A Secure Access Control Mechanism against Internet Crackers

Kenichi Kourai and Shigeru Chiba

January 11, 2001

ISE–TR–01–176

Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba 305 Japan

A Secure Access Control Mechanism against Internet Crackers

Kenichi Kourai Shigeru Chiba

January 11, 2001

Abstract

Internet servers are always in danger of being "hijacked" by various attacks like the buffer overflow attack. To minimize damages in cases where full control of the servers are stolen, imposing access restrictions on the servers is still needed. However, designing a secure access control mechanism against hijacking is not easy because that mechanism itself can be a security hole. In this paper, we describe the access control mechanism of our Compacto operating system. Compacto uses our new technique called *the process cleaning* so that malicious code injected by a cracker cannot illegally remove access restrictions from a hijacked server. According to the results of our experiments, the process cleaning can be implemented with acceptable performance overheads.

1 Introduction

Internet servers, such as web servers and mail servers, are always in danger of attacks by crackers. Their typical attack is the buffer overflow attack [19], which injects malicious code into a server and obtains full control of the server, that is, "hijacks" it. Another attack is to abuse a server plug-in or a Common Gateway Interface (CGI) program, which can maliciously hijack a server [2]. Once a server is hijacked, the cracker can use the server for performing malicious operations. To protect the servers from these attacks with negligible cost, several techniques such as StackGuard [8] has been developed [9, 3, 14]. These techniques are for detecting the buffer overflow attack and invalidating it. Since those technique cannot detect all types of buffer overflow attack, imposing access restrictions on a server is still necessary. Access restrictions minimize damages by the attack in cases where the server is unfortunately hijacked.

However, it is difficult to design a secure access control mechanism against hijacking. An access control mechanism should prevent hijacked servers from illegally removing access restrictions and obtaining higher privileges for accessing system resources. On the other hand, it must allow legitimate servers to remove access restrictions if they need higher privileges. Unfortunately, it is difficult to determine whether a server is hijacked or not; even if the server has not been hijacked yet, malicious code might have been already injected and it might be activated later for hijacking the server.

In this paper, we present an access control mechanism provided by the Compacto operating system, which we are developing. It allows the users to impose access restrictions on a particular process. They can prohibit a process from issuing specific system calls and restrict the range of parameters that a process can pass to a system call. To prevent hijacked servers from illegally removing access restrictions, we have developed a technique called *the process cleaning*. If a hijacked process attempts to remove access restrictions, Compacto recovers the process from malicious code that has hijacked it. It first resets the thread of control so that the execution of the malicious code terminates. Then it restores the state of the process including a memory image and thereby eliminates malicious code from the process.

We also describe the implementation of the process cleaning. Performance overheads due to the process cleaning mainly come from restoring a memory image when access restrictions are removed. To reduce the overheads, Compacto allows the users to choose a strategy for restoring a memory image. To show performance improvement by this technique, we measured the performance of the Apache web server running on Compacto. We describe the results of this experiment and discuss the overheads of the process cleaning.

The rest of this paper is organized as follows. Section 2 describes security risks caused by removing access restrictions. Section 3 presents the process cleaning technique, which makes it secure to remove access restrictions, and describes details of our implementation of the process cleaning. Section 4 shows the results of our experiments for measuring overheads due to the process cleaning. Section 5 discusses the related work. Section 6 concludes this paper.

2 Access Restrictions

The Compacto operating system, which we are developing, allows the users to impose access restrictions on a server process. With this facility, Compacto can protect the rest of the system if the server is "hijacked," for example, by the buffer overflow attack. Since preventing all hijacks is not realistic, access restrictions are still necessary for minimizing damages. Suppose that a hijacked server attempts to modify a security-related system file. If the user prohibits the server from issuing the `write` system call on that system file, Compacto can deny that modification. Compacto also provides the `setuid` system call originating from UNIX. It is used for giving a server process a lower access privilege, which corresponds to the specified user.

