A Secure Access Control Mechanism against Internet Crackers

Kenichi Kourai
University of Tokyo
7–3–1 Bunkyo-ku, Tokyo
Japan 113–0033
kourai@is.s.u-tokyo.ac.jp

Shigeru Chiba
University of Tsukuba
1–1–1 Tennodai, Tsukuba, Ibaraki
Japan 305–8573
chiba@acm.org

Abstract

Internet servers are always in danger of being "hijacked" by various attacks like the buffer overflow attack. We propose the process cleaning technique for making an access control mechanism secure against hijacking. To minimize damages in cases where the full control of the servers is stolen, access restrictions must be imposed on the servers. However, designing a secure access control mechanism is not easy because that mechanism itself can be a security hole. Process cleaning prevents malicious code injected by a cracker from illegally removing access restrictions from a hijacked server. In this paper, we describe the access control mechanism of our Compacto operating system using process cleaning. According to the results of our experiments, 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 [4], which injects malicious code into a server and takes the full control of the server, that is, "hijacks" it. Once a server is hijacked, the cracker can use the server for performing malicious operations. To protect the servers from these attacks with negligible costs, several techniques for detecting the buffer overflow attack have been developed, but those techniques cannot detect all types of the buffer overflow attack. Thus, imposing access restrictions on a server is still necessary since access restrictions minimize damages by the attack in cases where the server is hijacked.

However, it is not easy to design an 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 might be activated later for hijacking the server.

In this paper, we present a secure 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. To prevent hijacked servers from illegally removing access restrictions, we propose a new technique called *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 malicious code terminates. Then it restores the state of the process and thereby eliminates malicious code from the memory.

We also describe the implementation of process cleaning. Performance overheads of process cleaning is mainly due to restoring a memory image. 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 show the results of this experiment and discuss the overheads of process cleaning.

The rest of this paper is organized as follows. Section 2 describes security risks caused by removing access restrictions. Section 3 presents process cleaning and describes details of the implementation. Section 4 shows the results of our experiments. Section 5 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 server is prohibited from issuing the `write` system call on that file, Compacto can deny that modification. Compacto also provides the `setuid` system call originating from UNIX for giving a server process a lower access privilege.

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 practice. Suppose that a web server is serving both the Internet and Intranet users. The administrators would want to impose strict access restrictions on the server while it is serving the Internet users. On the contrary, they would want to impose loose ones while it is serving the Intranet users. For example, while the server is handling a request from an Intranet user, it should be able to read a file that only the user can read.

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. It temporarily lowers the privileges of a privileged server and later gives the original privileges back. Since it allows that operation 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 difficult. 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 after the access restrictions are removed and thereby execute an arbitrary command indicated by that variable.

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 on itself depending on a request. If the server is connected from a client, then the server spawns a child process, imposes additional restrictions on that child process, and makes 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.

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 use the process pool technique; 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 either efficiency or security.

3. Process Cleaning

We propose a new technique called *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. The imposed access restrictions minimize damages in cases where the server is hijacked and after that they are removed without security risks.

3.1. Recovering a Hijacked Process

For securely removing access restrictions from a process, 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 typical web server running on Compacto uses the `save_state` and `restore_state` system calls as in Figure 1. (1) After initialization, the web server issues the `save_state` system call. (2) Then the server waits until a client connects to it. (3) If a client connects, access restrictions depending on that client are imposed on

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

Figure 1. Server code using process cleaning.

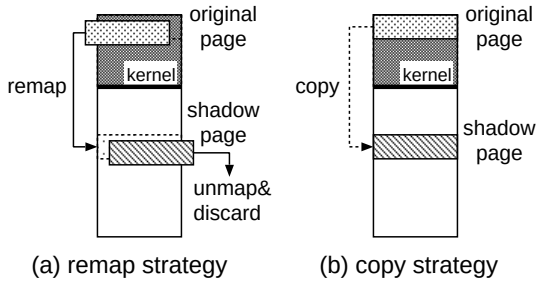

Figure 2. Two strategies to restore memory.

the server. The server handles a request from that client under those access restrictions. (4) After the server finishes handling the request, it issues the `restore_state` system call. This system call recovers the state of the server. Finally, the thread of control is moved back to the next statement of `save_state`. The server repeatedly handles another request from a client.

3.2. Restored Process State

The `restore_state` system call restores the values of all the registers, including an instruction pointer. If injected malicious code successfully takes the control of the process, the execution of that malicious code is terminated when this system call is issued. Thus the malicious code cannot remove access restrictions without losing the control of the process.

`Restore_state` also restores the memory image of a process. This eliminates the Trojan horse, which is malicious code left on memory and later activated for hijacking. Restoring the whole memory image is necessary since distinguishing malicious memory accesses from regular accesses is difficult. Restoring a memory image also keeps a server stable. If an attack causes a memory fault, the server can recover the memory image and avoid termination.

