USENIX Association

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Privman: A Library for Partitioning Applications

Douglas Kilpatrick
Network Associates Laboratories

Abstract

Writing secure, trusted software for UNIX platforms is difficult. There are a number of approaches to enabling more secure development, but it is apparent that the current set of solutions are neither achieving acceptance nor having sufficient impact. In this paper, we introduce a library to address a particularly difficult problem in secure code development: partitioning processes to isolate privileges in trusted code.

Privilege separation is a technique that isolates trusted code, therefore reducing the amount of code that needs to be carefully audited. While the technique is not new, it is not widely used due to difficulties of implementation. We present Privman, a library that makes privilege separation easy. The primary benefit of the Privman library is a systematic, reusable framework and library for developing partitioned applications. We demonstrate the feasibility of the approach by applying it to two real systems, thttpd and WU-FTPD.

1. Introduction

According to the SANS list of critical security vulnerabilities [SAN02], simple programming errors such as buffer overflows cause approximately 40% of the top vulnerabilities. Despite a large collection of knowledge on writing secure software, programmers still make the same mistakes today as 10 years ago, and the same types of services cause most of the problems.

Standard UNIX systems do not handily support the concept of least-privilege, instead granting all privileges to superuser (root), and none otherwise. While specific systems may have some least-privilege features [IEE97], the relevant APIs have not achieved sufficiently widespread usage, and therefore cross-platform applications are not able to take advantage of them. Although the widespread use of techniques like those in StackGuard [COW98a] would protect against these common buffer overflows, even the strongest compiler techniques are useless against application design errors.

There are a number of reasons for the lack of progress in developing secure software. The methodologies for developing secure software are not used, as general application developers with little security awareness write most software. Additionally, the existing body of knowledge commonly available to developers consists of a list of missteps, not positive guidelines [KIE02a]. Even when the developers know of the issues, they often drop security concerns in the rush to add functionality.

Finally, the secure operating system features that would allow secure software to be written more easily have not achieved widespread deployment for a number of different reasons [COW98], including deployment costs and more general networking/interoperability issues.

Secure programming practices have significant value, but most developers are not security experts and therefore cannot take full advantage of those practices. This trend will not change. Therefore, we seek to lower the barriers of entry for writing secure system software.

Our contribution towards this goal involves facilitating a specific technique for writing secure software: partitioning applications between security-specific (trusted) code and non-security-specific (untrusted) code. By validating requests in the privileged code, an application can be limited to the required set of privileges.

2. Problem: Partitioning Applications

Software is, by its very nature, extremely sensitive to mistakes [BRO75]. Privileged software is no different. In all common operating systems, software with the ability to perform *any* privileged operation also has the ability to perform *all* privileged operations. Therefore, this software must be trusted to not misbehave, even when under attack from untrusted sources. When writing trusted software, a single mistake can compromise the entire system.

An important defensive programming technique is encapsulating the security-sensitive components within small, simple components. These components can then be more easily verified. This pattern [KIE02b] of separation provides additional assurance when requests for privileged operation have to be validated by the secure component. By separating the security sensitive components of the application in a different process from the bug-prone components, mistakes in the majority of the application will not and cannot result in total compromise of the entire application. Hence, attackers would be unable to compromise the system.

The most common example of this pattern of enforced separation is the kernel/user-space split of the major operating systems, but this is also the pattern of any trusted computing base (TCB). This split can be applied to user-space software by splitting the software among processes [VIE98].

On UNIX systems, many exercises of privilege fundamentally exist as file accesses. For example, validating a user's authentication normally requires read access to the /etc/passwd and /etc/shadow files. UNIX systems validate file-access privilege upon `open(2)` only, and therefore consider file descriptors to be effectively access tokens.

The standard POSIX APIs allows file descriptors to be passed between processes via UNIX domain sockets (pipes)[i][POS00]. This API therefore allows software to pass privilege tokens between processes. Programs can proxy many useful operations by passing file descriptors between processes.

Several projects have used this POSIX capability to split their process in a one-off and ad-hoc way, including recent versions of the ubiquitous OpenSSH daemon [PRO02]. Several other projects have used other forms of process compartmentalization, including qmail [NEL02] and postfix [BLU01].

Traditionally, it has been very difficult to retrofit this design to existing applications. Instead, almost all such applications have been written from the ground up around this design.

The Privman library seeks to overcome this limitation by allowing developers to easily adapt existing applications. The Privman library provides a systematic and reusable approach to enabling development of a partitioned application.

