USENIX Association

Proceedings of the 13th USENIX Security Symposium

San Diego, CA, USA
August 9–13, 2004

Privtrans: Automatically Partitioning Programs for Privilege Separation

David Brumley and Dawn Song
Carnegie Mellon University
{david.brumley,dawn.song}@cs.cmu.edu *

Abstract

Privilege separation partitions a single program into two parts: a privileged program called the monitor and an unprivileged program called the slave. All trust and privileges are relegated to the monitor, which results in a smaller and more easily secured trust base. Previously the privilege separation procedure, i.e., partitioning one program into the monitor and slave, was done by hand [18, 28]. We design techniques and develop a tool called Privtrans that allows us to automatically integrate privilege separation into source code, provided a few programmer annotations. For instance, our approach can automatically integrate the privilege separation previously done by hand in OpenSSH, while enjoying similar security benefits. Additionally, we propose optimization techniques that augment static analysis with dynamic information. Our optimization techniques reduce the number of expensive calls made by the slave to the monitor. We show Privtrans is effective by integrating privilege separation into several open-source applications.

1 Introduction

Software security provides the first line of defense against malicious attacks. Unfortunately, most software is written in unsafe languages such as C. Unsafe operations may lead to buffer overflows, format string vulnerabilities, off-by-one errors, and other common vulnerabilities. Exploiting a vulnerability can subvert a programs' logic, resulting in unintended execution paths such as inappropriately running a shell. Privileged programs — programs that run with elevated privileges — are the most common attack targets. A successful exploit may allow the attacker to execute arbitrary instructions with the elevated privileges. Even if attackers cannot execute arbitrary instructions, they may be able to change the semantics of the code by disabling a policy of the program. For example, an exploit may disable or alter an "if" statement that checks for successful authentication.

The number of programs that execute with privileges on a system is typically high, including setuid/setgid programs (e.g., ping), common network daemons (e.g., webservers), and system maintenance programs (e.g., cron). In order to prevent a compromise, every privileged program on a system must be secured.

Privilege separation is one promising approach to improving the safety of programs. Privilege separation partitions a single program into two programs: a privileged *monitor* program that handles all privileged operations, and an unprivileged *slave* program that is responsible for everything else. The monitor and slave run as separate processes, but communicate and cooperate to perform the same function as the original program. When necessary, a program can be separated into more than 2 pieces.

In this paper we show how to automatically add privilege separation to a program. The overall procedure for adding privilege separation to a program is depicted in Figure 1. The programmer supplies the source code and a small number of annotations to indicate privileged operations. Our tool, Privtrans, then automatically performs inter-procedural static analysis and C-to-C translation to partition the input source code into two programs: the monitor and slave.

Safety between the slave and monitor is primarily provided by process isolation in the operating system. Thus, a compromise of the slave does not compromise the monitor. The slave and monitor communicate via either interprocess or inter-network sockets.

*This research was supported in part by NSF and the Center for Computer and Communications Security at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views and conclusions contained her are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, NSF, Carnegie Mellon University, or the U.S. Government or any of its agencies.

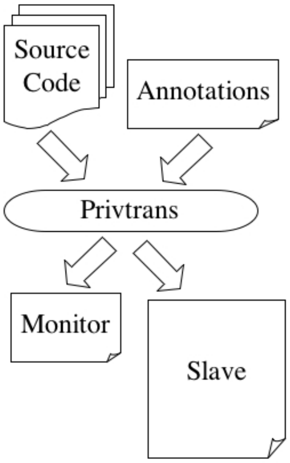

Figure 1: We automatically incorporate privilege separation into source code by partitioning it into two programs: the monitor which handles privileged operations and the slave which executes everything else. The programmer supplies a few annotations to help Privtrans decide how to properly partition the input source code.

The monitor exports only a limited interface to the slave. As a result, a compromised slave can execute only a limited number of privileged operations. Without privilege separation, a compromised slave may be able to run arbitrary instructions with the elevated privileges.

The monitor can further limit allowed privileged operations by employing policies. Since the slave asks the monitor to perform privileged operations on its behalf, the monitor can be viewed as interposing between privileged operations and the main execution in the slave. Policies can be fine-grained and express what privileged operations (or sequence of operations) are allowed, and are enforced during interposition.

1.1 Related approaches

In this section we discuss closely related approaches for the purpose of comparison. A thorough treatment of related work can be found in section 6.

System call interposition [1, 5, 14, 16, 27] monitors system calls and decides whether to allow or deny a call based upon a user-specified policy. Privilege separation is different from system call interposition because it statically changes the source code of a program. As a result, privilege separation can interpose on any function call, not just system calls.

Static analysis can be used to find bugs in programs [9, 12, 11, 19, 32, 37, 42]. However, it is difficult to perform precise static analysis on C programs. Our approach is to use static analysis as a tool to help partition the input source code, not find bugs. We rely upon process isolation for safety. We also use dynamic information to augment static analysis to reduce the number of expensive calls made by the slave to the monitor.

Provos et al. demonstrated the value of privilege separation in OpenSSH [28]. However, they manually edited OpenSSH to incorporate privilege separation. When privilege separation is enabled, OpenSSH resists several attacks [8, 23, 24]. Our techniques enable automatic privilege separation for programs, including OpenSSH.

Privman [18], a library for partitioning applications, provides an API a programmer can use when adding privilege separation to a program. However, the library can only make authorization decisions and does not provide complete mediation. Further, the programmer must manually edit the source at every call point to use the corresponding Privman equivalent. Our method uses data flow techniques to automatically find the proper place to insert calls to the monitor, and allows for finer-grained policies than access control. Policies are discussed in section 2.3.

