Towards Automated Privilege Separation

Dhananjay Bapat, Kevin Butler, and Patrick McDaniel

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802 USA
dbapat,butler,mcdaniel@cse.psu.edu

Abstract. Applications are subject to threat from a number of attack vectors, and limiting their attack surface is vital. By using privilege separation to constrain application access to protected resources, we can mitigate the threats against the application. Previous examinations of privilege separation either entailed significant manual effort or required access to the source code. We consider a method of performing privilege separation through black-box analysis. We consider similar applications to the target and infer *states* of execution, and determine unique *trigger* system calls that cause transitions. We use these for the basis of state-based policy enforcement by leveraging the *Systrace* policy enforcement mechanism. Our results show that we can infer state transitions with a high degree of accuracy, while our modifications to Systrace result in more granular protection by limiting system calls depending on the application's state. The modified Systrace increases the size of the Apache web server's policy file by less than 17.5%.

1 Introduction

Applications in computing environments are often insecure. Insecure applications open the system and user data to exploitation. The number of attack vectors, from buffer overflow attacks to worms and other malware, is large and growing. However, the system can defend itself against attack and reduce the amount of potential damage by limiting an application's access to trusted system resources.

In Unix systems, an application runs at the same privilege level as that of the user executing it. Often, this gives applications more privileges than they need to perform their tasks. Mitigating the exposure surface of applications to attack can be achieved by adhering to the principle of least privilege [22, 21], restricting access to resources such that only the parts of the application that require them are eligible to request them.

While mechanisms to control the interaction between the application and its environment exist, they are often onerous to enact on the part of the user. For example, virtual machines provide isolated environments but require the user to be running within the machine and administer the virtual machine manager. Mechanisms such as *chroot* and the *Tripwire* utility [12] provide notifications of abnormal activity but require constant user vigilance and cognizance of these

notifications. In addition, these mechanisms are specific to a user's particular machine; no guarantees are made if the user moves to a different computer, e.g., from an office to a home machine. Often, as a result, the best place to minimize privilege is within the application itself. This is best demonstrated through the concept of *privilege separation*, which refers to separating the parts of the application that run at different privileges [18]. In a privilege-separated application, the privileged portions of the program handle sensitive inputs, while the other parts of the program have their access restricted to sensitive resources. Because Unix treats different processes as protection domains, even if a less privileged portion is compromised by a malformed input, user can't take control of the privileged portion. Hence, the risk of data compromise is greatly reduced. Provos et al. manually separated the privileged part of the OpenSSH program from the less privileged part, requiring considerable effort. Efforts to automate the process of privilege separation include Brumley et al.'s *Privtrans* work [3]. However, these processes all require access to source code, which may not be available in closed-source systems or for legacy applications.

In this paper, we consider how to automatically perform privilege separation within applications by considering application execution as a series of states. At differing points during the execution, different levels of resource access will be required by the application. We examine a variety of web server applications and show that by understanding a subset of them, we can accurately identify state transitions common to the vast majority of web servers, thus allowing black box analysis of software we do not possess the source to; our tests show that measured by the number of system calls performed, our inferred transitions occur less than 170 system calls, or less than 0.5% of the total number of measured system calls, from where they would occur if we had access to application's source code.

We leverage the differing requirements as applications transition between states to enforce policy at the system call level through use of the *Systrace* policy enforcement framework [17]. Systrace provides simple policies for governing system call access, and enforces only application policies, unlike more heavyweight solutions such as SELinux. There are two ways of achieving certain security objectives with systrace: it can be run interactively, allowing a user to allow system calls as they are invoked, or it can be run in automatic mode, where a pre-existing policy is enforced. By using program states to determine what system calls should and should not be allowed at various stages of a program's execution, we can enforce these policies with Systrace and maintain least privilege within the application. Our modifications to Systrace, making it aware of states and more granular in its enforcement over the course of an execution, adds less than 17.5% to the size of the Systrace policy file for the Apache web server.

The rest of the paper is structured as follows: Section 2 presents related work and concepts; Section 3 introduces our methodology for examining states and how we correctly find state transitions and validate our findings; Section 4 describes how we integrate policy semantics discovered from state transitions

into Systrace; Section 5 provides an evaluation of our modified Systrace in terms of policy file size and performance; Section 6 concludes.