2.1. Constraints

For developers to adopt our approach, it must fit into the "real world" of application development. This places several constraints upon any solution.

- The framework must exist in a form convenient for development. We believe this requires the system to be available as a library.
- The framework must be portable between most UNIX systems.
- The framework cannot rely upon custom kernel changes or custom system libraries.
- The library API should be in terms of existing, well-understood APIs, simplifying any porting efforts.

The barriers to changing a piece of the core infrastructure of a distribution (like the compiler or the fundamental nature of the security policy of the kernel) are high. In contrast, the barriers to small changes in a specific daemon are low. (A minimal install of Red Hat Linux version 7.2 includes approximately 70 packages of libraries, most of which are only used for one or two different pieces of software.) We expect, therefore, that a well-designed library may lure developers of system software.

2.2. Related Work

Many systems have approached the problem of least privilege, even under the constraints listed above; OpenSSH, for example, currently uses a framework very similar to Privman to protect systems from possible bugs in the ssh daemon. However, the privilege-separation framework in OpenSSH is application-specific in many ways. The majority of the privileged operations in privilege separated OpenSSH are SSH-specific operations, and the framework assumes that application-specific operations are the norm. Privman does not make that assumption, and is more useful for general applications.

Other projects have modified the kernel on systems to enforce application-specific security policies. Systrace[PRO03], GSWTK[FRA99], and Tron[BER95], mediate at the system call layer to enforce their policy. We avoided this approach to improve portability across systems. SubOS[IOA02] uses a modified kernel to associate permissions with files, and restrict the permissions of programs that open those files. While using a modified kernel is faster and offers higher levels of granularity, prior approaches have not achieved widespread adoption, although Systrace may break through.

Similar to the kernel-based sandbox systems are the systems that enforce a policy on running applications by using system-specific debugging interfaces. Janus [GOL98] and MAPbox [ACH00] are two examples of this approach. These systems are less portable across platforms. They can also suffer performance issues, as each system call requires the intervention of a user-space application. Privman may be slightly quicker, as only those calls that require privilege need to be passed to the Privman server.

None of these sandbox systems require modification of the running applications. On the other hand, application-specific systems do not require custom kernel builds or non-standard debugging APIs. In all cases, developing a correct and strict policy is the primary difficulty. Systrace, unlike GSTWK, can automatically build policies for applications running under the system, but the policy-building process for Privman and most of sandboxing systems is still fairly difficult and manual.

Various projects such as qmail and vsftp have developed application-specific methods of privilege separation, normally involving process separation and clean interfaces between processes. These approaches are used only in the creation of new software, as it is impractical to adapt existing software to this design.

Several projects restrict the privileges given to daemons by placing the daemons in some form of "sandbox". FreeBSD has its `jail(2)` system call, Linux has User-Mode-Linux[UML03], and many systems have some form of total-system virtualization. The Java VM's security manager may be the most famous sandbox. Privman is more portable across common systems than most of these sandboxing techniques, and unlike the Java VM, supports C and C++.

3. Solution: The Privman Library

We present a library, called Privman, which simplifies the task of partitioning applications for a particular class of applications, privileged UNIX daemons.

Programs that use Privman split themselves into two processes: a privilege server, and the main application. The main application gives up all privilege, and asks the privilege server to perform any privileged operations on its behalf.

```
int main(int argc, char *argv[])
{
    char buf[4096];
    int i;
    priv_init("mycat");

    for (i=1; i < argc; ++i) {
    /* Privileged use of "fopen" */
        FILE *f=priv_fopen(argv[1],"r");
        while(n=fread(buf,1,1024,f) >0)
            fwrite(buf, n, 1, stdout);
        fclose(f);
    }
    exit(0);
}
```

Listing 1

```
struct pam_conv conv = {
    misc_conv,
    NULL
};
int main(int argc, char *argv[])
{
    pam_handle_t   *pamh = NULL;
    const char     *user = argv[1]
    priv_init("check_user");
    if (priv_pam_start("login", user,
            &conv, &pamh) != PAM_SUCCESS)
        goto failed;
    if ( priv_pam_authenticate(pamh, 0)
            != PAM_SUCCESS)
        goto failed;
    if ( priv_pam_acct_mgmt(pamh, 0)
            != PAM_SUCCESS)
        goto failed;
    fprintf("user %s authenticated "
        "%s\n",user);
    return EXIT_SUCCESS:
failed:
    fprintf("could not authenticate %s\n",
        user);
    return EXIT_FAILURE:
}
```

Listing 2

Privman uses an application-specific configuration file to limit the available privileges. The policy can limit a program to opening specific files, binding to specific ports, or otherwise limit access to the privileged operations.

