# IMPLEMENTING A SECURE SETUID PROGRAM

Takahiro Shinagawa
Dept. of Computer, Infomation and Communication Sciences
Tokyo University of Agriculture and Technology
2-24-16 Naka-cho, Koganei-shi, Tokyo 184-8588, Japan
email: shina@cc.tuat.ac.jp

Kenji Kono
Department of Computer Science
University of Electro-Communications
1-5-1 Chofugaoka Chofu-shi, Tokyo 182-8585, Japan
email: kono@cs.uec.ac.jp

**ABSTRACT**
Setuid programs are often exploited by malicious attackers to obtain unauthorized access to local systems. Setuid programs, especially owned by the *root* user, are granted root privileges, allowing attackers to gain root privileges by exploiting vulnerabilities in the setuid-root programs. The vulnerabilities usually lie in code that does not require root privileges. Nevertheless, the entire code of setuid-root programs is granted root privileges. This paper presents a scheme called *privileged code minimization* that reduces the risk to setuid programs. In this scheme, setuid-root programs are divided into *privileged code* and *non-privileged code*. Privileged code is granted root privileges, while non-privileged code is not. This scheme reduces the size of trusted computing base (TCB) because it reduces the code running with root privileges, reducing the chances of attackers gaining root privileges by subverting setuid programs. Protection between privileged code and non-privileged code are enforced by fine-grained protection domains: a novel protection mechainsm of the operating system proposed by the authors.

**KEY WORDS**
Security, Operating System, Fine-grained Protection Domain

## 1 Introduction

The risk to setuid programs in UNIX have been well-known over the years. This type of program is executed with the privileges of the *owner* rather than those of the *user* executing it. In particular, *setuid-root* programs are executed with the privileges of the super-user, regardless of who is executing them. Therefore, malicious users often attempt to exploit security flaws in setuid-root programs to gain root privileges, especially after cracking a target machine over the Internet by hacking into users' accounts. Because these programs are a part of a trusted computing base (TCB), they should not contain security flaws that could be exploited in attacks. Unfortunately, all setuid programs are not robust enough to be free of security flaws. To date, many of these programs have been reported to contain numerous security flaws; in the last year CERT/CC Vulnerability Notes [1] reported 5 and Security Focus Vulnerabilities [2] reported 65.

In a setuid program the entire code is executed with the privileges of the owner of the program. However, the code that actually needs these privileges is only a *small* portion of the code; the remaining portion can be executed with ordinary users' privileges. For instance, `/bin/passwd`, a setuid-root program for changing users' login passwords, is granted root privileges to modify entries in the `/etc/passwd` file. Although the entire code for `/bin/passwd` is executed with root privileges, only the code that actually modifies password files needs these privileges. All other parts of the code do not need any root privileges. According to our analysis of the security notes [1, 2], most vulnerabilities in setuid programs lie in codes that have root privileges but do not need them.

In this paper, we present a scheme called *privileged code minimization* to improve the robustness of setuid programs. In this scheme, the code executed with the owner's privileges is *minimized* to only a small portion of the code, *not* the entire code of the program. In case of setuid-root programs, the code executed with root privileges is only a portion of the code that actually needs root privileges. All the other portions of the code are executed with ordinary privileges, i.e., the privileges of the user who started the program. As a result of minimizing privileged code, attackers cannot gain root privileges even if security flaws in non-privileged code are exploited. In addition, the scheme contributes to reducing the size and complexity of the TCB because only the code executed with the root privileges, not the entire code, of setuid-root programs must be trusted.

Several setuid programs partially incorporate the idea of privileged code minimization by dropping its privileges as soon as possible after the privileged operations are finished. These programs use *uid-setting system calls* [3], such as `seteuid()`, to temporarily drop the privileges. For example, a setuid-root program can drop root privileges by calling `seteuid()` and execute the remaining code with ordinary privileges. Unfortunately, dropping the privileges using uid-setting system calls is not enough to prevent attackers from gaining privileges because attackers can *regain* the privileges by calling uid-setting system calls again; attackers can induce setuid programs to issue such system calls by injecting malicious code. To prevent such attacks, privileged code must be clearly divided and protected from non-privileged code so that non-privileged code can't gain privileges.