2 Related Work

Running an untrusted application on a machine opens it to data and system compromise. As a result, application confinement is an area of sustained research. Systems may be protected by a myriad of defenses: active measures constantly check the application's capability to perform certain operations, while passive measures use resources such as like system logs and call traces to identify intrusion attempts. One of the traditional security tools used for these purposes is *Tripwire*, which monitors critical system files and directories and identifies changes made to them. I^3FS [11] improves on Tripwire with a kernel level implementation that provides real-time access checking. A parallel area of research looks at the application-data relationship for better security. Formal methods of preserving information integrity include the Clark-Wilson [5] and Chinese Wall [7] models, as well as sub-operating systems [9, 19].

System calls are important to identify malicious behavior, as they are the only way for an application to get access to the privileged kernel operations. Forrest et al. [8] used system call monitoring to identify intrusion detection attempts in a system, creating a system call database of normal behavior and comparing the active system call trace to the database. Any deviation from the database indicated ongoing intrusion. Authenticated system call mechanisms [20] augment system calls with extra arguments that specify policy. An HMAC guarantees integrity of the system call policy and arguments. However, this approach does not take into account malicious user input, nor does it protect the system from an inherently malicious application. System call interposition [10, 24] allows for the system to intercept system calls and *sandbox* the application by mediating access to the rest of the system. Sandboxing has been extensively explored through the use of kernel modules [4], user-level OS extensions [1] and system call APIs [16]. In particular, virtual machines environments such as Xen [2] provide almost complete isolation between processes by running on logically different platforms, giving users the illusion of their own private machine. An exploit that targets the virtual machine manager, however, could put all virtual machines on the system at risk [13].

Confinement of processes is best effected through complete mediation of processes. The Flask architecture [23] and its successor, SELinux [14], implement a flexible policy enforcement infrastructure by assigning security identifiers to every object, and having a security server enforcing all accesses in a reference monitor like approach. SELinux in particular uses type enforcement and role-based abstractions to achieve a wide range of security objectives [15], the price being that one must run this operating system to gain these benefits. By contrast, the benefits of privilege separation of applications can be spread to any platform the application executes on.

Privileged and daemon programs in UNIX are the source of most security flaws, and the large codebase of application programs makes it difficult to identify those flaws. Fink et al. [6] use specifications against which a program is *sliced* to significantly reduce the size of code that needs to be checked for flaws [6]. Similarly, privilege separation creates a smaller trust base that is more easily secured. Provos et al. [18] demonstrated that SSH could be privilege-separated through extensive manual techniques. Brumley et al. considered automated privilege separation [3] and developed a prototype, which works on annotated source code for an application and creates a master and a slave application. Their tool, *Privtrans*, performs inter-procedural static analysis and C-C translation to achieve the goal. A disadvantage of this approach is that the authors of the application must conform and identify higher privileged variables and code for the tool to work. All of the privilege separation mechanisms discussed require access to the program's source code. We have built a privilege separation tool that performs a black-box analysis on an application, with policy enforcement provided by *Systrace* [17].

3 Privilege State Identification and Analysis

3.1 Introduction

Traditionally, privileges within an application were identified as either root or non-root privileges. We extend this concept by considering the state of an application to be its privilege level, such that every application can be described in terms of its state machine. State analysis is an important step to analyze the operation of applications, as understanding valid and invalid states in an application can help the developer identify transitions leading to an error state. While privilege separation in previous approaches was achieved by physically separating parts of application, we can achieve similar results by identifying states in an application and by enforcing a specific application policy for each of the states.

States in an application can be identified by looking at the source code and by identifying major steps an application takes. However, we wanted our approach to be usable for legacy applications where source code may not be available, such that a black-box analysis would be necessary. Hence, to identify states in an application, we looked at the externally observable behavior of an application, i.e., system call traces. System calls are gateways to privileged operations of the kernel and hence can give information about the state of the application and any child processes spawned by it.

3.2 Environment

To collect system call traces, we used a Linux machine running the Debian/GNU 2.4.27 distribution. System call traces were generated with the *strace* utility found in Linux, which intercepts and records the system calls and signals respectively called by and received by an executing process. We selected a set of

Table 1. Web server: Number of unique system calls per state

Application	Start	Listen	Accept
Apache	64	11	12
Caudium	134	33	16
dhttpd	8	29	7
lighttpd	21	5	7
luahttpd	50	0	11
nullhttpd	16	4	16
thttpd	49	18	8
xshttpd	25	1	29

web server applications to perform system call analysis on because of their wide usage and the ability to easily differentiate sets of distinct states, simplifying the state analysis. The list of web servers examined can be found in Table 1, and represent a diversity in functionality, from the lightweight, simple functionality of *lighttpd* to the full-featured Apache server.