1.2 Our contributions

In this paper, we describe our techniques that allow our tool Privtrans to automatically add privilege separation to programs. The programmer provides a few simple annotations to variables or functions that could be privileged. Privtrans then statically propagates the attributes by performing inter-procedural analysis on the source code to find privileged call sites. Privtrans then performs C-to-C translation to partition the input source code into the source code for the monitor and slave. Privtrans also automatically inserts dynamic checks which reduce overhead by limiting the number of expensive calls from the slave to the monitor.

Our contributions include:

- We design new techniques that allow us to develop the first tool for automatic privilege separation. Our automatic approach makes it easy to add privilege separation to many programs. We use a strong model for privilege separation (section 2).

Our approach allows for fine-grained policies (section 2). With only a few annotations provided by the programmer, our tool automatically performs interprocedural static analysis and C-to-C translation to partition a program into the privilege-separated monitor and slave programs (section 3).

Our results show that our approach is able to limit the interface exported by the monitor to the slave automatically. Furthermore, our experiments (section 4) demonstrate that the interface exported between the monitor and slave using our automatic privilege separation is comparable to manually integrating privilege separation. These results indicate that our automatic privilege separation can enjoy similar security as manual privilege separation.

As an additional benefit, automatic program translation, as opposed to manually changing code, allows us to track and re-incorporate privilege separation as the source code evolves.

- We design and develop techniques to augment static analysis with dynamic information to improve efficiency. Since static analysis of C programs is conservative, we insert dynamic checks to reduce the number of expensive calls made by the slave to the monitor.
- We allow for privilege separation in a distributed setting. Previous work only considered the monitor and slave running on the same host [18, 28]. Running the monitor and slave on different hosts is important in many scenarios (section 2), such as privilege separation in OpenSSL (section 4).

1.3 Organization

Section 2 introduces the model we use for privilege separation, the components needed for automatic privilege separation, and the requirements for programmers using privilege separation. Section 3 details our techniques and implementation of Privtrans. Section 4 shows Privtrans works on several different open-source programs. We then discuss when our techniques are applicable in section 5. We discuss related work in section 6, followed by the conclusion.

2 The general approach to automatic privilege separation

In this section we begin by describing the model we use for privilege separation. We then discuss the components needed for automatic privilege separation. Last, we discuss components that need to be supplied by the programmer.

2.1 Our model for privilege separation

In our model the monitor must mediate access to all privileged resources, *including the data derived from such a resource*. Specifically, it is not sufficient for the monitor to only perform access control. The monitor, and hence privileged data, functions, and resources, must be in an address space that is inaccessible from the slave. Our model is the same used by Provos et al. [28], but is stronger than that of Privman [18], since it encompasses both access control and protecting data derived from privileged resources.

It is often insufficient to only perform access control on privileged resources – it is also important to protect the data derived from the privileged resource. For example, if a program requires access to a private key, we may wish to regulate how that key is used, e.g., the key should not be leaked to a third party. Access control only allows us to decide whether to allow or deny a program access to the private key. A subsequent exploit may reveal that key to a third party. With privilege separation, the monitor controls the private key at all times. As a result, the monitor can ensure the key is not leaked. In our model policies can be expressed for both access control and protecting data data derived from privileged resources.

We assume that the original program accesses privileged resources through a function call. This assumption is naturally met by most programs, as privileges are only needed for a system call (such as opening a file and subsequently reading it) or library call (such as reading in a private key and subsequently using it to decrypt).

2.2 Components needed for automatic privilege separation

In order to create the monitor and slave from the given source code, automatic privilege separation requires:

```
1.  int sndsize = 128*1024;
2.  int s = socket(AF_INET,
  SOCK_RAW, IPPPROTO_ICMP);
3.  setsockopt(s, SOL_SOCKET,
  SO_SNDBUF, & sndsize,
  &(sizeof(sndsize)));
```

Figure 2: The monitor must track the socket created on line 2, and relate it to a subsequent call such as `setsockopt` on line 3.

1. A mechanism for identifying privileged resources, i.e., functions that require privileges or data acquired from calling a function that requires privileges.
2. An RPC mechanism for communication between the monitor and slave. The RPC mechanism includes support for marshaling/demarshaling arbitrary data types that the slave and monitor may exchange.
3. A storage mechanism inside the monitor for storing the result of a privileged operation in case it is needed in a later call to the monitor.

The third mechanism, storage, is needed when multiple calls to the monitor may use the same privileged data. Consider the sequence of calls given in Figure 2. On line 2 a socket is created. Since the socket is a raw socket, its creation is a privileged operation and must be executed in the monitor. At this point the monitor creates the socket, saves the resulting file descriptor `sock`, and returns an opaque index to the file descriptor to the slave. On line 3 the slave calls `setsockopt` on the privileged socket. To accomplish this the slave asks the monitor to perform the call and provides it with the opaque index from line 2. The monitor uses the index to get the file descriptor for `sock`, performs `setsockopt`, and returns the result. Note that if the monitor passed the file descriptor to the slave, we could not enforce any policies on how the file descriptor may be used.

Our tool, Privtrans, provides all three mechanisms. The programmer supplies a few annotations to mark privileged resources. Privtrans then automatically propagates attributes to locate all privileged functions and data. We supply a base RPC library, drop-in replacement wrappers for common privileged calls, and the monitor itself including the state store. Thus, the programmer is responsible only for adding a few annotations and defining appropriate policies.

2.3 Annotations and policies supplied by the user

Annotations Privtrans defines two C type qualifier [2] annotations for variables and functions: the “priv” and “unpriv” annotations [1]. The programmer uses the “priv” annotation to mark when a variable is initialized by accessing a privileged resource, or when a function is privileged and should be executed in the monitor. The “unpriv” attribute is only used when downgrading a privileged variable.

After the programmer supplies the initial annotations, propagation infers the dependencies between privileged operations and adds the privileged attribute as necessary. Propagation is discussed further in section 3.

The “priv” and “unpriv” attributes are used to partition the source code into the monitor source and the slave source. If a variable has the privileged attribute, it should only be accessed in the monitor. Similarly, if a function has the privileged attribute, it should only be executed in the monitor. All other statements and operations are executed in the slave.