The Privman library makes implementing privilege separation much easier, by providing standard C library equivalent functions for many operations that traditionally need privilege. The library supports several file-access methods, PAM authentication, `bind()`, and `daemon()`. The library supports several "`<foo>_as()`" methods to enable changes in the effective unprivileged user.

By limiting the amount of active code that runs with privilege, the amount of code requiring serious audit also shrinks. This should improve overall system security by increasing the assurance of the system.

3.1. Usage of the Library

The library has a very straightforward API at its core. The process starts by calling "`priv_init()`", and then calls various "`priv_*()`" methods when it wants to perform privileged operations. For example, a variant of `cat` that uses the library to read otherwise unreadable files might look like listing 1 (Privman specific parts in bold). Similarly, a stripped down program that authenticated a user is shown in listing 2.

As shown here, not all requests handled by the library are file-related. The Privman libraries can manage any request that can be proxied (in the case of PAM, by invoking input functions in the context of the unprivileged process).

The Privman library handles two types of extensions. Info functions return a string, and capability functions return a file descriptor. Both types take an array of `char *`. To register an extension function, the program calls `priv_register_{cap,info}_fn()` before it calls `priv_init()`. (Allowing the process to add extension methods after priv_init() is essentially allowing the process to run arbitrary code as root at any time.) The register functions return a handle which is used as an argument to later calls to the invoke functions. If a program needs to transition to a different user, it uses any of the "`_as()`" methods. All the "as" methods take a user name and a chroot jail, then spawn a second process to continue execution as the specified user.

The complete list of supported APIs is presented in Appendix 1.

3.2. Policy

Every Privman managed application has a config file, which in the case of a logfile review program might look like:

```
open_ro {
        # Anything under /var/log
        /var/log/*
}
```

or in the "check_user" case might look like:

```
# simple app.  Only needs PAM.
auth true
```

The proper configuration file is critical to successfully partitioning a process. Obviously, if the privilege manager automatically responds to any request, then Privman would only provide an illusion of security. The attacker would simply rewrite shell code to invoke the privilege manager instead of directly attacking the system. Instead, Privman relies on the configuration file to specify tight constraints on the allowable actions of a client.

This has the extra benefit of expressing the security policy openly, instead of leaving it buried in the code.

The configuration file should be written tightly enough to allow the process its privileged operations but nothing else, *i.e.*, the policy should follow the least privilege principle. For example, the following is the configuration for a simple network daemon.

```
# echo daemon.  The app is allowed
# to bind to a low port: 7
# and to write to a log file
bind echo
open_ao {
        /var/log/myecho.log
}
fork true
```

The grammar for the policy file is presented in Appendix 2.

3.3. Implementation

The Privman server performs all the privileged operations. When the application invokes a privileged operation, the library marshals the arguments, sends them over a UNIX pipe to the Privman server, and the operation is invoked there. Consequently, operations that do not actually need privilege may need to be executed by the Privman server. As an example, only `pam_authenticate()` really needs privilege to execute, but the Privman library proxies all of the PAM methods so that the state in the libpam library is consistent.

The Privman server is created from the main process during the call to `priv_init()`. The process, when it returns from `priv_init()`, is actually a child process forked off from the Privman server.

This design causes the library to export more methods than might be expected. For example, the `daemon()` call, which detaches a program from the controlling terminal, does not need privilege. However, any shell waiting on the daemon to exit will be waiting on the Privman server. The Privman server, then, needs to call `daemon()` to fully detach from the controlling terminal.

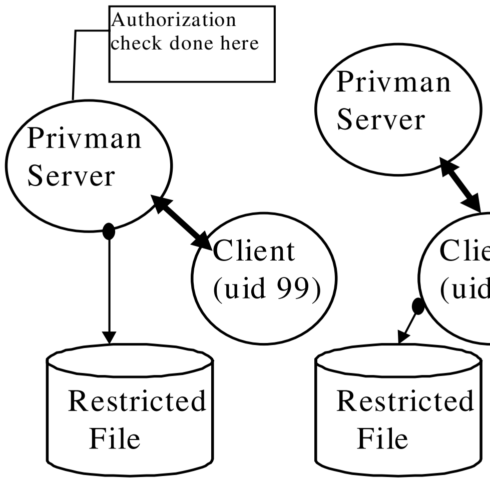

Figure 1

Figure 1 represents the program in listing 1 after it opens the file and starts to read. The Privman server passes the file descriptor to the Privman client, and the client is now reading freely, without involving the server for read calls.

