Design and Implementation of a Self-Repairing Reference Monitor
自己修復型リファレンスモニタの設計と実装

by

Toshihiro Yoshino
吉野 寿宏

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science, the University of Tokyo
on February 10, 2004
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

A reference monitor is a system which monitors actions of the target process and checks if they are safe (i.e., not malicious). It can be used to ensure security when we run suspicious code or when we expose a network service with potentially unsafe software. Unfortunately, a reference monitor itself may contain some bugs that cause security problems since it is also a program. In this thesis, we propose a reference monitor system with self-repairing feature. The distinguishing features of the system are: (1) that each elementary mechanism of a reference monitor is implemented with an independent module and (2) that the system introduces a monitor process that watches and controls the modules. Since the implementation of our system is modularized, it is uncommon that the whole system is fell into malfunctioning as compared to traditional systems. If one module crashes due to bugs or attacks, the monitor process automatically reboots the module and the whole system comes back to the normal. We first discuss the design of the system and the implementation of the prototype. Then we compare our system with traditional reference monitors in several respects including security and overhead added to target programs.

論文要旨

リファレンスモニタは、ターゲットプロセスの動作を監視し、セキュリティの観点でそれが安全かどうかをチェックするシステムである。疑わしいコードを実行する必要があるときや、安全でない可能性のあるソフトウェアによりネットワークサービスを公開する場合などに有用である。しかし、リファレンスモニタもプログラムである以上、セキュリティ上の問題につながるようなバグを含む可能性がある。本論文では、自己修復の機能を持ったリファレンスモニタシステムを提案する。このシステムの特徴は、リファレンスモニタの各要素機能が独立したモジュールによって実装されている点と、それらが正常に動作しているかどうかを監視するモニタプロセスを導入している点である。我々のシステムは実装がモジュール化されているため、リファレンスモニタ全体が機能停止に陥ることが従来のシステムに比べて少ない。

もし外部からの攻撃やバグなどの理由により、あるモジュールがクラッシュした場合には、モニタプロセスがそのモジュールを再起動することにより、システム全体を正常状態に戻すことができる。本論文ではまずシステムの設計に関して論じ、実際に作成したプロトタイプシステムについて述べる。その後、安全性やオーバーヘッドなどの点で、従来のリファレンスモニタと我々のシステムを比較する。

Acknowledgements

First of all I would like to acknowledge the thesis supervisor Professor Akinori Yonezawa.

Mr. Yoshihiro Oyama gave me considerable help. Without his contribution, this thesis could not have been completed.

Also I want to say thank you to my friends, who were willing to argue technical topics with me. I am deeply grateful for it.

I appreciate all other supports by members of Yonezawa laboratory, too.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Every software may contain some bugs." — This principle has not changed since computers were invented. Bugs are not the problem only of public domain software, but also of commercial software and OSes. They may lead to unexpected crash of applications, system stall, and sometimes even serious security problems.

As we have had more and more opportunities today to execute a number of large-scaled and complicated software distributed over the Internet, we are confronting a great threat caused by bugs. On the other hand we have experientially perceived that the larger software grow, the more difficult it gets to keep them bug-free. Chou et al.[3] showed that the rate at which the number of bugs per thousand lines of code can be indeed reduced by some support, but this rate is still far outpaced by the rate at which software size increases. Therefore, as we add many new functions to the system, the number of bugs increases as a result.

Several solutions to this problem have already been proposed by many researchers[10]. One of them is to confine untrusted programs by using reference monitors, such as Janus[4] or SoftwarePot[6, 11]. Reference monitor is a system which monitors actions of the target program through systemcall hook or some other techniques, and checks them against the security policy specified by the user. When it detects a malicious action, it forces the target to terminate or to skip the action and avoids security problem.

Unfortunately, reference monitors may also contain some bugs just like other programs. At least their possibility of having bugs seems to be as high as other programs, so long as they are implemented using C or C++ language. If bugs in reference monitors are exploited, there is a certain possibility that their security checks become bypassable and they may lose their meaning. In spite of this fact, most existing reference monitors, to our knowledge, still seem to be designed postulating that they are flawless and dependable.

So we aim to make reference monitors more solid and safe. In this thesis we propose a new design of reference monitor and introduce our prototype implementation called ”SeRene system”. This is a highly-modularized reference monitor system which admits that it has some bugs. Instead, it has self-repairing feature to cope with them.

Chapter 2

Design and implementation of SeRene reference monitor

2.1 Basic idea

Figure 2.1 illustrates the popular system structure of traditional (i.e., existing) reference monitors.