The programmer decides where to place annotations based upon two criteria: what resources are privileged in the OS, and what is the overall security goal. A resource is privileged if it requires privileges to access, e.g., opening a protected file.

Annotations can also be placed so that the resulting slave and monitor meet a site-specific security goal. For example, a site-specific goal may state all private key operations happen on a secured server. With properly placed annotations the source will be partitioned such that only the monitor has access to the private keys. The monitor can then be run on the secured server, while the slave (say using the corresponding public keys) can be run on any server.

Policies A monitor policy specifies what operations the slave can ask the monitor to perform. The monitor policy is written into the monitor itself as C code. Therefore, our model does not limit the complexity or detail of policies. Our approach guarantees the enforcement of policies on privileged resources or data since all privileged calls must go through the monitor.

Many policies are application specific, and thus need to

[1] Note our annotations are similar to, but not the same as, subtypes.

be supplied by the programmer. However, there are several policies that can be automatically generated. For example, many compilers create control flow graphs during optimization. The control flow graph (CFG) can be used to build a finite state machine (FSM) model of possible privileged calls.

The FSM of privileged calls is produced by the CFG by first removing edges in the CFG that do not lead to a privileged call. The resulting FSM is collapsed by removing unprivileged calls. The result is a directed graph of valid privileged call sequences. The modified FSM is saved to a file, and read in by the monitor at run time during initialization. Requests from the slave are checked against the FSM by the monitor: a call is allowed only if there is an edge from the proceeding call to the current call in the FSM. As a base case, the monitor initialization routine (`privwrap_init`) is always allowed.

One potential problem with FSM's is the call policy may still be too coarse-grained. For example, if a privileged call `f` is made during a loop, the policy will allow an infinite sequence of calls to `f`. One approach to further limit FSM's is to create a PDA based upon the source. The PDA may further limit the number of allowable call sequences.

Others have shown how to automatically create FSM's, PDA's, and similar structures which can be used to limit the call sequences which can be used by the monitor to limit call sequences [9, 15, 22, 30, 35, 40]. We do not duplicate previous work here, as our framework supports ready integration of fine-grained policies. The policies are written into the monitor after partitioning. Policies can be as expressive as needed, since they can be written directly into the monitor source code.

Note that because our approach enables the monitor to export a limited interface, policies need only be written for privileged operations. This fact may make it easier to write a more precise policy than the system call interposition approach. In system call interposition, a model is needed for both privileged and unprivileged system calls. The policy in system call interposition is usually more complex as the number of system calls increases. Privilege separation limits the number of privileged operations to only the interface exported by the monitor, which may reduce the complexity of the resulting policy.

Downgrading data Since the monitor mediates all access to privileged data, it is sometimes useful to *downgrade* data (i.e., make previously privileged data unprivileged). The purpose of a downgrade is to allow otherwise privileged data to flow from the monitor to the slave.

```
1.  int __attribute__((priv)) a;
2.  __attribute__((priv)) void
 myfunction();
3.  int b = f(a);
```

Figure 3: Line 1 marks a variable `a` as privileged. The annotation is added because the programmer expects `a` to be initialized by a privileged function call, `f` in this example. `a` is transmitted to the monitor which executes `f` on behalf of the slave. Further, `b` will also be marked privileged, and any subsequent use of `b` will be executed in the monitor. On line 2, we mark the `myfunction` function privileged. Any call to this function will be executed in the monitor.

Consider a program that reads a file containing a public/private key pair. Accessing the file is privileged, since it contains the private key. However, the public key is not privileged. With privilege separation, the monitor has access to the file, while the slave does not. Programmers are free to define *cleansing* functions that downgrade data. In the scenario above, the programmer writes an extension to the monitor that returned only the public key to the slave, while maintaining the private key in the monitor. Cleansing functions are application specific, and should be provided by the user.

3 The design and implementation of Privtrans

We first discuss at a high level the process of running Privtrans on existing source code to produce the monitor and slave source code. We then discuss how Privtrans implements each step in the process, and how we reduce the number of calls from the slave to the monitor. We also show how the programmer can easily extend Privtrans for new programs using our base RPC library. We conclude this section by describing the monitor state store.

3.1 High-level overview

We begin by describing the process of adding privilege separation at a high level. Privtrans takes as input source code that we wish to have rewritten as two separate programs: the monitor source code and the slave source code.

Annotations First, the programmer adds a few annotations to the source code indicating privileged operations. Annotations are in the form of C attributes. An annotation may appear on a function definition or declaration, or on a variable declaration, such as in Figure 3.

Attribute propagation Privtrans propagates the programmer's initial annotations automatically. After propagation, a call site may either have a privileged argument, or the result may be assigned to a privileged variable. Additionally, a function callee itself may be marked privileged. We wish to have the slave ask the monitor to execute any such call on its behalf.

Call to the monitor Privtrans automatically changes a call site that is identified privileged to call a corresponding wrapper function, called a *privwrap* function. A privwrap function asks the monitor to call the correct function on the slave's behalf by: 1) marshaling the arguments at the call site, 2) sending those arguments to the monitor, along with a vector describing the run-time privileged status of each variable, 3) waiting for the monitor to respond, 4) demarshaling any results, and 5) arrange for the proper results to be returned to the slave.

Execution and return in monitor Upon receiving a message from the slave, the monitor calls the corresponding *privunwrap* function. The privunwrap function 1) demarshals the arguments sent to the monitor, 2) checks the policy to see if the call is allowed, 3) looks up any privileged actuals described as privileged in its state store, 4) performs the function requested, 5) if the results are marked privileged, hashes the results to its state store and sets the return value of the function to be the hash index, and 6) marshals the return values and sends them back to the slave.

