monitor terminates.

First, we identify the actions that require privileges in OpenSSH and show which request types can fulfill them. In the following, we describe the privileged requests for the pre-authentication phase:

- Key Exchange: SSH v2 supports two different key exchanges. One of them is the Diffie-Hellman Group Exchange which allows the client to request a group of a certain size from the server [8]. To find an appropriate group the server consults the */etc/moduli* file. However, because the slave has no privileges to access the file system, it can not open the file itself. For that reason, it issues an informational request to the monitor. The monitor returns a suitable group after consulting the moduli file. The returned group is used by the slave for the key exchange. As seen in Figure 5, the slave may issue this request only once.

- Authenticated Key Exchange: To prevent man-in-the-middle attacks, the key exchange is authenticated. That means that the SSH client requires cryptographic proof of the server identity. At the beginning of the SSH protocol, the server sends its public key to the client for verification. As the public key is public, the slave knows it and no special request is required. However, the slave needs to ask the monitor to authenticate the key exchange by signing a cryptographic hash of all values that have been exchanged between the client and the server. The signature is obtained by an informational request.

- User Validation: After successful key exchange, all communication is encrypted and the SSH client informs the server about the identity of the user who wants to authenticate to the system. At this point, the server decides if the user name is valid and allowed to login. If it is invalid, the protocol proceeds but all authentication attempts from the client fail. As the slave can not access the password database, it has to issue an informational request to the server. The server caches the user name and reports back to the slave if the name is valid.

- Password Authentication: Several methods can be used to authenticate the user. For password authentication, the SSH client needs to send a correct login and password to the server. Once again, the unprivileged slave can not access the password database, so it asks the monitor to verify the password. The monitor informs the slave if the authentication succeeds or fails. If it succeeds, the pre-authentication phase ends.

- Public Key Authentication: Public Key Authentication is similar to password authentication. If it is successful, the pre-authentication phase ends. However, two informational requests are required to use public keys for authentication. The first request allows the slave to determine if a public key presented by the client may be used for authentication. The second request determines if the signature returned by the client is valid and signs the correct data. A valid signature results in successful authentication.

At any time, the number of requests that the slave may issue are limited by the state machine. When the monitor starts, the slave may issue only the first two requests in Figure 5. After the key exchange has finished, the only valid request is for user validation. After validating the user, all authentication requests are permitted. The motivation for keeping the number of valid requests small is to reduce the attack vector available to an intruder who has compromised the slave process.

All requests up to this point have been informational. The pre-authentication phase ends with successful authentication as determined by the monitor. At this point, the slave needs to change its identity to that of the authenticated user. As a result, the slave obtains all privileges of the user, but no other privileges. We achieve this with a change of identity request.

The monitor receives the state of the slave process and waits for it to exit. The state consists of the following: the encryption and authentication algorithms including their secret keys, sequence counters for in-

```
  struct mon_table mon_dispatch_proto20[] = {
    {MONITOR_REQ_MODULI, MON_ONCE, mm_answer_moduli},
    {MONITOR_REQ_SIGN, MON_ONCE, mm_answer_sign},
    {MONITOR_REQ_PWNAM, MON_ONCE, mm_answer_pwnamallow},
    {MONITOR_REQ_AUTHSERV, MON_ONCE, mm_answer_authserv},
    {MONITOR_REQ_AUTHPASSWORD, MON_AUTH, mm_answer_authpassword},
[...]
    {MONITOR_REQ_KEYALLOWED, MON_ISAUTH, mm_answer_keyallowed},
    {MONITOR_REQ_KEYVERIFY, MON_AUTH, mm_answer_keyverify},
    {0, 0, NULL}
  };
```

Figure 5: The table describes the valid requests that the slave may send to the monitor in the pre-authentication phase for SSH protocol version two.

coming and outgoing packets, buffered network data and the compression state.

Exporting the cryptographic key material is uncomplicated. The main problem is exporting the compression state. The SSH protocols use the *zlib* compression format [5, 6] which treats network data as a stream and not packet by packet. Treating it as a stream allows zlib to improve its dictionary with increasing amount of compressed data. On the other hand, it also means that compression in the server can not be stopped and then restarted as the client uses a dictionary that depends on all the preceding data. Fortunately, zlib provides hooks for user supplied memory management functions. We provide it with functions that use *mm_malloc* and *mm_free* as back end. After the child exits, the monitor only needs to call *mm_share_sync* to import the compression state.

The monitor forks a new process that then changes its process identification to that of the authenticated user. The slave process obtains all the privileges of the authenticated user. At this point, we enter the post-authentication phase which requires only a few privileged operations. They are as follows:

- Key Exchange: In SSH protocol version two, it is possible to renew cryptographic keys. This requires a new key exchange, so just as in the pre-authentication phase, the monitor chooses a suitable group for the Diffie-Hellman key exchange and signs for authentication.