2.1 Changing Access Restrictions

If access restrictions imposed on a server cannot be changed during the execution of that server, several problems would happen in reality. Suppose that a web server is serving both the Internet and Intranet users. The administrators would want to impose more access restrictions on the server while it is serving the Internet users. On the contrary, they would want to impose less while it is serving the Intranet users. For example, while the server is handling a request from an Intranet user `ken`, it should be able to read a file that only the user `ken` can read so that `ken` can browse the contents of that file through the web.

Elevating the privileges of a process needs to remove some access restrictions from the process. However, allowing the users to remove access restrictions implies security risks. At least, a hijacked server must be prevented from illegally removing access restrictions imposed on that server. For example, the `seteuid` (not `setuid`) system call provided by UNIX can be a security hole [18]. It is used for accomplishing the least privilege principle [16]. It temporarily lowers the privileges of a privileged server and later gives the original privileges back. Since it gives the original privileges back whether the process is hijacked or not, even a hijacked server may recover the original higher privileges.

Confirming that a server is not hijacked is not easy. A server that seems to be running normally may include malicious code for hijacking the server later. If this server is allowed to remove access restrictions, then the malicious code may be activated after the access restrictions are removed. The server's execution environment may be compromised. For example, if the variable `argv[0]` in a process is modified, a cracker can send a `HUP` signal to the process and thereby execute an arbitrary command indicated by that variable [5]. Detecting the existence of hidden malicious code and compromised execution environment is extremely difficult.

2.2 Spawning a Child Process

Since removing access restrictions implies security risks, a number of operating systems such as UNIX, in general, allow a process only to impose access restrictions. For example, the `setuid` system call provided by UNIX can change the access privilege from higher to lower but not from lower to higher. However, in spite of this limitation, a server running on those operating systems can still impose different access restrictions depending on a request from a client. If the server is connected from a client, then the server spawns a child process, imposes additional restrictions on that child process, and has the child process handle the request from the client. Removing the access restrictions from the child process is not necessary because the child process just terminates after handling the request. The server is kept running with the initial access restrictions and is securely protected since it does not receive any data from a client.

This technique, however, is not workable if the performance of the server is crucial. Since spawning and terminating a child process involves serious performance penalties, practical Internet servers uses the process pool technique [11]; the servers spawn several child processes in advance and repeatedly reuse them instead of spawning a new child process for every request. The child processes never terminate since they must handle the next request. Thus the technique for imposing different access restrictions on a newly spawned process for each request cannot be used together with the process pool technique. The implementor of a server running on UNIX must choose efficiency or security.

3 Process Cleaning

Compacto uses a new technique called *the process cleaning* so that removing access restrictions does not involve security risks mentioned in the previous section. Thereby the users can impose access restrictions on a server only while the server is handling a request from an untrustworthy client or it is executing an

```
save_state();                    (1)
accept();                        (2)
if (from_Internet)               (3)
    impose strong restrictions
else
    impose weak restrictions
handle a request
restore_state();                 (4)
```

Figure 1: A typical server using the process cleaning.

untrustworthy plug-in code. The imposed access restrictions minimize damages in cases where the server is hijacked and after that they are removed without security risks.

The access restrictions provided by Compacto are, for example, to disable some system calls. They also include changing the user/group ID of a process into a less privileged user/group ID, and changing the root directory for a process. These are implemented by the `setuid`, `setgid`, and `chroot` system calls as in UNIX although UNIX does not allow removing the access restrictions imposed by these system calls.

3.1 Recovering a Hijacked Process

For securely removing access restrictions, the thread of control must be recovered from malicious code injected by, for example, the buffer overflow attack. Then the malicious code must be eliminated from memory even if it has not been activated yet. To do this, Compacto provides the `save_state` system call, which saves the state of a process. This system call must be issued before access restrictions are imposed on a process. These access restrictions are removed if the `restore_state` system call is issued. This system call removes the access restrictions and also restores the saved state of the process. Since the saved state includes an instruction pointer and the memory image of the process, the thread of control is recovered and, if any, malicious code is eliminated from the memory.