The Privman server constitutes of very little code, only approximately 1400 lines (including comments) in the Privman 0.9.1 release. The equivalent code from the application-specific OpenSSH privilege separation framework is approximately 1500 lines[ii]. By limiting the amount of code running with privilege, we also limit the amount of code that requires serious auditing.

4. Security Properties

Assuming we code the Privman libraries correctly, programs that use Privman are incapable of performing privileged operations not allowed by the policy, even if the program's logic is compromised by an attacker. However, if the policy for an application is sufficiently permissive, attackers may be able to achieve their goals inside the constraints of the policy.

Generating a correct and minimal policy is difficult. Automatically generated policies, like in Systrace, can provide a quick first estimate of least privilege, but do not attempt to determine if an application is using privilege it does not actually need.

We have made no attempt to formally prove the correctness of Privman. The small size of the Privman server allows for careful auditing of the privileged code. Additionally, Privman is fail-closed. Whenever Privman detects error conditions, it makes no attempt to recover. Instead, the privilege server exits after logging the error message to the system's logs. Without a privilege server, the application is unable to perform any further privileged operations.[iii]

The largest area of vulnerability involves the protocol for communication between the privilege server and the client. This code runs with privilege and also processes incoming requests; the code is therefore vulnerable to malicious input. Fortunately, the amount of code required to process the incoming request is extremely small.

Additionally, the policy language itself should be carefully analyzed to make sure that policies can express least privilege.

5. Open Issues

The majority of issues stem from the decision to run the unprivileged portions of the program in a child process. By changing around the processes, we break reasonable assumptions of state inheritance in fork(). Some assumptions about the process identifier (pid) of the active process are also broken.

5.1. API Design Issues

The work, as originally proposed, involved a single Privman server per system[iv]. The single server would be contacted by all managed clients, and it would perform privileged operations on behalf of the entire system.

This design would allow programs to start execution without having any real pre-existing privilege outside of an ability to authenticate themselves with the privilege manager. Unfortunately, on stock UNIX systems, there is no practical way to verify the identity of a process[v].

Without that verification, this design would invite identity attacks, where an attacker would attempt to spoof the identity of a permitted privileged process. Rather than try to solve this authentication problem, we decided to leverage the pre-existing source of privilege: the fact that a process is already running in a privileged state.

The Privman managed process starts `main()` with heightened traditional (root) permissions. The `priv_init()` call then divides the process into two separate processes: an unprivileged child that runs the original program, and a parent that becomes the Privman server.

From the perspective of the original process, after the `priv_init()` call the process is no longer running as a privileged process and is instead running as the "nobody" user[vi]. Any privileged operations must be proxied and validated by the Privman server.

By splitting a process in this manner, we simplified the design of the Privman server, decreasing the probability of serious coding flaws.

Our approach has significant drawbacks when compared to traditional Mandatory Access Control-style secure systems. For example, we are unable to handle any concept of revocation. We are unable to handle permissions at granularity smaller than file access (or other mediated call). In addition, not all security decisions on a UNIX system are in terms of files or file descriptors.

In particular, there are a small number of calls that change the security context of the current process. One obvious example is `chroot()`, but the list also includes `limit()` and the very important `setuid()` family. Much of the software in question may need to make these changes to its security context, but our design makes accommodating this software difficult.

Obviously, other projects that use this type of process split have run into similar issues. We chose to model our approach on the OpenSSH solution [PRO02].

In the OpenSSH solution, the client process packages up state and sends it back to the parent. The parent then creates a second child process with the desired security context and state, and allows that child to continue execution in that context.