- Pseudo Terminal Creation: After authentication, the user requires a pseudo terminal whose creation requires superuser privileges. For a Unix application, a pseudo terminal is just a file descriptor. The slave issues a capability request to the monitor. The monitor creates the terminal and passes the corresponding file descriptor to the child process. An informational request suffices when the slave wants to close the pseudo terminal.

Observe that the majority of all privileged operations can be implemented with informational requests. In fact, some degree of privilege separation is possible if neither capability nor change of identity requests are available. If the operating system does not support file descriptor passing, privilege separation perforce ends after the pre-authentication phase. To fully support the change of identify request shared memory is required. Without shared memory, the compression state can not be exported without rewriting *zlib*. Nonetheless, systems that do not support shared memory can disable compression and still benefit from privilege separation.

The changes to the existing OpenSSH sources are small. About 950 lines of the 44,000 existing lines of source code or about 2% were changed. Many of the changes are uncomplicated:

```
- authok = auth_password(authctxt, pwd);
+ authok = PRIVSEP(auth_password(authctxt, pwd);
```

The new code that implements the monitor and the data marshaling amounts to about three thousand lines of source code, or about seven percent increase in the size of the existing sources. While support for privilege separation increases the source code size, it actually reduces its complexity. Privilege separation requires clean and well abstracted subsystem interfaces so that their privileged sections can run in a different process context. During the implementation, the interfaces for several subsystems had to be improved to facilitate their separation. As a result, the source code is cleaner and less complex.

5 Security Analysis

To measure the effectiveness of privilege separation in OpenSSH, we analyse how it would have affected security problems in the past. We do not discuss problems of cryptographic primitives, our assumption is that the employed cryptography is secure.

The *SSH-1 Daemon CRC32 Compensation Attack Detector Vulnerability* permits an attacker to gain superuser privileges remotely without authenticating to the systems [19]. The problem is caused by an integer overflow in a function that processes network packets. With privilege separation, the function is executed without any privileges making it impossible for an attacker to directly compromise the system.

Similarly, the off-by-one error in OpenSSH's channel code allows an attacker to gain superuser privileges after authenticating to the system [14]. With privilege separation, the process has only the privileges of the authenticated user. As a result, an attacker can not obtain system privileges in this case either.

A security problem in the external *zlib* compression library was found that might allow a remote attacker to gain superuser privileges without any authentication [2]. As this problem occurs in a third-party library, no audit of the OpenSSH source code itself can find it. Privilege separation prevents a system compromise in this case, too.

At the time of this writing, more security problems were found in OpenSSH. A bug in the Kerberos ticket passing functions allowed an authenticated user to gain superuser rights. A more severe problem in code for challenge-response authentication allowed a remote attacker to obtain superuser privileges without any authentication [3]. Privilege separation already part of OpenSSH prevents both of these problems and is mentioned in the CERT advisory as a solution.

These examples demonstrate that privilege separation has the potential to contain unknown security problems in the future. It prevents the problems discussed above.

Subsystem	Lines of Code	Percentage
Unprivileged	**10360**	**75.27%**
Ciphers	267	1.93%
Packet Handling	1093	7.94%
Miscellaneous	7944	57.71%
Privsep Interface	1056	7.67%
Privileged	**3403**	**24.73%**
Authentication	803	5.84%
Miscellaneous	1700	12.35%
Monitor	900	6.54%

Table 1: Number of source code lines that are executed with and without privileges.

After privilege separation, three quarters of the source code are executed without privileges as shown in Table 1. The numbers do not include code from third-party libraries that runs unprivileged now, too. If we assume that programming errors are distributed fairly uniformly, we can estimate the increase of security by counting the number of source code lines that are now executed without privileges. We argue that seventy five percent of all programming errors will not result in privilege escalation or that only twenty five percent of the source code requires auditing.

A programming error in the slave process might allow an attacker to gain complete control over it. One way an attacker may try to gain additional privileges is to attack the interface between the privileged monitor and the slave. The attacker could send badly formatted requests in the hope of exploiting programming errors in the monitor. For that reason, it is important to carefully audit the interface to the monitor. In the current implementation, the number of valid requests is small and any request detected as invalid causes the privileged monitor to terminate.

Nonetheless, there may be other ways that an attacker might try to harm the system. She might try to starve the resources of the system by forking new processes or by running very time intensive computations. As a result, the system might become unusable. The effect of such an attack can be mitigated by placing process limits on the slave process. For example, we can limit the number of file descriptors the slave may open and the number of processes it is allowed to fork. The monitor may also watch other resource utilization like CPU time and terminate the process if a certain threshold is reached. While an attacker controlling the unprivileged child has no access to the file system, he may use other system calls to continue the attack. For example, the unprivileged child can initiate local network connections and potentially abuse trust relations based on IP addresses. We may further restrict the child's ability to access the system by employing external policy enforcement mechanisms like Systrace [15].