For example, a web server running on Compacto uses the `save_state` and `restore_state` system calls as follows. After finishing initialization, the web server issues `save_state` system call (Fig. 1 (1)). Then the server waits until a client connects with it (Fig. 1 (2)). If a client connects, access restrictions depending on that client are imposed on the web server (Fig. 1 (3)). The web server handles a request from that client with those access restrictions. After the web server finishes handling the request, the web server issues the `restore_state` system call (Fig. 1 (4)). This system call recovers the state of the web server. Since it recovers the instruction pointer, the thread of control is moved back to the next statement of the `save_state` system call (Fig. 1 (2)). The web server repeatedly handle another request from a client.

3.2 Saved/Restored Process State

The `restore_state` system call restores the values of all the registers. It is analogous to `longjmp` provided by the standard C library. The restored registers include an instruction pointer (a.k.a. a program counter). If injected malicious code successfully obtains the control of the process, the execution of that malicious code is terminated when the `restore_state` system call is issued. Thus the malicious code cannot remove access restrictions without losing the control of the process. The malicious code can issue the `save_state` system call illegally. If it issues that system call, the process state saved by the previous issue is overwritten; the successive issue of the `restore_state` system call does not remove access restrictions imposed before the last issue of the `save_state` system call.

The `restore_state` system call also restores a memory image. This eliminates the Trojan horse, which is malicious code left on memory and later activated for hijacking, and restores the value of an environment variable recorded on memory if it has been modified since the last issue of the `save_state` system call. Restoring the whole memory image is necessary since distinguishing malicious memory accesses from regular memory accesses is extremely difficult. Restoring the whole memory image also keeps a server stable. If the server is attacked by a cracker, the cracker might collapse part of the memory image of that server and thereby cause a memory fault for terminating the server. A server running on Compacto is protected from this type of the denial-of-service attack since it can catch a memory fault and restore the whole memory image for repairing the collapsed image.

The `restore_state` system call also restores signal handlers. If there are pending signals, the `restore_state` delivers those signals before restoring signal handlers. The pending signals are delivered to correct handlers. If a signal handler is replaced with a malicious handler, a cracker could activate this malicious handler by sending a signal after access restrictions are removed. Then the malicious handler could hijack the server.

The `restore_state` system call closes files and sockets that have been opened since the last issue of the `save_state` system call. This is for avoiding the exhaustion of file descriptors, which a cracker may cause for the denial-of-service attack. If malicious code injected by a cracker closes a file or a socket, the `restore_state` system call opens it again and restores the file descriptor for it.

3.3 Restoring Memory

Performance penalties of the process cleaning are mainly due to copying memory for saving and restoring the state of a process. To reduce the amount of copied memory, Compacto uses a technique known as *copy-on-write* [4, 15]. Furthermore, the users can select an implementation strategy for restoring a memory image. As shown in Figure 1, a typical server running on Compacto saves its state only once and repeatedly restores the saved state whenever it finishes handling a request. The user can select an implementation strategy so that the process cleaning works efficiently in that case.

The `save_state` system call saves the memory image of a process so that the `restore_state` system call can restore it. However, that system call does not immediately duplicate the whole memory image. It first changes the state of

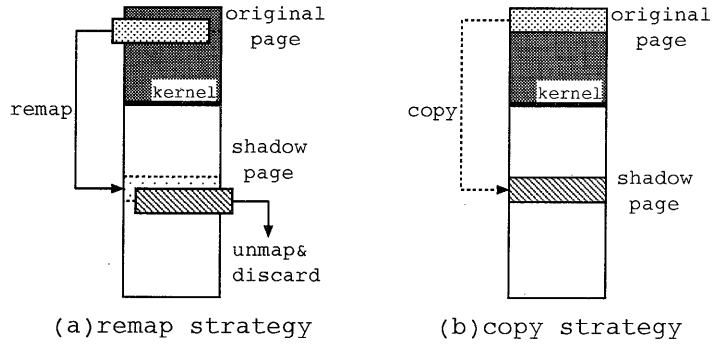

Figure 2: Two strategies for restoring memory.

every writable memory page into the write-protected mode. The memory page is duplicated only if the process attempts to write in the page and hence a page fault occurs. The page table is modified so that the original page is moved into the kernel address space and a new memory page allocated for the duplication is mapped at the original virtual address.[1] We call this new memory page a *shadow* page. Since the shadow page is writable, no page fault never occurs after the first one.