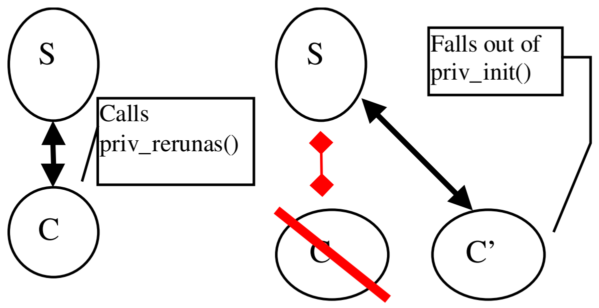

Figure 2

The problems of packaging up and managing state are considerable when attempting to retrofit the Privman library to preexisting applications. One approach might be to define a new `malloc()` in terms of a shared memory segment to help automate the process. This approach makes sharing large quantities of state easy, but at the expense of a large amount of trusted code. Defining a new memory allocation function opens a significant category of security problems and at the same time fails to handle global memory. We chose a different mechanism, one which makes sharing large amounts of state more difficult and requires more thought, but minimizes the trusted code.

Most applications needing to change their user id or other security state tend to follow a similar pattern: they authenticate as a user, and then they execute the bulk of the program as that user. Most of the state built up during the authentication stage is incidental, and can be safely ignored. Therefore, the amount of state that needs to be moved between processes is minimized.

Our library provides a "`rerunas()`" method. This method causes the Privman server process to re-execute "`priv_init()`" The Privman server process forks a second child, and the second child becomes the specified user. It then returns from `priv_init()` with the program's state being basically the same as when `priv_init()` was originally called. To distinguish between the original case, and the rerunas case, the calling process can supply a function pointer and an arguments string vector. The function will be invoked in the context of the new child, allowing the child to set up global state so that the proper code paths will be followed. The calling process will exit, transferring control of the application logic to the new child process.

This method allowed the port of wu-ftpd to support non-anonymous users easily. Tracking all the state that changed during execution is equivalent to process migration. Instead, we simply changed wu-ftpd so that it could run as if the user had already been authenticated. The structural changes to wu-ftpd were minimal.

5.2. Security weaknesses

One of the advantages to performing process partitioning manually is a heightened ability to match the policy to the application's specific needs. For example, the OpenSSH daemon knows that certain privileged requests can only happen once and so only allows them once.

The Privman library is unable to know the security requirements of an application to the same degree. The configuration, while more detailed than traditional Unix system privilege, may allow an overly broad amount of privilege to an attacker. As an example, the policy of the _as() methods has changed several times as we developed the library.

Programs that require more finely-tuned security policies than are provided by the library's policy are encouraged to write their privileged methods using the extension framework. Applications that use the extension framework have the ability to perform arbitrary permission checks before performing any privileged operations.

6. Feasibility

For a best-case example of conversion, we converted the thttpd server to use the Privman libraries. thttpd is a simple, small, portable, fast and secure HTTP server. The version we used, 2.20c, consists of approximately 2800 lines of C code. The conversion process changed 26 of them, or less than 1% of the code base. The patch is available for download from the Privman web site, http://opensource.nailabs.com/privman/.

thttpd is essentially a best-case scenario. The server does no complex credentials management, and the preexisting user id management mapped well into the Privman usage pattern. Only four library calls needed to be supported: `open()`, `fopen()`, `bind()`, and `daemon()`. The Privman version actually gets simpler in some respects, as the user id management can be handled by Privman and moved out of the daemon.

In contrast, converting a standard BSD ftp server is closer to a worst-case scenario. The standard FTP server does significant work before it changes the user id of the server. Consequently, the process has built up large state that must be managed. To use `priv_rerunas()`, some reworking of the base code is required.

Wu-ftpd, a derivative of the standard BSD ftp server, totals approximately 32,000 lines of C code, ignoring the build system. The patch only changes 75 lines of C code, most of which are simple replacements of privileged calls with `priv_<foo>` calls. The remainder of the patch merely changes the build system to detect and build against the Privman library.

We believe these examples demonstrate that porting server processes to the Privman library is simple and non-invasive. While there are applications Privman does not handle well in its current state, most server applications should be easily handled.

Privilege separation has shown itself a valuable tool in hardening software against coding errors. OpenSSH has already had one vulnerability stopped by turning on privilege separation, and constructing a demonstration of the technique is as easy as it is useless. Only active deployment by commonly used servers will show the long-term value of the technique.

7. Performance

Privilege separation techniques have performance implications, given the two-process model and the IPC involved. To evaluate these costs, we conducted both micro- and macro-benchmarks. The micro-benchmarks measure the performance costs of specific operations, the macro-benchmark measures the effect on a larger system.