Starting the monitor Privtrans inserts a call to `priv_init` as the first executable line in `main`. `priv_init` can optionally fork off the monitor process and drop the privileges of the slave, or else it contacts an already running monitor. The slave then waits for notification from the monitor that any initialization is successful. Initialization of the monitor consists of initializing the state store, along with any policy-dependent initialization. After `priv_init` returns, the slave can begin main execution.

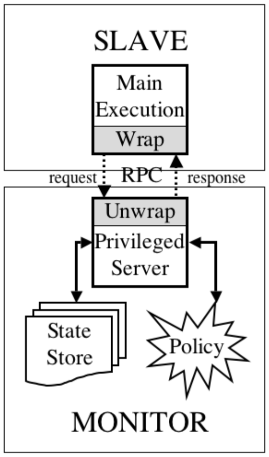

Figure 4: The output of translation partitions the input source code to create two programs: the monitor and the slave. RPC between the monitor and slave is accomplished via the privwrap/privunwrap functions. The monitor may consult a policy engine when asked to perform a privileged function. Finally, the monitor may save results from a function call request in case later referenced by the slave.

This process is depicted in Figure 4. We detail each stage in the following subsections.

3.2 Locating privileged data

Privtrans uses CIL [22] to read in and transform the source code. Privtrans performs inter-procedural static analysis to locate all potentially privileged call sites. To reduce overhead, Privtrans also inserts run-time checks to limit the number of calls from the slave to the monitor.

3.2.1 Static analysis and rewriting privileged calls

The programmer annotates a few variables or functions using C attributes. Privtrans uses two attributes, *priv* and *unpriv*, used to respectively mark privileged and unprivileged variables or functions. The programmer need only use the unpriv attribute when casting a privileged variable to unprivileged. Privtrans performs propagation of the initial annotations by adding the priv to any variable that may become privileged.

Annotations are required since a program may rely upon configuration files, environment variables, etc., which determine whether a call will be privileged. For example, web-server's typically read a configuration file which determine whether to bind to a privileged port (e.g., port 80) or not. Understanding application-specific configurations is beyond the scope of static analysis.

Recall that the original program accesses privileged resources through a function call. The slave should ask the monitor to execute any call where the arguments, return value, or callee function is marked privileged. Privtrans rewrites a call to `f` that may be privileged to the corresponding wrapper function `privwrap_f`. Wrapper functions such as `privwrap_f` use the underlying RPC mechanism to ask the monitor to call a function (`f` in this case), wait for the reply, and arrange for the proper return values.

Privtrans static analysis is standard meet-over-all-paths data-flow analysis: the priv attribute is added to a variable if it can be assigned to by another privileged variable over any path in the program. Privtrans performs inter-procedural analysis by iteratively adding the privileged attribute across defined functions. Since we do not have the function body for procedures declared but not defined, we assume that the privileged attribute could be added to any pointer argument, i.e., a pointer value could be a return value.

The *priv* attribute can be added incrementally to the source code. Without any privileged annotations, the entire input program will be rewritten as the slave. After adding a *priv* attribute, the resulting slave and monitor can be run to see if they work. If an attribute is missing the slave will attempt a call without appropriate privileges, and the call will fail. Regression test suites can be used to insure that the slave and monitor cooperate at all necessary privileged call sites.

The result of the propagation phase is a set of calls that potentially should be executed by the monitor. Our analysis is conservative [2], so any call site that may be privileged is considered privileged. In 3.2.3 we explain how we augment our static analysis with run-time information to reduce unnecessary calls to the monitor.

3.2.2 Polymorphic functions

During static analysis, we may determine a function callee is polymorphic, i.e., some calls to the function are privileged and some are not. Privtrans uses variable annotations with the priv attribute to support polymorphism. If the priv attribute appears on a variable used as an argument to a function, or assigned to the result of a function, then the call is considered privileged and the caller should ask the monitor to perform the called function.

Consider Figure 5(a). On line 3 there are two calls to function `f2`. The first call passes `a`, a privileged variable, while the second call passes `b`, an unprivileged variable. The attribute distinguishes between the privileged and unprivileged call. In this example, the first call would be rewritten as `privwrap_f2`, while the second call would remain unchanged.

3.2.3 Improving static analysis with dynamic information

Since static analysis is conservative, not all potential calls to the monitor are really privileged during run-time. An example of such a call is given in Figure 5(a). After static analysis, we determine that `f` may be a privileged call, thus we should invoke `privwrap_f` which calls the monitor to call `f`.

However, every time the slave asks the monitor to perform a call, the slave suffers the overhead of 1) marshaling all arguments on the slave and demarshaling them in the monitor, 2) calling the monitor, which can result in a context switch if the monitor and slave are on the same host, and 3) marshaling the results in the monitor and demarshaling them on the slave. Thus, we want to make the slave only ask the monitor to perform a call if absolutely necessary.

Normally expensive context or path sensitive analysis is used to improve the accuracy of simple dataflow analysis. A key insight is that during the process of translating the input code into the monitor and slave, we can insert dynamic checks to limit the number of calls from the slave to the monitor. The dynamic checks allow for the same or better accuracy in determining privileged call sites than full context and path sensitive analysis.

In order to limit the number of calls to the monitor, we add an extra vector to the slave for every privileged callee

[2] We do not handle function pointers. The programmer can add the priv attribute to the pointed-to function if necessary.

```
1.  int __attribute__((priv)) a;
2.  int b = 0;
3.  f2(a); f2(b);
4.  if ( some expression ) b = a;
5.  b = f(arg1, arg2);
```

(a) The call to `f` should be executed in the monitor when the `if` statement on line 4 is true, else the call can be executed by the slave directly. We cannot know statically which case will happen. Also, on line 3 we encounter the polymorphic function `f2`. The first call to f2 is privileged, the second is not.

```
1.  int __attribute__((priv)) a;
2.  int b = 0;
3.  int privvec_f[3] =
  {E_UNPRIV,E_UNPRIV,E_UNPRIV};
4.  int privvec_f2[1] = {E_PRIV};
5.  privwrap_f2(a, privvec_f2);
  f2(b);
6.  if ( some expression )
  { privvec_f[0] = E_PRIV; b = a; }
7.  b = privwrap_f(arg1, arg2,
  privvec_f);
```

(b) We add a vector describing the run-time privilege status of the return value and each argument to `privwrap_f`. Initially, the vector indicates that none of the arguments are privileged. If the `if` statement on line 4 is true, we mark `b` as privileged and thus `f` will be executed in the monitor.