The `restore_state` system call restores only the memory pages that have been saved since the last `save_state` system call was issued. For restoring them, Compacto can choose one of two strategies. The first strategy is to unmap and discard a shadow page and move the original page back from the kernel address space. We call it the *remap* strategy (Fig. 2 (a)). The second strategy is to copy the contents of the original page into the shadow page. The original page remains in the kernel address space. We call this the *copy* strategy (Fig. 2 (b)).

Since the remap strategy does not need copying memory for the restoration, Compacto normally selects this strategy. It is also good with respect to memory consumption. However, the users can request Compacto to use the copy strategy. A typical server running on Compacto repeatedly restores the state saved by the `save_state` system call. The `save_state` system call does not alternate with the `restore_state` system call. In this case, the remap strategy may be less efficient. If Compacto selects the remap strategy, the `restore_state` system call must make the restored memory page write-protected so that it is duplicated again if the process attempts to write in that page between this and the next issue of the `restore_state` system call. If a page fault occurs, the page fault handler must allocate a shadow page and copy the contents of the page into it.

On the other hand, with the copy strategy, the restored memory page is still a shadow page; the original page is left in the kernel address space. Compacto does not have to make the restored page write-protected or to catch a page fault. Since the copy strategy reuses a shadow page, it causes a smaller number of page faults than the remap strategy. The number of memory copying depends on the pattern of memory access until the next issue of the `restore_state` system call.

[1] The saved memory image must be stored in the original page because the original page may be shared with parent and/or child processes for copying on write.

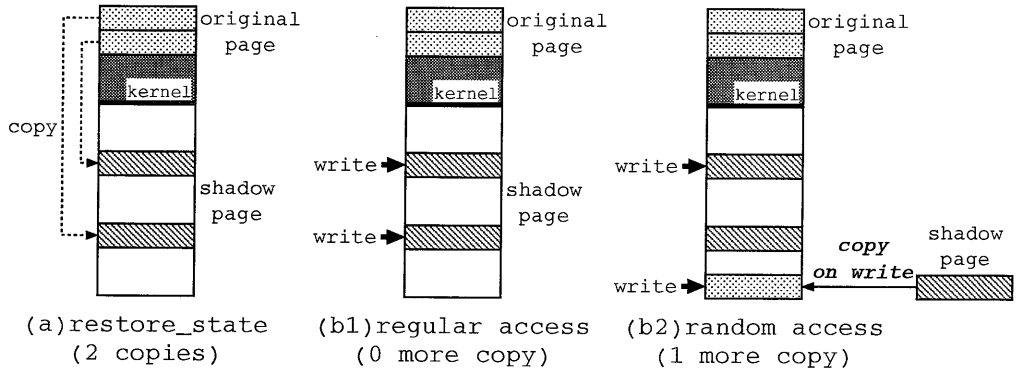

Figure 3: The performance of the copy strategy depends on a memory access pattern while a server handles a request. After the `restore_state` system call is issued (a), if the access pattern has locality (b1), the copy strategy is faster. Otherwise (b2), it is slower.

If the process writes in the same set of memory pages, that is, only the shadow pages, the copy strategy needs only the same number of memory copying as the remap strategy (Fig. 3 (b1)). Since no page faults occur in this case, the copy strategy is faster than the remap strategy. On the other hand, if the process writes in a totally different set of memory pages, maintaining shadow pages is useless and hence the copy strategy needs a larger number of memory copying. Compacto must copy the saved images to all the shadow pages for the restoration (Fig. 3 (a)) and duplicate other pages when page faults occur (Fig. 3 (b2)).