For a macro benchmark, we used the dkftpbench-0.45 ftp benchmark. This benchmark measures the maximum number of concurrent low-bandwidth users an ftp server can support while maintaining a minimum level of responsiveness. We tested the system with both stock wu-ftpd and the Privman-enabled wu-ftpd. The two versions of the ftp server were compiled from the exact same code, differing only in options presented to the configure script. The system was not optimized for FTP, as the goal was to isolate changes from the introduction of Privman, not maximizing FTP server speed.

The server system was a P3-866 Dell Optiplex GX110. The client loads were produced by a P3-1G Dell Latitude, connected to the server by an otherwise empty

100Mb Ethernet link. The benchmark repeatedly downloaded a ten-kilobyte file. The non-Privman-enabled server supported an average of 229 ± 20 clients over 5 runs, while the Privman-enabled server supported an average of 217 ± 4 users. The difference in performance is approximately 5%. Prior work retrofitting mandatory access controls to common Unix systems have shown similar minimal performance penalties[FRA99].

For a micro-benchmark, we compared the priv_* operation to the libc equivalent. The benchmark is distributed with Privman as the "microb" test. Slower operations like "`rerunas`" or "`pam_authenticate`" run 10,000 times per execution of the benchmark, and faster operations like "`open`" run 100,000 times per execution of the benchmark.

We ran the benchmark on the server system from the macro benchmark. We ran the benchmark five times, as a memory leak in `pam_authenticate` prevented us from increasing the iterations-per-run. We collected $\sum(x)$ and $\sum(x^2)$ from the runs, and determined the collective average and standard deviations.

When an application invokes any privileged operation, there is a constant overhead of approximately 30 microseconds from the two context switches. Since this overhead is a constant, the simplest operations show the largest performance penalty.

The time to marshal and un-marshal the arguments is also significant. The arguments to `priv_open` and `priv_fopen` are passed as strings, and so take longer to marshal than the arguments to `priv_bind`. In our benchmarks, `priv_open` showed the largest penalty, running 19.6 times slower on average than an equivalent `open(2)`.

The PAM operations took almost exactly the same amount of time whether they were proxied or ran directly. Loading and linking the shared libraries that make up the PAM framework is an expensive operation. This cost may have hidden the costs of marshaling the arguments.

Our experience leads us to believe that the performance penalties for additional access control are generally acceptable in comparison to the security benefits. As a general statement, the operations that require the most careful analysis are also the operations that tend to be slow initially. The performance penalty to an application is therefore minimized, although any particular operation may be a tenth as fast.

Operation	Average	Speed Penalty	Std Dev
bind	8 ms	3.77	13.16%
priv_bind	31 ms		8.01%
open	3 ms	19.6	20.60%
priv_open	64 ms		6.54%
fopen	5 ms	14.1	14.74%
priv_fopen	78 ms		3.46%
pam_auth	4,270 ms	1.06	0.95%
priv_pam	4,540 ms		0.77%
fork+exit	9,780 ms	2.15	1.08%
rerun	21,000 ms		0.56%

8. Conclusion

We have presented an overview of the Privman library. The Privman library enables applications to easily take advantage of process partitioning, which will help them become robust against certain types of programming errors.

Applications that use the Privman library can express the required security policy at a fine level of granularity. Consequently, Privman managed applications can approximate least-privilege in a way not common on UNIX systems.

Certain types of applications, primarily simple UNIX servers, are trivial to convert to using the Privman library. Other classes of applications, mainly those that need to change their security policy during normal execution, are less easily handled, but code changes are minimal.

For most cases, applications that use the Privman library will continue to perform adequately. The improvements to security justify any measurable performance degradation, although applications should be careful not to invoke Privman methods in the middle of tight time-sensitive loops.

The current release of the Privman libraries can be found at http://opensource.nailabs.com.

9. Bibliography

[BER95] Berman, Andrew, Virgil Bourassa, Erik Selberg, "TRON: Process-Specific File Protection for the UNIX Operating System", USENIX 1995.

[ACH00] Acharya, Anurag, Mandar Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Applications", USENIX Security, August 2000

[BLU01] Blum, Richard. *Postfix*. Sams Publishing, 2001.

[BRO75] Brooks, Frederick. *The Mythical Man Month*. Addison-Wesley, 1975.

[COW98a] Cowan, Crispin, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "Stack Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". USENIX Security, January 1998.

[COW98b] Cowan, Crispin, Calton Pu, and Heather Hinton. "Death, Taxes, and Imperfect Software: Surviving the Inevitable". New Security Paradigms Workshop, September 1998.