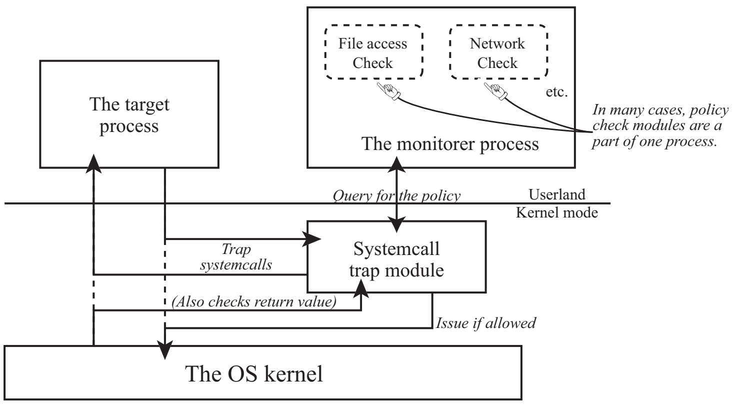

Figure 2.1: Overview of traditional reference monitors

In many cases, a reference monitor consists of two modules; one is systemcall hook module and the other is main module that manages the security policy and decides the action for each systemcalls. The most important problem of systems like this, we think, is that even a small bug in any part of code can cause the whole system to crash.

Then how can we make such systems bug resistant?

Candea et al.[1, 2] suggest ”crash-only design” for Internet systems, based on the idea of Recovery-Oriented Computing[7]. In this approach we do not try to reduce bugs, but we cope with them by microreboots, in other words fine grain component-level restarts. Crash-only software consider crashes, hangs or all other system failures to be facts, and recover from the failure by microreboots.

2.2 Overview of SeRene system

The overview of the structure of our reference monitor SeRene is illustrated in the figure 2.2. It looks similar to that of traditional system mentioned before in total, except that the userland part, which corresponds to the monitorer process in traditional systems, is divided into some modules.

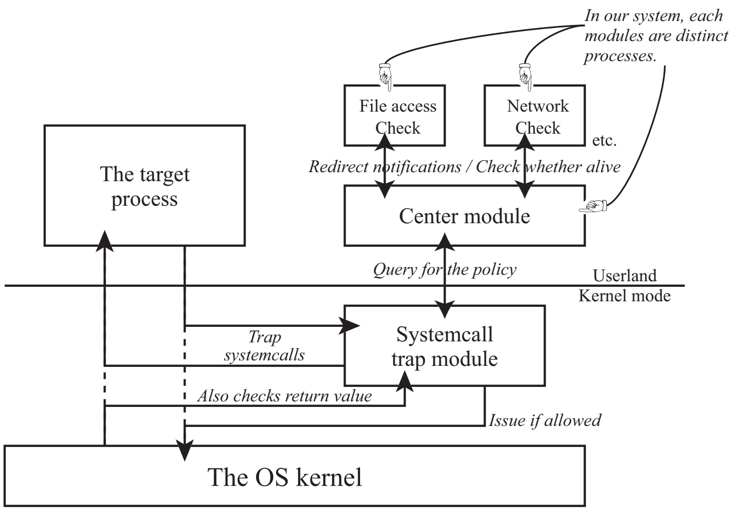

Figure 2.2: Overview of SeRene system

Below is an example of the list of all modules in the system.

- Systemcall hook module
- Center module
- Submodules
  - File access policy module[1]
  - Network access policy module

[1]In the current prototype, file access policy module is not included for some reasons. We discuss this problem later.

  – Generic systemcall policy module

The system is named "SeRene" after its self-repairing feature. It is designed so that the whole system will not be fell into malfunctioning by crashes of submodules. Since each modules are distinct processes on the platform, a crash of one module does not affect others. And if the center module detects a module crash or stop, it tries to restart the module.

Modularized structure also increases the extendability. When any existing submodules are insufficient for the user's purpose, he or she can create and add a new module without modifying code of the whole system. The task he or she has to do for adding a new module is only implementing a new module and writing its declaration to the policy file. Recompilation of the whole system is not needed. Then the center module reads the policy file and runs all the submodules listed there.

In the rest of this chapter we discuss detailed designs of each modules and the prototype system implemented on FreeBSD 5.2-CURRENT platform running on IA-32 architecture.

2.3 Design of the systemcall hook module

2.3.1 Basic design

SeRene watches systemcall invocations of the target process by systemcall hook module. For each systemcall that the target issues, this module catches it and ask the upper layer (the monitorer process running on userland) what to do. The upper layer can allow or deny (return error `EPERM` to the target) systemcall invocation, or it can also kill the target process for serious conditions.

On several platforms such as Linux and FreeBSD, Loadable Kernel Module (LKM) is suitable for this purpose. Since LKM runs under the supervisor privilege, it can easily replace the systemcall entry table, which allows us to add the hook to systemcalls without any special patches to the kernel.

The upper layer receives the notification about these three events happening on the target processes.