If the copy strategy is selected, the `restore_state` system call restores only the memory pages whose dirty bit is set. The dirty bit is set by the hardware if a process writes in the memory page associated with that dirty bit. After the restoration, all the dirty bits are reset so that Compacto can determine whether the page is written in between this and the next restoration. The `restore_state` system call does not copy a memory page whose dirty bit is clear. If the memory page has not been written in for long time, Compacto discards the shadow page of that page and reduces consumption of memory pages. It unmaps and discards the shadow page and moves the original page back from the kernel address space.

4 Experiments

We have developed the Compacto operating system on top of the Linux 2.2.16 kernel. This section reports the results of our experiments for measuring the overheads due to the process cleaning. The machine we used for the experiments is a PC with a Pentium III 933MHz processor[2] and 256MB memory.

Table 1: The execution time of the `save_state` system call.

# of pages	12	16	32	64	128	256	512	1024
μsec	5.29	5.62	6.34	7.91	11.4	18.1	28.8	56.1

Table 2: The cost of restoring the states of resources (μsec).

# of resources	0	1	2	4	8	16	32
memory pages (remap)	N/A	0.63	0.84	1.18	1.90	3.43	8.92
memory pages (copy)	N/A	1.59	3.13	9.15	17.9	35.2	146
signal handlers	0.08	0.27	0.33	0.38	0.60	1.05	N/A
open files/sockets	0.05	0.60	0.69	0.83	1.05	1.49	2.84

For memory, we did not measure the case where the number of memory pages is zero because at least one memory page is used for a stack frame and it must be restored.

4.1 Micro Benchmark

We first measured the execution time of the `save_state` and `restore_state` system calls. The execution time of the `save_state` system call depends on the number of writable pages in a process (Table 1). For example, the Apache web server uses approximately 60 writable pages. The total execution time includes the cost of saving the states of the resources other than memory pages (3.4 μsec): 2.12 μsec for signal handlers, 0.90 μsec for open files and sockets, 0.05 μsec for registers, and 0.33 μsec for issuing the system call. The influences of the number of files and sockets were negligible.

For executing the `restore_state` system call, 0.33 μsec is needed for issuing the system call and 0.05 μsec is for restoring registers. The rest of the execution time of the system call depends on the number of the restored resources as listed in Table 2. As for memory pages, the remap strategy needed a smaller cost than the copy strategy. However, the remap strategy may need extra costs. If the `save_state` system call is not issued and the restored memory page is updated again before the next issue of the `restore_state` system call, Compacto must catch a page fault and allocate a shadow page. This cost is listed in Table 3. Therefore, in the worst case, the remap strategy is 1.3 to 1.8 times slower than the copy strategy.

4.2 Apache Web Server

We also measured the execution performance of a web server running on Compacto. As the web server, we used the Apache 1.2.13 [1], which is implemented with the process pool technique. As client machines, we used PCs with a Celeron 300MHz and 64MB memory. The operating system of the client machines is

Table 3: The cost of handling page faults.

# of pages	1	2	4	8	16	32
μsec	2.21	4.96	13.1	26.6	52.8	185

[2]The L1 cache is 16KB (instruction) + 16KB (data). The L2 cache is 256KB.

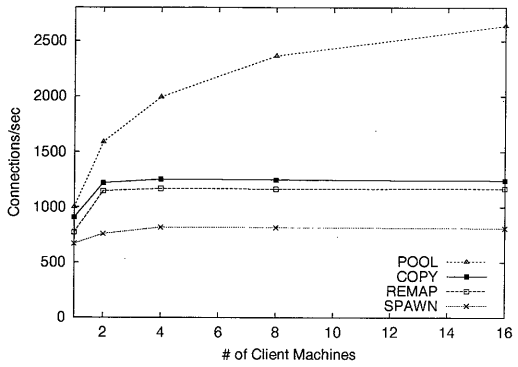

Figure 4: The server throughput (0 byte file was requested).