Figure 5: Privtrans rewrites the code on the left to make use of the monitor, as shown on the right.

(as determined by static analysis). The vector contains the current run-time privilege status of variables used at a possibly privileged call site we found with static analysis. Each position in the vector contains one of two values: E_PRIV for privileged or E_UNPRIV for unprivileged.

An example is given in Figure 5(b). The vector "privvec_f" describes the run-time privilege status of the return value and arguments of the call to "f", read left to right. When the vector contains only E_UNPRIV, the wrapper "privwrap_f" can decide to make the call locally instead of calling the monitor.

It is safe to use the dynamic information even if the slave is compromised. Consider the two cases of a compromise: a privileged call is made unprivileged or an otherwise unprivileged call is considered privileged. The former case is always safe, since it does not give an attacker any privileges.

In the latter case, the monitor receives a spurious call that the slave should be able to make itself. Such spurious calls are also safe. First, since the slave could have made the call by itself, the slave is gaining no additional information or privileges by asking the monitor to perform the call on its behalf. Second, if the call conflicts with the monitor's policy it could refuse the call (and possibly exit if a brute force attack is suspected). The second approach, refusing the call, is the recommended solution.

3.3 RPC and the wrapper functions

Privtrans supplies a library of common privwrap/privunwrap functions such as opening a file or creating a socket. The wrappers are reused for each program on which we perform privilege separation. The wrappers are implementations of functions created using the "rpcgen" protocol compiler.

We provide wrappers for common privileged calls instead of automatically generating them from the source because we may not know statically how to wrap a pointer argument to a call. Wrapping pointers requires knowing the pointer's size. Generally functions that take a pointer argument also take an argument indicating the pointer's size. Finding this out is easily done by a human, say by consulting the appropriate man page, but is difficult to do with static analysis alone. The wrapper functions are created only once, and then can be reused.

Using a shared memory region between the monitor and slave for passing pointers may seem like an attractive solution, but this approach violates the abstraction boundary between monitor and slave [3] The monitor must maintain a separate copy of any pointers it uses similar to the user/kernel space distinction.

Although we supply wrappers for many common functions, a programmer may occasionally need to define their own. Creating additional privwrap/privunwrap

[3] A shared memory region that is read-only for the slave could be used to pass messages from the monitor to slave.

function is not difficult since Privtrans provide a base RPC library. The typical privwrap/privunwrap function is less than 20 lines of code. The wrapper functions are simple implementations of the declarations generated by rpcgen.

3.4 Execution in the monitor and the monitor state store

The slave uses a privwrap function to request the monitor to execute a function. Upon receiving a request, the monitor demarshals all arguments. If an argument is marked privileged in the monitor's corresponding privunwrap function, then the argument supplied by the slave is an index to a previously defined privileged value. This fact follows from the observation that the slave alone could not have derived a valid index to data on the monitor.

We use a hash table to lookup each privileged argument, and return the appropriate reference. If the index is not valid, the monitor aborts the operation. Assuming a valid index, the monitor executes the correct call. If the return values (recall pointer arguments are also considered return values) are cast (statically through user annotations) to unprivileged, then the monitor returns the values directly. If the return values are privileged, then the monitor stores the results and returns the index to the slave.

The state store itself is implemented as a collection of hash tables, one for each base C type [4]. The opaque index returned to the slave is an index into the hash table. The opaque indexes are secure since the client cannot generate a valid index on its own. While there are many methods to create opaque indexes, we simply associate a random number to each indexed value.

4 Experimental results

To demonstrate Privtrans, we use it to automatically integrate privilege separation into several open-source programs and one open-source library: thttpd [26], the Linux "ping" program, OpenSSL[34], OpenSSH [33], chfn and chsh [20]. Table 1 summarizes our results.

name	src lines	# user annotations	# calls automatically changed
chfn	745	1	12
chsh	640	1	13
ping	2299	1	31
thttpd	21925	4	13
OpenSSH	98590	2	42
OpenSSL	211675	2	7

Table 1: Results for each program with privilege separation. The second column is the number of annotations the programmer supplied. The third column is the number of call sites automatically changed by Privtrans

4.1 OpenSSH

Provos et al. has previously manually added privilege separation to OpenSSH version 3.1p1 [28]. The privilege separated code is available as OpenSSH version 3.2.2p1. Automatically adding privilege separation to OpenSSH 3.1p1 serves as a benchmark for our automatic approach.

Our results produce a slave and monitor that are similar to the manual OpenSSH separation by Provos et al.. The OpenSSH server runs as root and monitors for incoming connections. Upon receiving a connection, OpenSSH forks off a slave and monitor process. The slave asks the monitor to perform authentication and perform private key operations [5].

After authentication, the monitor changes the uid and gid of the slave to be that of the authenticated user. This is accomplished through a new system call that allows a monitor to change the uid of the corresponding slave. Provos et al. [28] have a complex, though portable solution where the slave exports any accumulated state to the monitor, which is exported back to the user's login shell. Our method is less portable but more simple.

To use Privtrans on OpenSSH, 2 annotations are needed: one for the private keys and one for the authentication mechanism. The interface exported by the monitor is thus limited to pam calls and the RSA private key operations. The result is a version comparable to Provos et al..

[4] Using multiple hash tables reduces the number of type casts done to eventually get the correct type.