6 Performance Analysis

To analyze the performance of privilege separated OpenSSH, we measure the execution time for several different operations between normal OpenSSH and the privileged separated version. We conduct the measurements on a 1.13 GHz Pentium III laptop with all data in the memory cache.

The first test measures the time it takes to login using public key authentication. We measure the time

Test	Normal	Privsep
Login		
- compressed	0.775s ± 0.0071s	0.777s ± 0.0067s
- uncompressed	0.767s ± 0.0106s	0.774s ± 0.0097s
Data Transfer		
- compressed	4.229s ± 0.0373s	4.243s ± 0.0411s
- uncompressed	1.989s ± 0.0223s	1.994s ± 0.0143s

Table 2: Performance comparison between normal OpenSSH and privilege separated OpenSSH.

with compression enabled and without compression. The next two tests measure the data transfer time of a 10 MB file filled with random data, with compression enabled, and without compression. The results are shown in Table 2. It is evident that privilege separated OpenSSH does not penalize performance.

7 Related Work

The principle of least privilege has guided application developers for a long time. There are several applications that make use of privilege separation as we discuss below. The main difference in this research is the degree and completeness of the separation.

Carson demonstrates how to reduce the number of privileges that are required in the *Sendmail* mail system [1]. His design follows the principle of least privilege. While Sendmail is a good example, the degrees of privilege separation demonstrated in OpenSSH are much more extensive. For example, we show how to change the effective UID and how to retain privileges securely for the whole duration of the session.

Evans *very secure* FTP daemon uses privilege separation to limit the effect of programming errors [7]. He uses informational and capability requests in his implementation. His work is very similar to the implementation of privilege separation in OpenSSH, but not as extensive and less generic.

Solar Designer uses a process approach to switch privileges in his *Owl* Linux distribution [4]. His POP3 daemon *popa3d* forks processes that execute protocol operations with lower privileges and communicate results back to the parent. The interaction between parent and child is based completely on informational requests.

Separating the privileges of an application causes a decomposition into subsystems with well defined functionality. This is similar to the design and functionality of a μ-kernel where subsystems have to follow the principle of independence and integrity [10]. For a privilege separated application, independence and integrity are realized by multiple processes that have separate address spaces and communicate via IPC.

8 Conclusion

Programming errors in privileged services can result in system compromise allowing an attacker to gain unauthorized privileges.

Privilege separation as a concept that allows the majority of an application to run without any privileges at all. Programming errors in the unprivileged part of the application can not lead to privilege escalation.

As a proof of concept, we implemented privilege separation in OpenSSH and show that past errors that allowed system compromise would have been contained with privilege separation.

There is no performance penalty when running OpenSSH with privilege separation enabled.

9 Acknowledgments

I thank Markus Friedl for his help with the final implementation of privilege separation in OpenSSH and Peter Honeyman for helpful suggestions and comments.

References

[1] Mark E. Carson. Sendmail without the Superuser. In *Proceedings of the 4th USENIX Security Symposium*, October 1993. 9

[2] CERT/CC. CERT Advisory CA-2002-07 Double Free Bug in zlib Compression Library. `http://www.cert.org/advisories/CA-2002-07.html`, March 2002. 8

[3] CERT/CC. CERT Advisory CA-2002-18 OpenSSH Vulnerabilities in Challenge Response Handling. `http://www.cert.org/advisories/CA-2002-18.html`, June 2002. 8

[4] Solar Designer. Design Goals for popa3d. `http://www.openwall.com/popa3d/DESIGN`. 9

[5] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, 1996. 7

[6] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950, 1996. 7

[7] Chris Evans. Comments on the Overall Architecture of Vsftpd, from a Security Standpoint. `http://vsftpd.beasts.org/`, February 2001. 9

[8] Markus Friedl, Niels Provos, and William A. Simpson. Diffie-Hellman Group Exchange for the SSH Transport

Layer Protocol. Internet Draft, January 2002. Work in progress. 6

[9] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997. 2

[10] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the Symposium on Operating Systems Principles*, pages 237–250, 1995. 9

[11] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4):409–445, 1976. 3

[12] Sun Microsystems. XDR: External Data Representation. RFC 1014, June 1987. 4

[13] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *Proceedings of the 1999 USENIX Technical Conference, FREENIX track*, June 1999. 3

[14] Joost Pol. OpenSSH Channel Code Off-By-One Vulnerability. `http://online.securityfocus.com/bid/4241`, March 2002. 8

[15] Niels Provos. Systrace - Interactive Policy Generation for System Calls. `http://www.citi.umich.edu/u/provos/systrace/`, May 2002. 8

[16] Jerome H. Saltzer. Protection and the Control of Information in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. 3

[17] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992. 4

[18] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. *16h Symposium on Operating System Principles*, pages 116–128, 1997. 2

[19] Michal Zalewski. Remote Vulnerability in SSH Daemon CRC32 Compensation Attack Detector. `http://razor.bindview.com/publish/advisories/adv_ssh1crc.html`, February 2001. 8