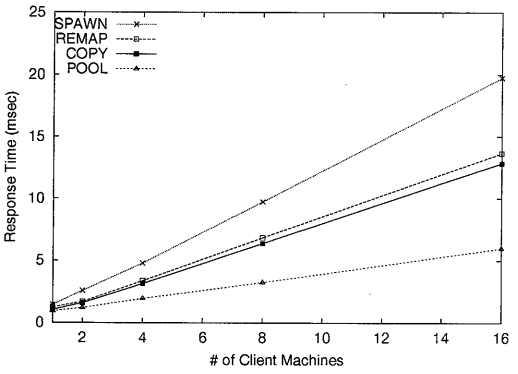

Figure 5: The average response time (0 byte file was requested).

FreeBSD 3.4. To avoid network saturation, the clients and the server are connected through the 100baseT Ethernet and the server machine has two Ethernet ports. We measured the execution time of the WebStone benchmark program [13] with various number of clients.

For comparison, we used four different types of Apache server. The POOL server is an Apache server which does not perform the process cleaning. The COPY server is an Apache server performing the process cleaning with the copy strategy. The REMAP server is an Apache server performing the process cleaning with the remap strategy. These three servers use 16 pooled processes. Finally, the SPAWN server is an Apache server modified for spawning a child process for every request. It does not use the process pool technique or perform the process cleaning. We did not impose access restrictions on any of the four Apache servers in this experiment.

4.2.1 Results

Figure 4 and Figure 5 show the server throughput (the number of acceptable requests per second) and the response time in the case that a 0 byte file was

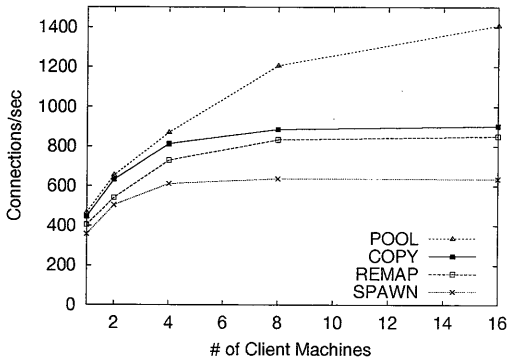

Figure 6: The server throughput (various sizes of files were requested).

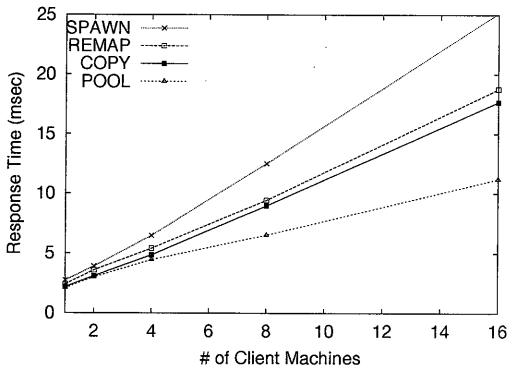

Figure 7: The average response time (various sizes of files were requested).

requested by the clients. All the servers modified 18 pages of memory, changed one signal handler, opened one file and one socket while handling every request.

Figure 6 and Figure 7 show the server throughput and the response time in the case that various sizes of files were requested. All the servers modified 18.3 pages of memory on average, changed one signal handler, opened one file and one socket while handling every request. The requested files were copied from our real web server[3]. The average file size was 7.6KB (from 73 bytes to 47KB).

4.2.2 Discussion

According to the results of our experiments, the COPY server is 1.5 times faster than the SPAWN server. Since secure servers, which impose access restrictions depending on each request, have had to spawn a new child process for every request as described in Section 2.2, the process cleaning achieves 50% improvement of the performance of those traditional secure servers. It allows the secure servers to handle a request with pooled processes although its performance penalties are not negligible if comparing the POOL server and the COPY server.