To implement protection between privileged code and non-privileged code, we use the mechanism of *fine-grained* protection domains, which have been proposed by the authors [4, 5, 6]. This mechanism allows multiple protection domains to co-exist inside a single process. Each protection domain is associated with a code fragment of the process and granted its own access rights. Therefore, different portions of the program code can be executed with privileges different from those granted to other portions of the code. To protect privileged code from non-privileged code, we assign a fine-grained protection domain to each of privileged and non-privileged code. A fine-grained protection domain associated with privileged code grants owner's privileges, while a fine-grained protection domain associated with non-privileged code does not. Therefore, only privileged code can perform privileged operations: non-privileged code is prohibited to perform privileged operations, including calling uid-setting system calls. The memory data of privileged code is also protected from non-privileged code by fine-grained protection domains to prevent non-privileged code from subverting privileged code.

A great deal of research effort has been devoted to developing security mechanisms to restrict the access rights of applications. Sandbox systems [7, 8, 9, 10, 11, 12, 13] allow us to restrict the privileges of the *entire* program, but it cannot grant different privileges to different portions of a program. Even if a setuid-root program is sandboxed, the program code is granted the root privileges it requires to accomplish its intended job. If the `/bin/passwd` is compromised, it could destroy all the entries in the password file even if properly sandboxed. Java allows each class to be granted its own privileges but it would be unrealistic to rewrite all setuid programs from scratch in Java because almost all of these are written in C. Although the guidelines for writing secure setuid programs [14, 15] are also known, the growing size and complexity of programs greatly hinders following these too closely.

In the previous short paper [16], we only show a brief summary of the basic idea of privileged code minimization. This current paper is an extended version of the previous paper and thoroughly describes the detail of the design, implementation, and evaluation of our approach to making setuid programs more secure.

The paper is organized as follows. Section 2 describes the existing techniques used to attack setuid programs and it analyzes vulnerabilities of setuid programs. Section 3 introduces our scheme called privileged code minimization. Section 4 is an overview of the mechanism of fine-grained protection domains. Section 5 describes the implementation of privileged code minimization. In Section 6, we apply the scheme to the `passwd` command. Section 7 describes related work and Section 8 concludes the paper.

## 2 Vulnerabilities of Setuid Programs

Before describing privileged code minimization, we will summarize the techniques used to subvert setuid programs and provides an analysis of vulnerabilities in setuid programs reported to CERT/CC Vulnerability Notes in 2001. Our analysis suggests that vulnerabilities exist in code that have root privileges but do not require them and this finding supports our claim that minimizing privileged code would reduce the risk of attackers gaining root privileges through subverting setuid programs.

### 2.1 Attacking Techniques

The techniques used to subvert setuid programs can be classified into three: 1) buffer overflows, 2) code substitution, and 3) symlink attacks. One of these techniques is used in almost all the attacks on setuid programs that have been reported to CERT/CC Vulnerability Notes and Security Focus Vulnerabilities (Table 1).

**Buffer Overflows** Buffer overflow [17] is one of the most common techniques of subverting programs. In this technique, an attacker injects and executes attack code by overflowing a buffer that has weak bounds checking on its input. By overflowing the buffer the attacker can overwrite the adjacent part of the program's state such as return addresses and function pointers, and make the program jump to the injected attack code. For instance, an attacker injects a code similar to "exec(/bin/sh)" to a setuid-root program. Since the injected attack code is running with the privileges of the vulnerable setuid program, the attacker successfully gains a root shell.

**Code Substitution** Code substitution is a technique of executing an arbitrary attack code with the privileges of setuid programs. In code substitution, an attacker replaces some programs called from a setuid program with prepared attack programs. For example, the attacker alters the `PATH` environment variable to replace external commands with attack programs, or changes the `LD_LIBRARY_PATH` environment variable to replace standard libraries with attack code. Since the program called from a setuid-root program is executed with root privileges, the attacker successfully gains these by substituting code.

**Symlink Attacks** Symlink attacks are a technique of gaining unauthorized access to privileged files. In these, an attacker prepares a symbolic link to a privileged file using the name of the file that the setuid program uses as a temporary file. During the execution of the setuid program, the privileged file located by the symbolic link is overwritten and the contents of the file are destroyed. Using this technique, the attacker can conduct a kind of denial-of-service attack. For example, if the `/etc/passwd` file is deleted by the symlink attack, no user can login to the targeted machine.