`Restore_state` also restores signal 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. Also, this system call restores the state of opened files and sockets. If malicious code closes a file or a socket, this system call opens it again and restores the file descriptor for it. To avoid the exhaustion of file descriptors, this system call closes files and sockets that have been opened between `save_state` and `restore_state`.

3.3. Restoring Memory

Performance penalties of process cleaning are mainly due to copying memory for saving and restoring the state of a process. To reduce the amount of saved memory, Compacto uses the *copy-on-write* technique. The `save_state` system call does not immediately duplicate the whole memory image. It only changes the state of every writable memory page into the write-protected mode. The memory page is duplicated only if the process first attempts to write in the page and hence a page fault occurs. The original page is moved into the kernel address space and a new page allocated for the duplication is mapped at the original address. We call this new page a *shadow* page. Since the shadow page is writable, no page fault 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 (Fig. 2 (a)). The original page is write-protected to detect the next write. We call this the *remap* strategy. The second strategy is to copy the contents of the original page into the shadow page and leave the original page in the kernel address space (Fig. 2 (b)). The dirty bit of the shadow page is cleared to detect the next write. We call this the *copy* strategy.

The remap strategy does not need to copy memory for the restoration and is also good with respect to memory consumption. However, the users can request Compacto to use the copy strategy. As shown in Figure 1, a typical server running on Compacto repeatedly restores the same state saved by the `save_state` system call at the beginning. In this case, the copy strategy may be more efficient than the remap strategy. The copy strategy does not have to make the restored page write-protected or to catch a page fault for a shadow page. If the process writes in the same set of memory pages for every request, the copy strategy is faster than the remap strategy. On the other hand, if the process writes in a totally different set of pages, the copy strategy is slower because maintaining shadow pages is meaningless. For details of our implementation, see a different article [2].

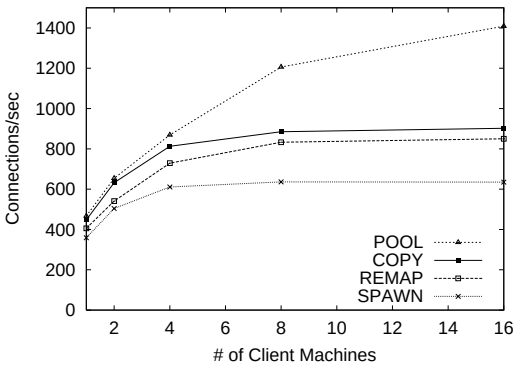

Figure 3. The server throughput.

4. Experiments

We measured the execution performance of the Apache web server performing process cleaning by the WebStone benchmark program. Apache is implemented with the process pool technique and is running on the Compacto operating system based on the Linux 2.2.16 kernel. The server machine is a PC with a Pentium III 933MHz processor. Client machines are PCs with a Celeron 300MHz processor and the operating system is FreeBSD 3.4. The clients and the server are connected through the 100baseT Ethernet. The details of our experiments are described in [2].

For comparison, we used four different types of Apache server. The POOL server is an Apache server which does not perform process cleaning. The COPY server is an Apache server performing process cleaning with the copy strategy. The REMAP server is an Apache server performing 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 neither uses the process pool technique nor performs process cleaning.

Figure 3 shows the server throughput (the number of acceptable requests per second) in the case that various requests were done. The average size of requested data was 7.6KB. 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.

According to the results of our experiments, the COPY server is 1.4 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, process cleaning achieves great performance improvement on those traditional secure servers. It allows the secure servers to handle a request with pooled processes although its performance penalties are not negligible if compared with the POOL server.

In our experiments, the COPY server was always faster than the REMAP server. The performance improvement was 8% on average. 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.

5. Conclusion

In this paper, we proposed process cleaning, which prevents hijacked servers from illegally removing access restrictions. 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 that technique. Our contribution is to apply that technique to the security domain instead of traditional domains such as fault tolerance.

According to our experiments, overheads due to process cleaning were 40% at the worst case. However, 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 that perform process cleaning achieved approximately 40% performance improvement against a web server spawning a child process.

We use a process as a protection domain but Takahashi et al. [3] and Chiueh et al. [1] 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. One of our future research directions is to apply process cleaning to such a fine-grained protection domain.

References

[1] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153, Dec. 1999.

[2] K. Kourai and S. Chiba. A Secure Access Control Mechanism against Internet Crackers. Technical Report ISE-TR-01-176, Institute of Information Sciences and Electronics, Univ. of Tsukuba, 2001.

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

[4] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. of the Network and Distributed Systems Security Symposium*, pages 3–17, Feb. 2000.