We generated traces of server executions under a variety of configurations, and used Perl scripts to collect and analyze the trace data. Systems calls of similar functionality were grouped for identification purposes (e.g., the `lstat`, `stat`, and `read` calls were mapped to the `fsread` system call). We found that system call traces were largely independent of changes in server configuration by changing numerous variables such as timeout period, maximum number of child processes, listening port, and password-protecting some files. Our results showed that 93% of system calls remained the same for Apache Web server while 92% remained the same for the Caudium Web server. While system call traces will vary with the kind of workload[1] run on the Web server, our representative workloads capture all system calls made by the server (we explain issues relating to coverage in Section 4).

3.3 Observations

To identify states, we first looked to find system calls that can be used to indicate that state transitions have occurred; we term these system calls *triggers*. We attempted to find individual system calls rather than a sequence of system calls to act as trigger. We also do not consider arguments to the system call for simplicity. Certain system calls occur rarely during the Web server's execution. For example, in the Apache Web server, the `listen` system call occurs only once. Since it is a significant event and can be related to the operation of Web server, it can be considered a trigger. We identified system calls that were rarely called during application execution, and used this set of calls to form an initial set for determining triggers.

[1] A web server's workload refers to factors such as the number of requests and amount of data served, and how frequently these requests occur.

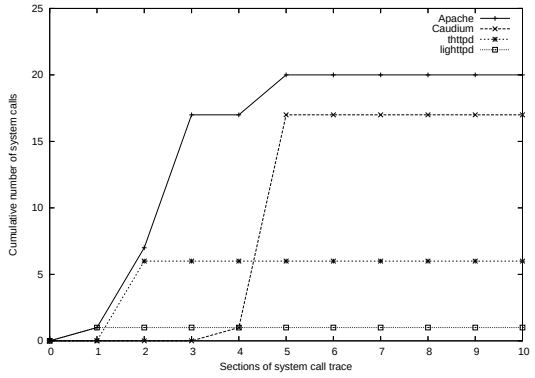

Fig. 1. Cumulative number of `bind` system calls for variety of Web servers.

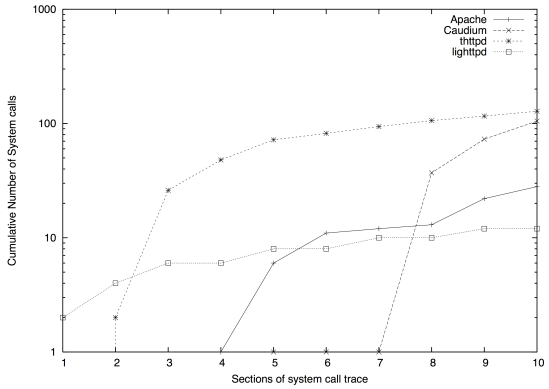

Fig. 2. Cumulative number of `accept` system calls for variety of Web servers.

Even if a system call is only rarely called, it may not be a good indicator of a state transition if it occurs throughout the server's execution. An important factor in determining whether a system call is a trigger is the point in the application's execution where the call occurs, and whether it repeats (i.e., the *locality* of the call relative to application execution). To further investigate this, we collected system call traces of multiple web servers given the workload of a single user browsing a variety of web pages. We divided the execution trace into ten equal stages to simplify discussion about where state transitions occur. Figures 1 and 2 show when during execution the `bind` and `accept` system calls respectively occur. Note that the large number of accept calls and the variance of this number between web servers requires Figure 2 to be shown on a log scale, reflecting the diverse implementations and methods of responding to requests between web servers. Of equal interest is noting where in the trace the calls occur. For the `accept` system call, the first calls happen early in the trace with

lighttpd and *thttpd*, while they do not occur until sections 4 and 7 for Apache and Caudium, respectively. In particular, note that unlike other applications, *lighttpd* calls `accept` in the first section of the trace; this may be a function of its quick startup time, whereas with larger servers such as Apache, many library calls and other startup procedures must occur before it is ready to accept connections. By contrast, `bind` system calls start early in the execution trace for most web servers. Both system calls are distinctive and are good candidates for a trigger.