[5] We did not add privilege separation for all authentication mechanisms, as with Provos et al.. Instead, we focused on PAM authentication for demonstration purposes.

4.2 chfn and chsh

chfn changes the “finger” information for a user. chsh changes the login shell for a user. Both are normally setuid in order to write to the password file and authenticate users, and both retain their privileges during program execution. chfn and chsh have historically had security vulnerabilities [31, 6]. Only 1 annotation needs to be specified for the PAM authentication handle. 12 and 13 call sites were automatically changed in chfn and chsh respectively.

4.3 thttpd

thttpd is a HTTP server written with performance in mind. thttpd requires privileges to `bind` and `accept` on port 80. Integrating privilege separation required the user to provide 4 annotations. It took approximately 2 hours from downloading the source to place the correct annotations. 13 call sites are automatically changed to use calls to the monitor: 1 `socket`, 1 `bind`, 3 `fcntl`, 1 `setsockopt`, 4 `close`, 1 `listen`, 1 `accept`, and 1 `poll`. Privtrans comes with wrappers for all functions.

Integrating privilege separation is valuable for thttpd. Although thttpd eventually drops privileges, privileges are retained for significant initialization. thttpd parses user input, sets up signal handlers, then creates and binds several sockets before dropping privileges. Thus, if an attacker can raise a signal before the program calls setuid, the signal handlers will be executed with elevated privileges. One such signal handler, SIG_ALRM, could cause the program to core dump in /tmp. With knowledge of the PID, an attacker may be able to overwrite any file in /tmp.

4.4 ping

The ping source is available as part of the iputils package in many Linux distributions. ping is normally setuid to root in order to create a raw socket. Although ping drops privileges after socket creation, an exploit could still break policies we may wish to enforce. For example, one may wish to allow ping to only send a certain number or limit the size of packets sent to a destination. Note access control is insufficient for such policies. Even after privileges are dropped such a policy may not be enforced if there is a buffer overflow or other type-safety violation.

Privilege separated ping is also useful for securing a site that wishes to limit internal ICMP messages. ICMP messages are commonly used for covert communication in hacker tools [10]. The normal solution is to use a firewall that disallows ping requests. However, this approach does not let legitimate internal ping clients ping outside hosts.

Using privilege separation we divide ping into the monitor and slave. The slave is ran on each internal host, while a single monitor can run from a trusted “ping host”. While there are other ways to accomplish the same objective, privilege separation gives a new alternative. Circumstances may make such alternatives attractive.

The privilege separated version of ping is created by the user adding 1 annotation to the original source. It took approximately 1.5 hours from downloading the ping source to place the proper annotation. 31 call site are automatically changed to use the monitor: 1 `socket`, 21 `setsockopt`, 1 `getsockopt`, 2 `ioctl`, 1 `sendmsg`, 2 `recvmsg`, 1 `poll`, 1 `getuid`, and 1 `bind`. Privtrans comes with all 14 wrapper functions.

4.5 OpenSSL

Integrating privilege separation into OpenSSL adds new avenues for securing a site. Many sites may wish to reuse certificates for multiple services since certificates are often expensive and are unique to a host, not a service. For example, a small business may want to use the same SSL certificate for both a web-server and an IMAP server. The main drawback for using one certificate for multiple services is that a compromise in any service will reveal the private keys for all services.

In this experiment we add privilege separation into OpenSSL, so that many SSL services (i.e. slave’s) will all use the same monitor to perform privileged RSA private key operations. Thus, trust is only given to one server, the monitor, while multiple services can use the certificate.

We added 2 annotations for the RSA operations that required the private key: one for `RSA_private_encrypt` and one for `RSA_private_decrypt`. It took approximately 20 minutes to find the correct place to add annotations. Privtrans then rewrites the library so these two functions will be executed in a monitor, while everything else will be in the slave. 7 call sites were automatically changed within the library. We then compiled and linked stunnel [17] against our OpenSSL library. stunnel encrypts arbitrary TCP connections inside an SSL session. We gave the monitor the private RSA key, and provided stunnel with only the RSA public key. As a result, all RSA decryptions needed during an SSL session were done by the monitor instead of in stunnel.

4.6 Performance overhead

We ran experiments on an Intel P4 2.4 GhZ processor with 1 GB of RAM running Linux 2.4.24. The base overhead for a cross domain call (i.e. between a client and server) vs. a local call is about 84% on our test machines. We could use techniques such as software-based isolation [38], which can reduce the cost of a cross domain call by up to three orders of magnitude.

We performed several micro-benchmarks. In each micro-benchmark we timed a system call done locally vs. the same call via the appropriate privwrap wrapper. Our results show a performance penalty factor of 8.83 for a `socket` call, 7.67 for an `open` call, 9.76 for a `bind` call, and 2.17 for a `listen` call. The average time difference between a local call and wrapper call is 19 μs vs. 88 μs. Our results compare favorably to Privman [18], the only other implementation with comparable wrapper functions. For instance, in Privman the cost of an open call done via their library is about 19.6 times slower, while our implementation only has a 7.67 performance penalty. Other measurements are similarly about the same or better than Privman.

We also performed macro-benchmarks for several application tested. thttpd was tested by measuring the average web-server response time over 1000 iterations to download index.html. For ping, we tested the time difference between the unmodified program and the privilege separation version when pinging localhost 15 times. For OpenSSL, we asked the OpenSSL library to decrypt 1000 randomly generated (but constant throughout the experiment) messages. The privileged separated OpenSSL library has an additional 15% overhead, ping has an additional 46%, and thttpd only suffered an additional 6% overhead.

The main cause of additional overhead in ping and OpenSSL is transferring data between the slave and monitor. With ping, for example, a 4K block of data is transferred twice each time a ping reply is received: once when calling privwrap_recvmsg from the slave to monitor, and once on the return from the monitor to slave. Such overhead is unfortunately unavoidable unless the ping source is rewritten. Alternatively, we could specialize the wrapper functions for ping to eliminate about half of the overhead.