Table 1. Vulnerabilities of setuid programs reported to CERT/CC Vulnerability Notes

| ID | system | program | vulnerability | vulnerability factor | privileges |
|---|---|---|---|---|---|
| VU#123651 | IBM AIX | lsfs | code substitution | external command exec. | devices |
| VU#860296 | CDE | dtprintinfo | buffer overflow | user inputs processing | files |
| VU#105347 | UNIX | xmcd | symlink attack | temporary file | devices |
| VU#916443 | Digital Unix | msgchk | buffer overflow | argument processing | network ports |
| VU#440539 | Digital Unix | msgchk | symlink attack | argument processing | network ports |

## 2.2 Vulnerability Analysis

Table 1 shows the vulnerabilities of setuid programs, which were all reported to CERT/CC Vulnerability Notes in 2001. As shown in the column of *vulnerability*, these setuid programs are vulnerable to one of the attacking techniques described above.

The *privileges* column shows that these programs require root privileges to access privileged devices, files, or privileged network ports. However, as the *vulnerability factor* column reveals, the vulnerable portions of the program code do not require any root privileges. For example, the `lsfs` command, listed at the top of Table 1, requires root privileges to access storage devices to obtain information on the file system such as its size and permission. The vulnerability in the code is in the portions that execute external commands `lslv` and `grep` to list logical volumes and parse the resulting output. These commands could be executed with ordinary users' privileges and thus code that executes external commands should not be granted root privileges. Similarly, vulnerable code requires no root privileges for the other vulnerabilities listed in the table. The processing of user inputs and command line arguments could be performed without root privileges. Also, a temporary file can be created with the privileges of ordinary users.

This finding that vulnerabilities exist in code that has root privileges but does not require them was also observed in the 65 vulnerabilities reported to Security Focus Vulnerabilities. Therefore, as previously stated, minimizing privileged code would effectively reduce the risk of attackers gaining root privileges through subverting setuid programs and potentially vulnerable code could be made to run with ordinary privileges. Even if attackers successfully subverted setuid-root programs, they would only be able to gain privileges granted to ordinary users.

# 3 Privileged Code Minimization

This section describes our scheme of *privileged code minimization* to improve the robustness of setuid programs. To achieve this minimization, the portion of the program code that actually requires root privileges is granted these, and the other portions are only granted those for ordinary users. Code granted root privileges is called *privileged* while that not granted these is called *non-privileged*.

## 3.1 Extracting Privileged Code

In minimizing privileged code, the programmers of setuid-root programs first identify the collection of procedures that require root privileges to complete their operations. These procedures are classified into privileged code, and all others are classified into non-privileged code. In Figure 1, `procedure1` and `procedure2` are classified into privileged code and `procedure3-6` are classified into non-privileged.

To reduce the chances of privileged code containing security flaws, the programmers must be careful to *minimize* its size. Which procedures should be included in the privileged code depends on the operations that setuid programs perform. An example of this minimization is `passwd` command which is granted root privileges to modify the password file. The code that requires root privileges is the procedure `change_passwd` that actually modifies the password file and all the other procedures, which process users' inputs and command line arguments, are classified into non-privileged code. When modifying the existing code, the programmers may be forced to redesign the structure of the program to cleanly extract the privileged code.

## 3.2 Defining the Interface

After extracting the privileged code, the programmers define the interface between the privileged and non-privileged code. As will be described later in Section 5, non-privileged code is forced to call privileged code only via this interface. As Figure 1 illustrates, only calls via the interface are allowed, while calls that bypass it are denied. If non-privileged code attempts to bypass the interface and directly jump into privileged code, the attempt is detected and our system raises a runtime exception. The implementation details are in Section 5.

To ensure the safety of setuid programs, the programmers must carefully design and implement the interface between the privileged and non-privileged code. If an attacker gains full control of the non-privileged code, she attempts to gain the privileges of the privileged code by attacking the interface. If the `change_passwd` procedure in the `passwd` command does not verify an old password before a password is changed, the attacker can change the root password to gain root privileges, immediately af-

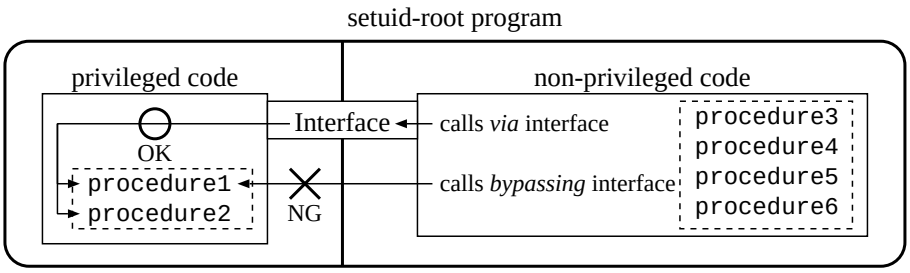


Figure 1. Privileged code minimization: privileged code is granted root privileges and non-privileged code is only granted ordinary user privileges. In this program, procedures 1-2 are classified into privileged code and procedures 3-6 are classified into non-privileged code. A procedure in non-privileged code is forced to call the procedures in the privileged code only via the interface. Our system prevents the interface from being bypassed.