There are other system calls that can act as a trigger. Some of them are not uniform throughout the Web server class of applications, e.g., `select`. Some of them occur too close to one another to create a distinguishable state, i.e., `bind` and `listen` occur in tandem. Applications can also use different system calls to execute similar functionality, e.g., `send`, `sendto` and `sendfile` are all used to send the requested Web page to the client. Some system calls, e.g., `socket`, do not provide sufficient information unless arguments to the call are also considered. As a result, much of the necessary information can only be inferred by studying a variety of applications of the same class. We inferred web server states by examining the applications identified in Table 1.

3.4 State Analysis

States of an application should possess features that distinguish them from neighboring states. States may differ in ways that include the types of resources accessed, type of user interface presented, amount of resources accessed, amount of network activity, or the type and number of system calls called. These unique characteristics support the idea that each state in the application is a privilege level.

We identified three states for Web servers: `start`, `listen`, and `accept`. We also identified the following three rules of the state transition:

1. Application starts in Start state.
2. *if (state == 'start') and if (systemCall == listen) then state = 'listen'*
3. *if (state == 'listen') and if (systemCall == accept) then state = 'accept'*

This choice of states was justified by our discovery of many system calls occurring in these states that do not occur in any other state of the application.

After identifying state from the system call trace, we separated system calls made in each state and determined the number of system calls that unique to each state of the application. Table 1 shows number of unique system calls per state for the applications we tested. We consider a system call to be unique if using the *Systrace* utility, a new policy is created for the call in automatic policy generation mode. As previously mentioned, we correlated system calls with similar functionality and only counted them once to eliminate redundancy (e.g., `lstat6` and `stat64` are both mapped to `fread` in *Systrace* and hence, are only counted once). In addition, individual file accesses on the web server are mapped to one system call that accesses the root of the website.

Table 2. Difference (in number of system calls) between our inferred state transition and where the transition actually occurs in code.

Web server	tclhttpd	Abyss	Boa	Cherokee
Listen	0	4	0	11
Accept	24	0	21	126

Unique system calls do not provide information about the size of the state. For example, while applications spend most of their time in the `accept` state, more unique system calls are called from the `start` state. A large number of policy statements exist for `start` because of the number of library file accessed in this state: unique system calls are generated for every path location examined by the application. Applications spend the least amount of time in the `listen` state, and for servers that do not create multiple child processes for servicing requests, the number of system calls is very low in this state.

3.5 Verification

We verified the inferred state engine on a different set of web servers, as shown in Table 2. The `listen` transition occurs when the application has loaded all of its libraries and is opening a socket to listen for incoming connections. An `accept` state transition coincides with the server entering an unbounded loop where it handles incoming HTTP requests. We inserted dummy system calls in the code for the two new web servers where server initialization is performed and where the server begins its loop of accepting connections, to determine how close our state identification came to the transition in the code itself. Table 2 describes the offset (in the number of system calls) between where we inferred a state identification and where the code transition occurred. The transition into the `listen` state is detected at the same time it occurs in code. Detection of the accept state is more variable, but the offset is still less than 0.5% of the total system calls observed in the trace. Thus, there is a high correlation between our inferred state transitions and their actual occurrence in code, making black-box analysis possible. In the next section, we use these state transitions to demonstrate how policy enforcement can be implemented.

4 Implementation

Having described how to identify states in an application, we are interested in using these as part of a policy enforcement infrastructure. We leverage the *Systrace* policy enforcement framework, which we use because of its clean policy semantics. Systrace policies describe the desired behavior of user applications on the system call level, which are enforced to prevent operations that are not explicitly permitted. Our prototype only modifies the policy semantics of Systrace while keeping the enforcement infrastructure intact.