The overhead from privilege separation is often not a limiting factor since cycles are cheap but secure software is not.

5 Discussion

In this section we discuss how our techniques work in practice.

Automatic vs. Manual. Our approach leverages sound dataflow analysis to rewrite generic applications. Although possible, it is unlikely the automatic approach will result in code with optimal performance. The reason is in the manual approach the programmer is free to use application-specific knowledge to fine-tune (or even rewrite!) the program for better performance.

However, our approach is much easier. Finding the annotation locations is very simple: the set of privileged operations on a system is generally well known and easy to spot. In addition, the result of missing an annotation is the slave attempting to perform a privileged operation without appropriate privileges. Thus, the programmer is free to incrementally add annotations until the slave and monitor perform correctly.

Our approach outputs human-readable C code, which allows the programmer to inspect and debug the monitor and slave easily. In our experience, it takes at most a few hours to find the proper places to add annotations.

Portability. Any platform that supports inter-process communication will likely be amenable to running privilege-separated programs. Since we rewrite C source code, the crux of our approach does not suffer portability problems. Also, since we separate out privileged resources and data derived from those resources, our approach typically does not require OS-dependent mechanisms such as file-descriptor passing. For example, our techniques and results should apply equally well to Microsoft Windows. In the future, we plan on applying our approach to other operating systems, including Microsoft Windows.

However, the implementation of the wrappers may have to be customized depending upon the interfaces supported by the OS. Last, our new system call that allows a process to change the privilege of it's children is not portable. Previous research has addressed this issue, as detailed below, and we do not duplicate their work.

Potential issues and solutions. There are several issues for privilege separation. Note many of these issues are not specific to our approach, and apply to any privilege separation approach.

The setuid and getuid-style routines may not behave as expected in the original program. For example, since the privilege-separated version drops all privileges immediately, a call to getuid will return the uid of the unprivileged user. This may break programs that expect to be setuid and checks for certain privileges through the getuid call. In our approach we changed getuid-style calls to return the uid of the monitor. setuid calls should similarly change the uid of the monitor, not the slave.

File descriptor numbering will also be different due to the socket between the slave and monitor. For example, with the `select` call the first argument is an integer indicating the highest number file descriptor to check for a change in status. If the slave asks the the monitor to perform a `select` call, the highest file descriptor argument supplied by the slave may not coincide with the correct file descriptor in the monitor. To solve this problem `select` calls should be rewritten as `poll`, since the `poll` call contains the list of actual file descriptors to check for a change in status.

Previous work has also reported issues around `fork` [18]. For example, consider a file descriptor opened by the monitor for a slave. Suppose the slave forks off a new child process, which asks the monitor to close the file descriptor. In the slave the parent process expects the file descriptor open, while the child expects to have the file descriptor closed. Thus, with privilege separation, we must distinguish in the monitor between the file descriptors owned by the child process in the slave from the parent process in the slave. Our solution is to fork off a new monitor when a new slave is forked.

Another important issue is resolving which elements of a collection data structure contain privileged and unprivileged data, such as an array that contains both types. The opaque identifier returned during privileged data creation can help identification, even though this may not work in all cases. For example, in thttpd poll() is called with file descriptors owned by the monitor and the slave, which must be distinguished. Our opaque identifiers start at 100, so we can distinguish between a file descriptor owned by the slave, which will be less than 100, and one owned by the monitor, which will be over 100.

We do not perform any pointer alias analysis. This leads to two potential problems. First, there may be a pointer in the slave to an opaque index, which is later used in an operation. We cannot know of such an operation without full pointer analysis. Second, since we don't know the liveliness of pointers we do not know when it is safe to free a variable in the monitor. Thus the monitor never frees memory for a privileged value. In our experience, neither has been a problem, i.e. the slave never tried to use a opaque identifier and the monitor's memory usage was modest.

Last, there is no simple way for a program that accumulates state as the unprivileged user to become another user. To solve this problem, we created a system call that allows a superuser process to change the uid of any running process, and a non-superuser process to change the uid of any of it's slaves. This system call makes sense: a superuser process could always run a program itself, granting the necessary privileges. The disadvantage of this approach is our system call is system-specific. Other portable but more complex techniques are explored by Kilpatrick [18] and Provos et al [28].

6 Related work

While privilege separation can drastically reduce the number of operations executed with privileges, it is even more important to write applications securely from creation. Programs should be developed with the principle of least privilege, which states every operation should be executed with the minimum number of privileges [29].

VSFTPD [13] and Postfix [36] use separate processes to limit the damage from a programming error. Both programs were created from the ground up following the principle of least privilege.

Provos et al. demonstrated the value of privilege separation in OpenSSH [28]. However, they manually edited OpenSSH to incorporate privilege separation. When privilege separation is enabled, OpenSSH resists several attacks [8, 23, 24]. Our technique entails automatic privilege separation.

Privman [18], a library for partitioning applications, provides an API a programmer can use to integrate privilege separation. However, the library can only make authorization decisions, and cannot be used for fine-grained policies. Further, the programmer must manually edit the source at every call point to use the corresponding Privman equivalent. Our method uses data flow techniques to find the proper place to insert calls to the monitor, and allows for fine-grained policies.

Several different mechanisms exist for dynamically checking system calls such as Systrace [27], GSWTK [14], Tron [5], Janus [16], and MAPbox [1]. Dynamically checking system calls does not allow for fine-grained policies on regular function calls, although this technique does not require program source code. Another drawback is that dynamic techniques cannot optimize the number of checks. Our approach works for arbitrary function calls, allows for fine-grained policies, and optimizes the number of expensive calls to the monitor.