[3] `http://www.hlla.is.tsukuba.ac.jp/`

Since the process cleaning is CPU- and memory-intensive, the overheads of the process cleaning is relatively small if most of the execution time of a server is spent for disk and network I/O. Therefore, both the COPY server and the REMAP server showed better performance in Figure 6 than in Figure 4. Requests for large files causes more disk and network I/O than requests for 0 byte files.

In our experiments, the COPY server was always faster than the REMAP server. The performance improvement was 8% on average. In the case that a 0 byte file was requested, the same set of memory pages was updated whenever a request was handled. This is the case that the copy strategy achieves the best performance. In the case that various sizes of files were requested, the copy strategy is also better than the remap strategy although the memory access pattern of the servers was less advantageous to the copy strategy than in the former case. Although the copy strategy is better than the remap strategy in the case of the Apache web server, the remap strategy may be better in the case of other kinds of Internet servers.

The COPY server needs more memory pages than the REMAP server since it keeps the original pages in the kernel address space. To reduce memory consumption, the `restore_state` system call should unmap and discard some of the shadow pages and then move the original pages back from the kernel address space. We also measured effects of this strategy. The modified version of the `restore_state` system call unmaps a shadow page if the dirty bit is clear because we expect that the shadow page will not be modified. According to our experiment, the memory consumption of the modified COPY server was reduced by 7.7 pages on average. On the other hand, the number of page faults was increased. Thus the performance improvement against the REMAP server was decreased from 8% to 6%.

4.3 FastCGI

We also measured the execution performance of the FastCGI module [10], which runs on top of the Apache web server. The FastCGI module enables the server to use pooled processes for running CGI programs. Without the FastCGI module, the server must spawn a child process whenever a CGI program runs. As a CGI program, we used wwwcount 2.5 [20], which is one of the most popular access counters.

Like the experiment in the previous subsection, this experiment compared four web servers: the POOL, COPY, REMAP, and SPAWN servers. Only the FastCGI module used by the COPY and REMAP servers performs the process cleaning. The SPAWN server does not use the FastCGI module. It spawns a child process for running a CGI program. The underlying servers of the four are the normal Apache server.

Figure 8 and Figure 9 show the results. These results are similar to the results in the case of the Apache web server. However, the performance overhead due to the process cleaning was smaller in this experiment. The COPY server was only 15% slower than the POOL server. This is because executing a CGI program was a bottleneck and hence the overheads due to the process cleaning were relatively small.

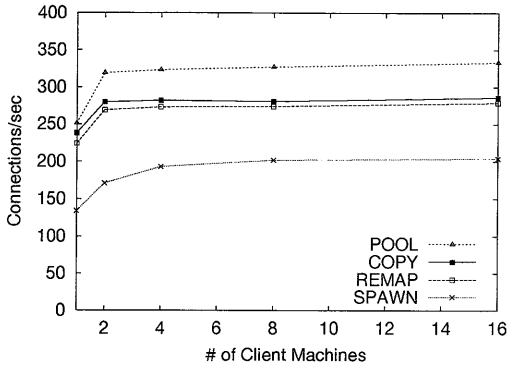

Figure 8: The server throughput (CGI program was requested).

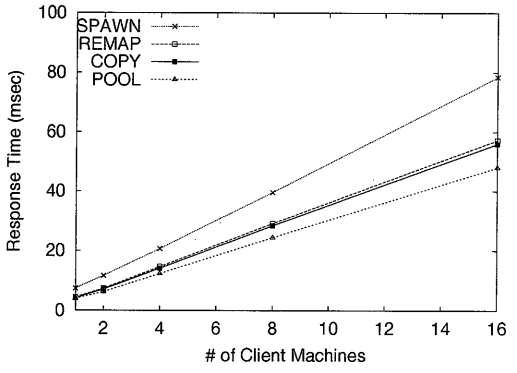

Figure 9: The average response time (CGI program was requested).