While policy creation is usually relegated to the user, Systrace is capable of creating policies automatically and interactively. Policy is created by enumerating all possible actions that an application will need for its correct execution. Each policy statement can be evaluated by itself; thus, it is possible to extend the policy just by appending new statements. In automatic mode, policy is created by running an application and recording the system calls that it executes. Systrace checks the existing policy; if none exists then one is created. If a policy exists and the system call is not covered by any of the existing policy statements, a new policy statement that allows the system call is appended to the policy. Automatic mode creates a comprehensive policy only if all execution paths are covered in the training run. Interactive mode of operation asks the user for the policy decision for every system call for which a policy statement can't be found in Systrace policy. This mode is useful if the user doesn't trust the source of the application and wants to check the access to the system resources before any harm is done; it is also useful for ensuring complete code coverage. Once a security policy for an application has been finalized, policy enforcement is employed. When an application attempts to execute a system call, the user is not asked for a policy decision. If Systrace can find a policy statement related to the system call, system call is allowed; otherwise, it is denied and an error code is returned to the application.

To create privilege separation in an application, we use the states identified in a program's execution as the basis for different policy enforcement parameters, such that at different points through the execution, we can constrain access to resources. The goal of our implementation is to integrate Systrace with our privilege state engine to simplify privilege separation in the application. This entails modifying Systrace to understand the concept of states and vary enforcement mechanisms depending on the state of the application. We describe the original Systrace architecture and our modifications in the following sections.

4.1 Systrace Design

Systrace uses a hybrid approach to system call interception. A small kernel part supports fast path for some system calls that are always allowed or denied. The kernel part is used to make Systrace fail-safe. When a monitored application executes a system call, kernel consults a small in-kernel policy database to check if system call should be permitted or denied. System calls like read, write are always permitted. In interactive mode, policy decision is deferred to the corresponding user space daemon. When policy daemon receives a request for policy decision, it looks up policy associated with the process. Kernel keeps track of all new processes; child processes inherit policies of the parent. If Systrace is terminated, all the processes Systrace was monitoring are also terminated, thus eliminating chance of an adversary circumventing Systrace. In absence of a fast path, Systrace asks the user space daemon for policy decision. Systrace blocks the process until the daemon returns with an answer explicitly allowing or denying the system call. The user space daemon uses kernel interface to monitor processes, get pending policy decisions and state changes. Before making a policy

decision, system call arguments are translated by Systrace. As a result, system call semantics don't affect system call decisions. Translation of
`socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);` takes form:
`socket: sockdom: AF_INET, socktype: SOCK_RAW`.
File-name arguments are translated to resolve all the symbolic links. This eliminates another avenue for an adversary to evade the application policy enforcement.

Policy Specifications

Systrace uses an ordered list of policy statements, with one statement per system call. A policy statement is a Boolean expression, B, combined with an action clause: *B* then *action*. Ask, deny and permit form the valid action set. If the Boolean expression evaluates to true then the specified action is taken. The ask action requires user to deny or permit the system call explicitly. Boolean expression consists of variables Y_n and logical operators: *and*, *or* and *not*. Variables Y_n are tuples of the form *(subject op data)* where *subject* is translated name of system call argument, *data* is a string argument and *op* is a function with a Boolean return value. To create different policies for different users, policy statements may carry predicates. Predicates are appended to the policy statement. They are of the form *if* {*user,group*} *op data*, where op is equality or inequality and *data* is a username or a group name. Predicates are matched first before any policy is evaluated. A log modifier can also be added to the policy statement to create a granular audit trail for a particular system call.

Policy Evaluation

A set of policy statements forms the security policy. Policy evaluation starts at the start of the list and ends when the first Boolean expression is true. The action from that policy statement determines if the system call is allowed or denied. If no Boolean expression becomes true then policy decision is forwarded to the user in interactive mode and denied in the enforcement mode. When a system call is denied, policy can specify which error code is to be passed to the application. Systrace supports interactive and automatic policy generation. In automatic policy generation, an application is run and Systrace records all the system calls it generates and creates policy to allow all those system calls. On following runs, the automatically created policy is used. While creating policies automatically, it is assumed that application itself isn't malicious. Interactive mode is used to make sure that all execution paths of the application are covered by the Systrace policy. If a policy statement can't be found for a system call then user is presented with a graphical notification that contains all the information about the system call. The user then explicitly allows or denies that system call.

Exhaustive Policy Creation

A good policy should allow only those system calls necessary for the intended functionality of the application and deny everything else. Each policy statement is evaluated by itself, so it is possible to extend a policy just by appending additional policy statements. A good policy is thus created by listing all the possible actions that an application needs. To create an exhaustive policy with automatic mode of Systrace, it is necessary to cover all execution paths in the training mode. If it is not possible to foresee or execute all the execution paths during training runs, then application should be run in interactive mode to create a policy with complete coverage. A user can use generic policy templates for some applications. These templates can be used as a starting point. After the security policy has been finalized, automatic policy enforcement can be employed. In such a case, system call is denied if a policy statement allowing it is not found.