ter gaining control of the non-privileged code. Thus, the `change_passwd` procedure should take three arguments: the old password, the new password, and the login name. Not only must the interface be properly designed but it must also be implemented correctly. If there are security flaws in the implementation, an attacker can hijack the privileged code by using one of the techniques described in Section 2. For example, an attacker can attempt to inject and execute attack code by overflowing the buffer.

The primary advantage of privileged code minimization is that programmers can concentrate on the design and implementation of the interface between the privileged and non-privileged code. Without it, programmers must correctly implement the entire code of a setuid program and carefully validate that there are no security flaws remaining. While this is theoretically possible, it is almost impossible in practice as the numerous reports on security flaws suggest. Minimizing privileged code reduces the chances of security flaws remaining, because it reduces the size and complexity of the TCB and makes it easier to validate — manually or automatically — the correctness of the implementation.

### 3.3 Protecting Privileged Code

Even if we minimize privileged code in setuid programs, attackers may subvert and take control of non-privileged code, and then attempt to gain privileges of privileged code. To prevent attackers from subverting privileged code, it must also be protected from non-privileged code. In our implementation, the operating system kernel prevents non-privileged code from bypassing the interface, and from accessing memory areas that are private to the privileged code. This is achieved by using the mechanism for fine-grained protection domains and we provides an overview of this in the next section.

## 4 Fine-grained Protection Domains

Before describing how to minimize privileged code, we will briefly summarize the mechanism for fine-grained protection domains. The fine-grained protection domain is a kernel-level extension, currently implemented as an extension to the Linux kernel. Although it is very versatile protection mechanisms, we only introduce the functionalities of the mechanism needed to explain how privileged code is minimized. The details of fine-grained protection domains should be referred to the papers [4, 5, 6].

### 4.1 Execution Model

Traditionally, the notion of the process has coincided with that of the protection domain and therefore, the entire process code is granted the same privileges to access computing resources. It is almost impossible to grant different privileges to different portions of the code executing within a single process.

The mechanism for fine-grained protection domains enables different portions of the code within a single process to be granted different privileges to access computing resources. Unlike the traditional model, the notion of a protection domain is decoupled from that of the process. Multiple protection domains called *fine-grained* protection domains co-exist inside a single process and can be granted privileges that are different from those granted to other domains in the same process. Figure 2 shows how two fine-grained protection domains $A$ and $B$ co-exist inside a process. In this figure, domains $A$ and $B$ are granted different privileges to access files. Domain $A$ is granted the right to access file `/etc/passwd`, because it is allowed to open it with the `RW` mode. However, domain $B$ is not allowed to open `/etc/passwd`.

Every fine-grained protection domain is associated with a code fragment and determines the access rights of the associated code. In Figure 2, the procedure `change_passwd` is associated with domain $A$ and the procedures `main` and `check_args` are with domain $B$.