[COW00] Cowan, Crispin, Heather Hinton, Calton Pu, and Jonathan Walpole. "The Cracker Patch Choice, An Analysis of Post Hoc Security Techniques". NISSC, October 2000.

[FRA99] Fraser, Timothy, Lee Badger, Mark Feldman, "Hardening COTS Software with Generic Software Wrappers". IEEE Symposium on Security and Privacy, May 1999.

[GOL98] Goldberg, Ian, David Wagner, Randi Thomas, Eric A. Brewer, "A Secure Environment for Untrusted Helper Applications". USENIX Security, 1998.

[IEE97] Portable Operating System Interface (POSIX) – part1: System Application Program Interface (API): Protection, Audit and Control Interfaces. C Language, PSSG/DB Posix.1e October 1997.

[IOA02] Ioannidis, Sotiris, Steven M. Bellovin, Jonathan M. Smith, "Sub-Operating Systems: A New Approach to Application Security.". SIGOPS European Workshop, 2002.

[KIE02a] Kienzle, Darrell, and Matthew Elder. "Security Patterns for Web Development". http://patterns.nailabs.com, June 2002.

[KIE02b] Kienzle, Darrell, Matthew Elder, David Tyree, Jim Edwards-Hewitt. "Partitioned Application". Security Patterns Repository Version 1.0, http://patterns.nailabs.com, June 2002.

[NEL02] Nelson, Russell. "The qmail home page". http://www.qmail.org/top.html, June 2002.

[POS00] Single Unix V3, POSIX 1.1g, from IEEE Std. 1003.1g-2000 draft standard, and RFC2292. http://www.unix.org/single_unix_specification

[PRO02] Provos, Niels. "Privilege Separated OpenSSH". http://www.citi.umich.edu/u/provos/ssh/privsep.html, 2002.

[PRO03] Provos, Niels. "Systrace – Interactive Policy Generation for System Calls". http://www.citi.umich.edu/u/provos/systrace/, 2003.

[SAN02] System Administration, Networking and Security (SANS) Institute. "The Twenty Most Critical Internet Security Vulnerabilities (Updates)". as of May 2, 2002, http://www.sans.org/top20.htm, May 2002..

[UML03] var. "The User-mode Linux Kernel Home Page" as of Feb 28, 2003, http://user-mode-linux.sourceforge.net/

[VIE02] Viega, John, and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.

---

[i] As an object of type SCM_RIGHTS passed as part of a msg_control structure via the `sendmsg()` API.

[ii] This count includes more code from OpenSSH for very application-specific policy, but excludes the actual server implementations of the OpenSSH specific methods.

[iii] The client program may still have access to privileged resources, *e.g.* if the application opened a protected file, but has not yet closed it.

[iv] Here "system" refers to an actual physical machine.

[v] That is, verify that a given process was actually spawned by a specific binary whose identity can be verified.

[vi] This is, of course, configurable by the program's configuration file.

Appendix 1: libprivman API

Priv_init initializes the Privman server. Call this method first. When this method returns, the application will no longer have root privilege.

```
void    priv_init(const char *appname)
```

These calls act just like the methods they are named after, except that they use the Privman Server. Please see the system man pages for more details about the format of the arguments. The `priv_fork()` method causes both processes to fork, so the child process will still have access to a Privman server.

```
int     priv_open(const char *pathname, int flags, ...);
FILE*   priv_fopen(const char *pathname, const char *mode);
int     priv_unlink(const char *pathname);
int     priv_bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
pid_t   priv_fork(void);
```

These methods use the Privman Server. If the application uses PAM at all, you need to use the `priv_pam_` methods for all PAM methods. Please see the PAM man pages for more details about the format of the arguments.

```
int     priv_pam_start(const char *service, const char *user,
                  const struct pam_conv *conv,
                  pam_handle_t **pamh_p);
int     priv_pam_authenticate(pam_handle_t *pamh, int flags);
int     priv_pam_acct_mgmt(pam_handle_t *pamh, int flags);
int     priv_pam_end(pam_handle_t *pamh, int flags);
int     priv_pam_setcred(pam_handle_t *pamh, int flags);
int     priv_pam_open_session(pam_handle_t *pamh, int flags);
int     priv_pam_close_session(pam_handle_t *pamh, int flags);
int     priv_pam_chauthtok(pam_handle_t *pamh, int flags);
int     priv_pam_set_item(pam_handle_t *pamh, int item_type, const void *item);
int     priv_pam_get_item(pam_handle_t *pamh, int item_type,
                 const void **item);
int     priv_pam_putenv(pam_handle_t *pamh, const char *name_value);
int     priv_pam_getenv(pam_handle_t *pamh, const char *name);
int     priv_pam_fail_delay(pam_handle_t *pamh, unsigned int usec);
```