4.2 Systrace Modifications

Systrace lists a set of statements that form policy for an application. We modify semantics of the Systrace policy to make it understand and interpret the concept of state. Original Systrace policy semantics look like this:

```
[<Boolean expression> then] <action>
[if \{user,group\} <op> <data>][log]
```

For modified Systrace application most of the policy statements will have at least one Boolean expression, which will be of the form:
`state eq "<state-name>"`.
The modified Systrace code has a state engine that will keep track of the privilege state of the application and help interpret and create policy statements.

We have created our prototype in NetBSD stable 3.0.1. Systrace code is integrated with NetBSD distribution. Size of code for Systrace in that distribution is 18047 lines of C code. Our modifications added 415 lines of code to various files in the source tree. We treat our state variable as a part of Boolean expression and it is added to most of the policy statements. When system call arguments are translated by Systrace, state information is added to the system call's translated argument queue. As a result, state is treated as any other system call argument. In the automatic mode, when the policy is written to a file, a Boolean expression for state is also added to the policy statement. In the enforcement mode, the state string found in policy is matched with data in the translate queue of the system call. If both strings match then the system call is allowed, otherwise it is denied.

A simple state engine is incorporated in the implementation. It takes in the current state, the current system call and the application type as input and it outputs the next state for the application. State engine for each application type is learnt by studying 4-5 applications of that type. Because all these applications exhibit similar functionality, they also possess similar privilege states. For example, a Web server application will typically start up, read the libraries, get DNS information from local DNS server, open a listening socket, and serve all

Table 3. Increase in policy size with modified Systrace

Application	Original Systrace	Modified Systrace
Apache	223	262
Bozohttpd	138	144
Mini-httpd	149	158
thttpd	140	150
w3c-httpd	191	197
lighttpd	114	134

the incoming connection. State engine design can be improved to give us an optimum set of state-system call relationship in terms of number of system calls per state and the uniformity across different applications.

Systrace modifications integrate the policy enforcement mechanism with the principles from privilege separation. The modifications do not physically separate privileged part of the application; they rather enforce the privileges by Systrace's policy enforcement architectures.

Modified Systrace policy creates more granular Systrace policy by introduction of privilege states. Because the policy is defined for every system call called from that state, we in tern are defining privileges for different time-periods in the lifetime of the application.

5 Evaluation

We created a prototype of the modified Systrace. In this section, we evaluate different characteristics of the modified Systrace. The number of policy statements is an important performance criterion, as the larger the policy file, the more time is required for the policy engine to search for a policy statement for a given system call. We created policies for couple of Web servers with our modified Systrace. The policies were then enforced in the enforcement mode of Systrace. Table 3 shows the policy size for the old and the new Systrace policy. The workload represents a single user choosing successive HTML documents from a web server. The policies were created using the automatic policy generation mode of Systrace. Policies increase in size with the addition of privilege state to the system call policies, and increased by an average of 15 statements. Size is minimized by introducing the privilege state for some of the system calls like `mmap`, `munmap`, `getuid`, or `getgid`. These system calls occur throughout the lifetime of the application and mostly deal with low-level memory management or provide information about user and group. Adding privilege state to their policy statement only increases the size of policy with no real advantage.

States of an application represent different privilege levels for that application. Each state has a unique signature in terms of type of system calls made in that state. We also considered uniqueness of a privilege state as an evaluation

criterion for the modified Systrace. The uniqueness is measured in terms of the percent of unique system calls called in that state. The results are laid out in table 4. [2]There are a high number of unique system calls in the `accept` state because one policy statement is created for each HTML file served. The `start` state also has a high percentage of unique system calls because it is the only state where all the library files are accessed. These results are similar to the unique system call values shown in table 1 which were obtained analysing the system call traces.