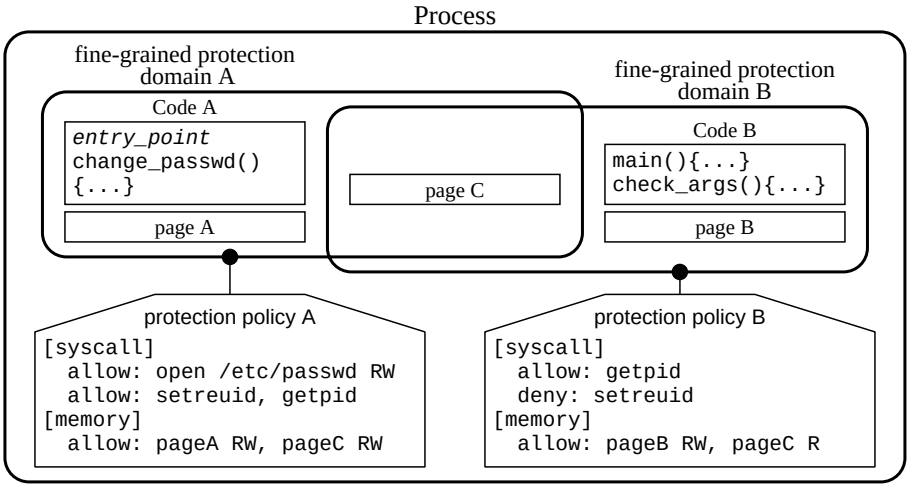


Figure 2. Fine-grained protection domains: two fine-grained protection domains $A$ and $B$ co-exist inside a process. Each fine-grained protection domain is associated with a protection policy. A protection policy defines 1) which system calls can be issued, 2) which memory pages can be accessed, and 3) which entry points of other domains can be invoked. A thread executing in domain $B$ can switch to domain $A$ by calling the entry point (`change_passwd`) of domain $A$.

When a thread is executing `change_passwd`, it is granted the access rights of domain $A$. Similarly, when the thread is executing `main` or `check_args`, it is granted the access rights of domain $B$. Therefore, the procedure `change_passwd` can read and write `/etc/passwd` that the procedures `main` and `check_args` cannot.

A thread executing in the one fine-grained protection domain can switch to another domain by calling an entry point of the domain. In Figure 2, a thread executing in domain $B$ can switch to domain $A$ by calling the entry point (`change_passwd`) of domain $A$. No thread can switch to any other domain without invoking entry points. A runtime exception is raised if a thread executing in domain $B$ attempts to jump into code associated with domain $A$ without invoking the entry point.

## 4.2 Protection Policy

A protection policy is associated with each fine-grained protection domain to determine their access rights. A protection policy defines 1) which system calls can be issued, 2) which memory pages can be accessed, and 3) which entry points of other domains can be invoked. Although we have developed various tools to define protection policies in ways that are more concise and abstract, this section describes the system call-level primitives directly supported by the kernel to define these policies.

For system calls, we can define which system calls can be issued and what arguments are allowed to these. For example, we can define a policy in which the `open` system call is only allowed if the file to be opened is under the `/tmp` directory. In Figure 2, fine-grained protection domain $A$ is allowed to issue `open`, `setreuid` and `getpid` system calls. With `open`, the path name must be `/etc/passwd` since the procedure `change_passwd` only accesses `/etc/passwd`. On the other hand, domain $B$ is only allowed to issue the `getpid` system call. When a system call is issued in a domain, the kernel checks if the protection policy associated with the domain permits the domain to issue the system call with the passed arguments.

To provide intra-process memory protection, protection policies can define which memory pages are readable, writable, or executable. Figure 2 shows the memory pages $A$ and $B$ are private to fine-grained protection domains $A$ and $B$, respectively. Memory page $C$ is shared by domains $A$ and $B$. Domain $A$ is allowed to read and write the page, while domain $B$ is only allowed to read the page. To prevent entry points from being bypassed, the code area of each domain must be inaccessible by other domains. If memory protection is violated, the memory management unit (MMU) detects the violation and raises runtime exceptions. For security reasons, a stack is allocated for each fine-grained protection domain.

To control domain switching, we can define which domains can invoke which entry points. Figure 2 shows that fine-grained protection domain $B$ is allowed to invoke the entry point of domain $A$. Calling an entry point must be done under the control of the kernel, because it involves switching domains and thus switching the access rights of the calling thread. To switch fine-grained protection domains, we prepared a software trap specialized in switching domains. To call an entry point, a thread presents the identifier of the invoked entry point to the kernel, and causes the specialized software trap. The kernel checks if it is permitted to switch from the calling domain to the called domain, and it then upcalls the entry point, if it is permitted.

### 4.3 Implementation Overview

The fine-grained protection domain is implemented as an extension of the Linux kernel. Its implementation centers around the *multi-protection* page table, a mechanism that accomplishes the intra-process memory protection. A multi-protection page table is an extension of the traditional one that enables each memory page to have multiple protection modes at the same time. At any instance in time, one of the multiple protection modes is effective. Since each memory page can have multiple protection modes, each fine-grained protection domain can have its own protection mode for each memory page, and thus memory protection is provided between fine-grained protection domains. To switch fine-grained protection domains, we prepared a software trap specialized in switching domains in which the effective modes to protect memory are switched. By utilizing the memory management unit (MMU) judiciously, we avoided flushing TLBs and caches that would have degraded performance.