5 Related Work

The process cleaning can be regarded as a variation of the technique known as checkpointing/recovery. Several researchers have proposed to use copy-on-write for efficiently implementing checkpointing/recovery [12, 6]. Our contribution is to apply that technique to the security domain instead of traditional domains such as database transactions, process migration, and fault tolerance. In fact, the design of the process cleaning is highly customized for access control mechanisms. For example, only updated memory pages are saved since preserving all the memory pages is not necessary. The saved memory pages are never written on a disk drive since the process cleaning is not for fault tolerance.

Takahashi et al. [17] and Chiueh et al. [7] proposed to divide a process into multiple protection domains. Their idea is to impose a different set of access restrictions on each domain. The process can switch a protection domain for changing a set of access restrictions. Their systems work well for protecting a server from untrusted server plug-ins, for example. On the other hand, they are not appropriate for protecting from the buffer overflow attack. If the process is hijacked by the attack, only the current protection domain is compromised.

However, unlike the process cleaning, their systems do not provide a mechanism for recovering the compromised domain so that the server can continue its service.

6 Conclusion

In this paper, we proposed the process cleaning, which prevents hijacked servers from illegally removing access restrictions. We also presented two implementation techniques for the process cleaning: the remap strategy and the copy strategy. According to our experiments, overheads due to the process cleaning were 50% at the worst case. However, the process cleaning enables Internet servers to use pooled processes for handling a request even if they must impose access restrictions depending on each request. Previous servers must spawn a child process for every request. A web server using pooled processes with the process cleaning achieved approximately 40-50% performance improvement against a web server spawning a child process.

Our current implementation of the process cleaning does not support multithreading. There are several problems for supporting it. For example, should we save/restore a state per thread or per process? If per thread, how should we deal with a memory image shared among multiple threads? Which thread should be allowed to save/restore a state? and so on.

Although the `restore_state` system call restores the whole state of a process, some Internet servers may want to leave part of the state as it is. For example, they may want to preserve some memory pages for carrying data. One of our future research directions is to develop a technique for enabling that without any security risks.

References

[1] Apache HTTP Server Project. `http://www.apache.org/`.

[2] AusCERT. Vulnerability in NCSA/Apache CGI example code. AusCERT Advisory AA-96.01.

[3] A. Baratloo, T. Tsai, and N. Singh. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[4] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson. TENEX, a Paged Time Sharing System for the PDP-10. *Communication of ACM*, 15(3):1135–1143, Mar. 1972.

[5] CERT. Sendmail Daemon Mode Vulnerability. CERT Advisory CA-96.24.

[6] D. R. Cheriton and K. J. Duda. Logged Virtual Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 26–39, Dec. 1995.

[7] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Exten-

sions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153, Dec. 1999.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, Jan. 1998.

[9] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference and Expo*, Jan. 2000.

[10] FastCGI. `http://www.fastcgi.com/`.

[11] J. Hu, S. Mungee, and D. Schmidt. Principles for Developing and Measuring High-Performance Web Servers over ATM. Technical Report 97-09, Department of Computer Science, Washington University, 1997.

[12] K. Li, J. F. Naughton, and J. S. Plank. Real-Time Concurrent Checkpoint for Parallel Programs. In *Proceedings of the 2th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Mar. 1990.

[13] Mindcraft. WebStone Benchmark. `http://www.mindcraft.com/webstone/`.

[14] Openwall Project. Non-Executable User Stack. `http://www.openwall.com/linux/`.

[15] R. Rashid, A. Tevanian, M. Young, D. Golub, and R. Baron. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Oct. 1987.

[16] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[17] M. Takahashi, K. Kono, and T. Masuda. Efficient Kernel Support of Fine-grained Protection Domains for Mobile Code. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 64–73, June 1999.

[18] NetBSD Security Alert Team. at(1) vulnerabilities. NetBSD Security Advisory NetBSD-SA1998-004.

[19] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2000.

[20] WWW Homepage Access Counter and Clock. `http://www.muquit.com/muquit/software/Count/Count.html`.