Type qualifier propagation has been used to find bugs in C programs [32, 42] . We use attributes as type qualifiers, and use them to guide rewriting the code. Type qualifiers are used to identify potentially sensitive data in Scrash [7]. CIL is used in this work to rewrite the application so that sensitive data may be removed from a core file.

JFlow/JIF [21, 41, 43] and Balfanz [4] show how to partition applications by trust level in Java. Since Java is type-safe it is less vulnerable to malicious attacks. Instead, JFlow/JIF and Balfanz focus on preventing unintentionally leaking information in a program.

Operating system mechanisms [3, 25, 39] can provide ways to reduce the privileges of applications. However, these mechanisms do not have access to the internals of a program, and thus cannot be used for arbitrary function calls as with privilege separation.

Static analysis can be used to find bugs in programs [9, 12, 11, 19, 32, 37, 42]. Our goals are different: we wish to limit the damage from an unknown bug. However, we use static analysis as a tool to automatically find privileged operations.

7 Conclusion

We have shown how to automatically integrate privilege separation into source code. We consider a strong model of privilege separation where accessing privileged resources is relegated to the monitor. The monitor can enforce policies on data derived from a privileged resource in addition to access control. Our tool Privtrans uses static techniques to rewrite the C code, and inserts dynamic checks to reduce overhead. Privtrans requires only a few annotations from the programmer, typically fewer than 5.

We ran Privtrans on several open-source programs successfully. Privilege separation has unique benefits for each program. The overhead due to privilege separation was reasonable. Thus, Privtrans is applicable to a wide variety of applications.

8 Acknowledgements

We would like to thank Niels Provos for helpful discussions, comments, and thoughts regarding our work. We would also like to thank Lujo Bauer, Robert Johnson, James Newsome, David Wagner, Helen Wang, and the annonymous reviewers for their helpful comments while preparing this paper.

References

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. In *the Proceedings 9th USENIX Security Symposium*, 2000.

[2] American National Standards Institute (ANSI). *Rationale for International Standard – Programming Languages – C*, October 1999.

[3] Lee Badger, Daniel Sterne, and David Sherman. A domain and type enforcement unix prototype. In *the Proceedings of the 5th USENIX Security Symposium*, 1995.

[4] Dirk Balfanz. *Access Control for Ad-hoc Collaboration*. PhD thesis, Princeton University, 2001.

[5] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: Process-specific file protection for the

unix operating system. In *the Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, 1995.

[6] Marc Bevand. OpenBSD chpass/chfn/chsh file content leak. `http://www.opennet.ru/base/bsd/1044293885_871.txt.html`, 2003.

[7] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating security crash information. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.

[8] CERT/CC. CERT advisory CA-2003-24 buffer management vulnerability in OpenSSH, September 2003.

[9] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *the Proceedings of the ACM Conference on Computer and Communications Security 2002*, 2002.

[10] David Dittrich. The Tribe Flood Network distributed denial of service attack tool. `staff.washington.edu/dittrich/misc/tfn.analysis`, 1999.

[11] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *the Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2000.

[12] Dawson Engler, David Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *the Proceedings of the Symposium on Operating System Principles*, 2001.

[13] Chris Evans. Very secure ftp daemon. `http://vsftpd.beasts.org`.

[14] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *the Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[15] Jonathon Giffin, Somesh Jha, and Barton Miller. Detecting manipulated remote call streams. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.

[16] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *the Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, USA, 1996.

[17] Brian Hatch and the stunnel development team. stunnel 4.04. `http://www.stunnel.org`, 2004.

[18] Douglas Kilpatrick. Privman: A library for partitioning applications. In *the Proceedings of Freenix 2003*, 2003.

[19] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *the Proceedings of the 10th USENIX Security Symposium*, 2001.

[20] Andries Brouwer (Maintainer). util-linux version 2.11y. RedHat RPMS.

[21] Andrew Myers. Jflow: Practical mostly-static information flow control. In *the Proceedings of the Symposium on Principles of Programming Languages*, 1999.

[22] George Necula, Scott McPeak, S. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction*, 2002.

[23] OpenSSH. Buffer overflow in AFS/Kerberos token passing code. `http://www.openssh.org/security.html`, April 2002.

[24] OpenSSH. Openssh remote challenge vulnerability. `http://www.openssh.org/security.html`, June 2002.

[25] David Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.

[26] Jef Poskanzer. thttpd. `http://www.acme.com/software/thttpd/`.

[27] Niels Provos. Improving host security with system call policies. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.

[28] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.

[29] Jerome Saltzer. Protection and the control of information in multics. In *Communications of the ACM*, July 1976.

[30] Fred Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[31] securiteam.com. Linux ’util-linux’ chfn local root vulnerability. `http://www.securiteam.com/unixfocus/5EP0V007PK.html`, 2002.

[32] Umesh Shankar, Kunal Talwar, Jeffrey Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *the Proceedings of the 10th USENIX Security Symposium*, 2001.

[33] OpenSSH Development Team. Openssh 3.1.1p1 for linux. `www.openssh.org`.

[34] OpenSSL Development Team. Openssl 0.9.7c. `http://www.openssl.org`, 2004.

[35] R. Sekar P Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *the Proceedings of the 8th USENIX Security Symposium*, 1999.

[36] Wietse Venema. Postfix overview. `http://www.postfix.org/motivation.html`.

[37] David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *the Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2000.

[38] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *the Proceedings of the Symposium on Operating System Principles (SOSP)*, 1993.

[39] Kenneth Walker, Daniel Sterne, and Lee Badger. Confining root programs with domain and type enforcement (DTE). In *the Proceedings of the 6th USENIX Security Symposium*, 1996.

[40] John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In *the Proceedings of the International Symposium on Software Testing and Analysis*, 2002.

[41] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew Myers. Secure program partitioning. In *Transactions on Computer Systems*, 2002.

[42] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQual for static analysis of authorization hook placement. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.

[43] Lantian Zheng, Stephen Chong, Andrew Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *the Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.