Our extended kernel is ported to a wide range of processor architectures. The prototype system is running on Intel IA-32, SPARC, and Alpha processors. To implement multi-protection page tables efficiently, we fully exploited processor-specific features. We used segmentation and ring protection mechanisms for the Intel IA-32 family [18]. We utilized tagged TLBs and register windows for the SPARC processors (versions 8 and 9) [19]. We also used tagged TLBs and PAL code for the Alpha processors [20]. We did not extend the kernel a great deal. We only added 1,398 lines of code to the Linux 2.2.18 in the IA-32 implementation. The details are discussed in our papers [4, 6, 5].

## 5 Implementation

This section describes how privileged code can be minimized. Through the mechanism of fine-grained protection domains, we only grant root privileges to privileged code and this prevents non-privileged code from using root privileges. A setuid-root program starts execution with ordinary user privileges, and privileged code gains root privileges, as needed, by issuing system calls to change the effective user ID to root. However, non-privileged code is not permitted to issue these system calls so that it cannot use root privileges.

### 5.1 Associating Fine-grained Protection Domains

To grant different privileges to privileged and non-privileged codes, we prepare two fine-grained protection domains for each. To prevent non-privileged code from accessing privileged code, it is not permitted 1) to access to the memory areas (including code, data, and stack sections) of the privileged code, 2) to issue system calls such as `setreuid()` that change the effective user ID, and 3) to change the protection policy of fine-grained protection domains. Details on the protection policies will be described in the next section.

Fine-grained protection domains are associated with privileged and non-privileged codes before a setuid program is executed. To associate domains before execution, we modified the program loader (an ELF loader for Linux) so that it could associate fine-grained protection domains with the privileged and non-privileged codes. After associating the domains, the loader sets up the protection policies of both domains. Then, it changes the effective user ID of the program from the root to an ordinary user and jumps into the `main` function after the remaining initialization.

The information needed to associate fine-grained protection domains, such as the addresses of entry points and the memory areas of the privileged code are explicitly specified by programmers using special directives. The entry points are specified using the directive `entry_point` and the memory areas are specified using the directive `privileged_section`. These directives are implemented using GCC extensions and they encode the information into the special ELF sections in the program file. These ELF sections are valid only if the setuid bit of the program file is on and the file is write-protected from other users. In addition, the type of ELF file is limited to the normal executable file: shared libraries are not allowed to contain privileged code to avoid the problem of the code substitution attack.

### 5.2 Protection Policies

As previously mentioned, a setuid-root program begins execution with ordinary users' privileges. To allow privileged code to execute with root privileges, it is permitted to issue system calls such as `setreuid()` to change the effective user ID to root. In the privileged code, a `setreuid()` system call is issued to gain root privileges as needed, and before returning to the non-privileged code the effective user ID is changed to an ordinary user ID from the root. However, non-privileged code is not permitted to issue these system calls so that it cannot execute with root privileges.

To prevent non-privileged code from accessing privileged code, it is denied access to the memory areas (including code, data, and stack sections) of the privileged code. To call privileged code, the non-privileged code is forced to call a pre-defined entry point for the privileged code. Since an entry point may take the arguments of pointer types (typically implemented as virtual addresses), the privileged code is permitted to access all non-privileged code memory areas. By doing this, the privileged code can access the data referenced by the pointer passed from the non-privileged code.

The protection policies themselves should not be changed by the non-privileged code. It is denied system calls related to setting up protection policies.

### 5.3 Calling Privileged Code

To allow non-privileged code to call privileged code, the functions defined in the interface between the privileged and non-privileged codes are registered as entry points for the fine-grained protection domain associated with the privileged code. The privileged code is invoked by calling an entry point for the domain. Then, the privileged code gains root privileges by issuing a system call such as `setreuid()` to change the current effective user ID to the root, and it continues execution with root privileges. Before returning to the non-privileged code, the effective user ID is restored to the ordinary user's by issuing the `setreuid()` system call.

## 6 Application

To demonstrate the usefulness of privileged code minimization, we applied our scheme to `passwd` commands. In the following, we discuss the results.

### 6.1 Minimizing Privileged Code

To minimize privileged code, we extracted it from the entire code of the `passwd` command. As mentioned in Section 3.1, code classified as privileged in `passwd` is only code that actually modifies password files. The other code is classified as non-privileged.

To invoke a privileged portion of the code from the non-privileged code, we defined the interface as follows:

```
/* the interface to change passwords */
int change_passwd(user, oldpw, newpw);
char *user;    /* user name */
char *oldpw;   /* old password */
char *newpw;   /* new password */
```

This interface changes the password of a user specified as `user` to a new password `newpw` only if the current password of the user specified as `oldpw` is correct. In `passwd` commands, `change_passwd()` is the only interface that can be invoked from non-privileged code. Therefore, no operations, except for changing passwords, can be performed with root privileges.