Table 4. Unique Policy Statements for Each Privilege State

Application	Start	Listen	Accept
Apache	56	15	69
Bozohttpd	18	3	91
Mini-httpd	27	10	86
thttpd	32	7	74
w3c-httpd	8	7	149
lighttpd	20	12	66

We evaluated the performance of Systrace and of our modified version. Benchmarking was done by ApacheBench. An Apache Web server was installed on a NetBSD machine and the workload was simulated with fifty total requests and twelve concurrent requests. Without Systrace running, the time per request with Apache averaged 5.01 ms. While running under Systrace, an average request took 8.6 ms to serve. Performance under the modified Systrace was further degraded, with an average request taking 13.61 ms to serve, almost 1.5 times longer than for the unmodified Systrace. However, this performance penalty is not because of unnecessary context switches but because of the time taken by the system to access and manage the TAILQ structures for each system call. Each system call in Systrace has a system-call argument TAILQ associated with it. In the original Systrace, arguments of most of the system calls were ignored for the purpose of application policy creation. However, in modified Systrace, each system call has at least 1 argument, i.e., the state of the application in which the system call is called. Systrace manages arguments for approximately 30 different system calls while Modified Systrace manages arguments for approximately 80 system calls. Hence, the number of TAILQs to be initiated and managed has increased by almost 2.5 times. We see this performance degradation because we want to integrate the application specific state engine with minimal changes in original Systrace as possible. We defer consideration of performance improvements for future work.

[2] Number of unique policy statements are comparable to the numbers in table 1. In table 4 accept state has more unique policy statements because no wildcards were used in any of the policy statements.

While creating the application policy with the modified Systrace, we only looked at web servers. We plan to extend this idea to other classes of applications as well. Web servers are chosen for the initial study because of their ubiquity and because the application class is not dominated by one particular vendor/application. This gave us opportunity to study a variety of Web servers, which are being used in the real world. Web servers also fit well in the state based application policy because it is easy to understand the states in a Web server. Further types of servers such as mail and DNS, and other server applications will be considered in future work.

6 Conclusion

Securing applications is a two-step process involving creating an effective policy enforcement mechanism as well as creating good policies. We also simplified privilege separation processes through inferring state transitions with high accuracy, enabling black-box analysis of the application. In addition, we found that by modifying the Systrace utility, we could provide policy enforcement over the defined states with a less than 17% increase in the size of the Systrace policy file for the Apache web server.

There are several directions of future work based on this study. We have only looked at the class of web server applications. State-based privilege separation techniques can also be extended to and demonstrated for other classes like database servers, mail servers and the most downloaded user applications like music players, web-browsers, IM clients. In future we also plan to compare this privilege separation technique to results from PrivTrans and manual privilege separation.

References

1. A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. *Proceedings of the USENIX Annual Technical Conference*, 1998.
2. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM Press, 2003.
3. D. Brumley and D. Song. Privtrans : Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72, 2004.
4. S. Chari and P. Cheng. BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System. *ACM Transactions on Information and System Security*, 6(2):173–200, 2003.
5. D. Clark and D. Wilson. A Comparison Of Commercial and Military Computer Security Models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, April 1987.

6. G. Fink and K. Levitt. Property Based Testing of Privileged Programs. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 154–163, 1994.
7. S. Foley. Building Chinese Wall in Standard Unix. *Computers and Security Journal*, 16(6):551–563, December 1997.
8. S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6:151–180, 1998.
9. S. Ioannidis, S. Bellovin, and J. Smith. Sub-operating systems: A New Approach to Application Security. In *Proceedings of the 10th ACM SIGOPS European Workshop: Beyond the PC*, pages 108–115, 2002.
10. K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 2000.
11. A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I^3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, 2004.
12. G. H. King and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Symposium on Communications and Computer Security (CCS'94)*, Fairfax, VA, USA, Nov. 1994.
13. S. T. King and P. M. Chen. SubVirt: implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2006.
14. P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX track of 2001 USENIX annual technical conference*, pages 29–42, 2001.
15. P. Loscocco and S. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.
16. D. Peterson, M. Bishop, and R. Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002.
17. N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, August 2003.
18. N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, August 2003.
19. M. Radhakrishnan and J. Solworth. Application Security Support in the operating system kernel. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, pages 201–211, 2006.
20. M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated System Calls. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 358–367, 2005.
21. J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of The IEEE*, volume 63, pages 1278–1308, September 1975.
22. F. Schneider. Least Privilege and More. In *IEEE Security and Privacy*, volume 1, pages 55–59, 2003.
23. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and J. Lepreau. The Flask Security Architecture: System Support For Diverse System Policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, August 1999.
24. D. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical report, CSD-99-1056, 1999.