These methods allow the application to control the process behavior of the Privman server. `priv_exit()` causes the Privman server to exit (and thus prevents you from performing future privileged operations), `priv_daemon()` causes the Privman server to detach from the controlling terminal.

```
void    priv_exit(int status); /* Causes the Privman monitor to exit */
pid_t   priv_wait4(pid_t pid, int *status, int options, struct rusage *rusage);
int     priv_daemon(int nochdir, int noclose);
```

These methods allow the application to change its execution. `priv_execve()` executes the specified program as the user specified and chrooted into the path specified. `priv_popen_as()` works like popen, except that the program will run as the user specified. Use `priv_pclose()` to close the stream from `priv_popen_as()`.

```
int     priv_execve(const char *program, char * const argv[],
                    char * const envp[], const char * user,
                    const char* chroot);
FILE   *priv_popen_as(const char *command, const char *type, const char *user);
int     priv_pclose(FILE *stream);
```

This method allows a program to restart, running in a different context. The Privman server will start a new process, running as the user specified and chrooted into the path specified. The new process will execute the function `fnptr` with the supplied arguments, and then will return from `priv_init()`. Both processes will be able to talk to a Privman server.

```
int     priv_respawn_as(void (*fnptr)(char * const *), char * const arg[],
                    const char *user, const char *chroot);
```

This method is like `priv_respawn_as()`, except that the Privman server does not duplicate itself. Pass in `PRIV_RR_OLD_SLAVE_MONITORED` as a flag if the application wants the original process to be able to talk to the Privman server, or 0 if it wants the new process to be able to.

```
int     priv_rerunas(void (*fnptr)(char * const *), char * const arg[],
                    const char *user, const char *chroot, int flags);
```

These methods interface with the extension framework. The application must call the register methods before it calls `priv_init()`, and must call the invoke methods after. The register methods return a handle, which is then passed into the invoke methods to specify which registered method is being invoked.

```
int     priv_register_info_fn(char *(*fnptr)(char * const *));
int     priv_register_cap_fn (int   (*fnptr)(char * const *));
char   *priv_invoke_info_fn(int handle, char * const args[]);
int     priv_invoke_cap_fn (int handle, char * const args[]);
```

Appendix 2: Configuration file grammar

```
config                    ⇒     config_stmt_list

config_stmt               ⇒     bind_stmt | open_ro_stmt | open_rw_stmt
                          |     open_ao_stmt | unlink_stmt
                          |     auth_stmt | fork_stmt | rerunas_stmt
                          |     auth_allow_rerunas_stmt | runas_stmt
                          |     unpriv_user_stmt | chroot_jail_stmt

config_stmt_list          ⇒     config_stmt_list config_stmt
                          |     config_stmt

% The list of ports the client is allowed to bind to
bind_stmt                 ⇒     bind { portlist }

% The lists of files the client is allowed to manipulate
open_ro_stmt              ⇒     open_ro { pathlist }
open_rw_stmt              ⇒     open_rw { pathlist }
open_ao_stmt              ⇒     open_ao { pathlist }
unlink_stmt               ⇒     unlink { pathlist }

% Is the client allowed to use the authentication functions?
auth_stmt                 ⇒     auth bool

% Is the client allowed to cause the Privman server to fork?
fork_stmt                 ⇒     fork bool

% Is the client allowed to re-execute as a different user?
rerunas_stmt              ⇒     allow_rerun boolean

% If true, any successfully authenticated user is a valid rerunas user.
auth_allow_rerunas_stmt   ⇒     auth_allow_rerun boolean

% The client can re execute as any of these users
runas_stmt                ⇒     runas { userlist }

% The default unprivileged user, and starting chroot path.
unpriv_user_stmt          ⇒     unpriv_user userid
chroot_jail_stmt          ⇒     chroot path

pathlist                  ⇒     path pathlist
                          |     ∊

portlist                  ⇒     port portlist
                          |     ∊

userlist                  ⇒     user userlist
                          |     ∊

user                      ⇒     userid
                          |     '*'
```