The implementation of this interface gains root privileges, as needed. In our implementation, `user` and `oldpw` are checked with normal users' privileges. If they are authenticated, the effective user ID is changed to root by calling the `setreuid()` system call and the new password is written to the password file. Thereafter, the effective user ID is restored to the normal user's and control is returned to the non-privileged code.

Privileged code is protected by fine-grained protection domains. Non-privileged code is denied system calls to change effective user IDs. It is also denied memory access to privileged code to avoid subverting privileged code. Therefore, attackers can not gain root privileges even if they exploit security flaws in the non-privileged code.

### 6.2 Protecting against Existing Attacks

In this section, we demonstrate that `passwd` command whose privileged code is minimized can protect against existing attacks. First, even if buffer overflow attacks against non-privilege code are successful in the `passwd` command, attackers fail to gain root privileges, because non-privilege code runs with normal user privileges. Although attackers can invoke the interface `change_passwd()` from non-privileged code, they can not change the password of any user unless they know their current passwords.

As discussed in Section 2, vulnerabilities tend to exist in code that has but does not requires root privileges. Therefore, potentially vulnerable code is classified as non-privileged code. In addition, since the privileged code is smaller and can be implemented more carefully than before, it contains fewer security flaws. Therefore, the chances of privileged code containing vulnerabilities is remarkably reduced.

Other attacks such as code substitution and symlink attacks can also be protected against. Since executing external command and making temporary files are normally performed in non-privileged code, these attacks fail to gain root privileges.

### 6.3 Overhead Assessment

To estimate the overheads resulting from minimizing privileged code, we measured the time it took to change a user's password. To reduce the effect of user interaction, the new password was read from a file. The experiments were conducted on a Pentium III 1GHz PC with 128MB memory, running the Linux 2.2.18 kernel modified to support the fine-grained protection domains.

The `passwd` command with the privileged code minimization took 22.1 ms to change the password, while an unmodified `passwd` command takes 21.3 ms. The overhead was 3.8% or 0.8ms. This included the cost of creating fine-grained protection domains, setting up protection policies, and performing cross-domain calls between the privileged code and the non-privileged code. The experimental results suggested that the overheads to minimize privileged code is acceptable.

## 7 Related Work

The necessity for writing secure setuid programs has long been recognized and Bishop [14] gave guidelines as long ago as 1987. General guidelines for writing secure programs have also been widely promulgated. Wheeler [15] provides a set of design and implementation guidelines to write secure programs for UNIX systems. Saltzer and Schroeder [21] discuss some principles of information protection such as "principles of least privilege". Although these guidelines are useful, they are not sufficiently effective by themselves to eliminate all risk to setuid programs.

Because of the growing size and complexity of programs, it is becoming increasingly more difficult to follow these guidelines in the manner in which they were intended.

Many sandbox systems [7, 8, 9, 10, 11, 12, 13] have been developed to restrict the access rights of applications. Sandbox systems reduce privileges granted to programs, thereby reducing the damage inflicted by attacks even if the programs contain security flaws. However, sandbox systems reduce the privileges of the *entire* code of the programs. Our scheme, however, only grants privileges to the *portion* of the code that actually needs them. Only granting privileges to a small portion of the code effectively reduces security flaw because many of these are in code that does not requires these privileges.

POSIX [22] defines the capability that divides root privileges into a series of capabilities and it only grants programs a subset of root privileges. The POSIX capability can reduce the root privileges granted to setuid programs. However, it does not allow root privileges to be granted to only a small portion of the code. The POSIX capability still allows attackers to gain and exploit reduced root privileges if the security flaws in the code are not eliminated.

Java allows a small portion (a class in Java) of the code to be granted its own privileges. However, it would be unrealistic to rewrite all setuid programs from scratch in Java. Almost all the existing setuid programs are written in C because they are extremely dependent on the functionalities of the underlying UNIX systems. Our scheme aims to improve the robustness of setuid programs written in C.

Language-independent approaches such as code certification [23, 24] and SFI [25] can be applied to improving the robustness of setuid programs. Code certification ensures type safety in language-independent ways. However, type safety is not enough to protect against attacks such code substitution and symlink attacks. SFI [25] provides intra-process memory protection by directly modifying binary code. However, SFI does not address the issues of access control of setuid programs.

Several operating systems [26, 27, 28, 29] provide protection mechanisms for component-based systems. These mechanisms may also be used to implement protection between privileged code and non-privileged code in the privileged code minimization.

## 8 Conclusion

This paper proposed *privileged code minimization*, a scheme that improves the robustness of setuid programs. In this scheme, code that runs with root privileges is minimized to the portion that only actually requires these root privileges. According to analysis, most vulnerabilities in setuid programs are in code that does *not* require root privileges. Therefore, minimizing privileged code reduces the risk to setuid programs because attackers cannot gain these privileges even if they exploit the vulnerabilities of code not running with root privileges.

In this paper, we demonstrated that privileged code minimization can be implemented by utilizing fine-grained protection domains, a mechanism that enables different parts of a single program to be granted different privileges. To demonstrate the feasibility of our scheme, we applied privileged code minimization to the `passwd` command, a setuid-root program for changing login passwords. We also demonstrated that the overheads were acceptable.

In future work, we will present concrete examples of applying the scheme of privileged code minimization to several existing setuid programs. Considering the way of dividing a setuid program into privileged code and non-privileged code and defining the interface between them is difficult but important work to prove the concept of this paper. Although the way will varies from programs to programs, we believe that some guideline common to many programs will be presented.

## References


[1] CERT/CC Vulnerability Notes. http://www.kb.cert.org/vuls/.

[2] SecurityFocus Vulnerabilities. http://online.securityfocus.com/bid.

[3] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proc. of the 11th USENIX Security Symposium*, August 2002.

[4] Masahiko Takahashi, Kenji Kono, and Takashi Masuda. Efficient kernel support of fine-grained protection domains for mobile code. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems* (*ICDCS '99*), pages 64–73, May 1999.

[5] Takahiro Shinagawa, Kenji Kono, Masahiko Takahashi, and Takashi Masuda. Kernel support of fine-grained protection domains for extention components. *Journal of Information Processing Society of Japan*, 40(6):2596–2606, June 1999. (in japanese).

[6] Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Department of Information Science, Faculty of Science, University of Tokyo, May 2000. An extended version of the previous paper [5].

[7] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proc. of the 6th USENIX Security Symposium*, July 1996.

[8] Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proc. of the 9th USENIX Security Symposium*, August 2000.

[9] Lincoln D. Stein. SBOX: Put CGI scripts in a box. In *Proc. of the 1999 USENIX Annual Technical Conference*, June 1999.

[10] Massimmo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proc. of the 7th ACM Conference on Computer and Communications Security* (*CCS '00*) , pages 174–183, November 2000.

[11] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proc. of the 2nd International System Administration and Networking Conference (SANE)*, May 2000.

[12] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined execution environment for internet computations. Available at http://www.cs.ucsb.edu/ berto/papers/99-usenix-consh.ps, 1998.

[13] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement UNIX prototype. In *Proc. of the 5th USENIX UNIX Security Symposium*, June 1995.

[14] Matt Bishop. How to write a setuid program. *;login:*, 12(1):5–11, Jan/Feb 1987.

[15] David A. Wheeler. Secure programming for linux and unix HOWTO. http://www.dwheeler.com/ secure-programs/.

[16] Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. Minimizing privileged code in setuid programs using fine-grained protection domains. *Computer Software*. A short paper, to appear (in japanese).

[17] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition*, pages 1119–1129, January 2000.

[18] Intel Corporation. *The Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. 1999.

[19] SPARC International. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, 1992.

[20] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, 1995.

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

[22] Institute for Electrical and Electronic Engineers. *Information Technology – Portable Operating System Interface (POSIX)*. IEEE Computer Society, 1997. IEEE Std 1003.1e-1997.

[23] George C. Necula and Peter Lee. Safe kernel extensions without runtime checking. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pages 229–243, October 1996.

[24] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, May 1999.

[25] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pages 203–216, December 1993.

[26] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proc. of the 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), pages 140–153, December 1999.

[27] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of $\mu$-kernel-based systems. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), pages 66–77, October 1997.

[28] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, and Susan Chambers, Craig an Eggers. Extensibility, safety and performance in the spin operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (*SOSP '95*), pages 267–284, December 1995.

[29] Greg Law and Julie McCann. A new protection model for component-based operating systems. In *Proc. of the 19th IEEE International Performance, Computing, and Communications Conference* (*IPCCC '00*), February 2000.