- Entry to systemcall (sent before the real systemcall routine)
- Exit from systemcall (sent after the real routine)

- Process termination (sent when the target process invokes `exit()` systemcall or it is killed by a signal[2])

Most of the policy checks are done on systemcall enter events, because inspection of the systemcall arguments is required. Systemcall exit events can be used when the return value from the systemcall is needed for the check. These events stop the target until the upper layer responds to them, thus the target is not allowed to invoke any unchecked systemcall.

But unlike these two types of notification, process termination notifications do not block, since they should not be restricted any more.

2.3.2 Communication specification

On installation, this module installs one new systemcall which creates a monitor context[3]. Monitor context is a file descriptor (often abbreviated as "FD") through which actions of target processes are notified. Attaching or detaching a process to the monitor context can also be done using `ioctl()` systemcall on this FD.

The upper layer communicates with the systemcall hook module by `read()` and `write()` systemcalls on this FD. When a target process attached with the monitor context terminates or invokes a systemcall, a notification packet (structure is illustrated on figure 2.3) becomes available on the FD. The monitorer reads it, inspects it and finally writes the response, which action has to be taken, to the FD. Since each monitor context has only one queue attached, requests are processed in First-In-First-Out (FIFO) order and monitor works well when there are multiple target processes. If no more processes are associated with the context, `read()` and `write()` systemcall on the FD returns zero immediately, indicating end of monitor.

The systemcall hook module restricts the access to this FD in order to ensure security. When requesting read or write, the caller must be the owner (the creator) of the monitor context.

If the FD is closed by the owner before all target processes terminates, the systemcall hook module forces all remaining targets to terminate by sending `SIGKILL` signal to them.

[2]Currently, this notification is limited only to the case in which the target is killed by the reference monitor for its security policy.

[3]Only one monitor context at the same time is allowed so far, for the ease of implementation.

2.4 Implementation of the systemcall hook module

In this section we introduce more detailed information about the systemcall hook module, especially platform dependent techniques used to implement the prototype.

2.4.1 Systemcall hook and monitored processes

The systemcall hook module traps systemcalls by replacing the systemcall entry table where the address of all systemcalls are stored. Since all processes are affected by this technique, we have to distinguish whether or not the caller process is a target. Although we can use a list of target processes the systemcall hook module manages, this check for each systemcalls surely slows down all processes running on the machine. We solved this issue by adopting a new flag on `p_flag` field in `proc` structure, which is a bit-vector of status information about the process. We assure that every target processes have `P_MONITORED` bit set, then unmonitored processes can be quickly determined.

We must consider `fork()` systemcall, which duplicates the caller process, because a new monitor target is generated. If one of the target processes invokes `fork()` systemcall, the systemcall hook module has to automatically add the new copied process to the target list of the monitor context which the caller is in. In FreeBSD kernel, `at_fork` handler is applicable for this operation. Whenever a process forks, the kernel notifies all registered `at_fork` handlers of process information. Since the current system allows only one monitor context at the same time, if a process is marked as monitored, it belongs to this context. Thus what the handler does is simply marking a new process as monitored and adding it to the target list.

2.4.2 Systemcall notification details

For `RT_SYSCALL_ENTER` notifications, the systemcall hook module copies the systemcall arguments in `data.args[]` array.

In most cases, each of them corresponds the first four arguments. But this is not the case with some special systemcalls that takes an argument that is longer than the length of registers on the platform. Because such argument does not fit into one register, we have to put some elements together to process these notifications.

There are three stores for return values in `RT_SYSCALL_EXIT` notifications. This is because the systemcall handler can return three values: error code, first and second return values.

On FreeBSD platform, `data.retval[2]` is an error code of the systemcall, which is a return value from the systemcall. If this element is not equal to zero, it indicates that an error occurred. Then the error code is stored into `EAX` register, and returned to the user process.

Other two elements, `data.retval[0]` and `data.retval[1]`, are also return values of the systemcall. They are a copy of `td_retval[]` field of `thread` structure (it is passed as an first argument for the systemcall handler) which indicates the invoker of the systemcall. Systemcalls such as `open()` or `pipe()` need to return file descriptor number, the field is used for this purpose. On success (user process can determine whether the systemcall succeeded or failed by checking the carry flag), instead of error code, these values are returned using `EAX` and `EDX` registers. Since kernel trap routine saves the value of these registers into `td_retval[]` field before calling the systemcall handler, the value of these register remain unchanged unless the systemcall handler overwrites them or an error occurs.

2.5 Design of submodules

SeRene must be used with one or more submodules, each of which has its policy domain (e.g., file access, network access, and so on) and checks the target process' action according to its security policy. There are several conventions for submodules.

- Every submodules MUST understand the policy file whose filename is specified in the command line argument. Its format is introduced later in section 2.7.
- Every submodules MUST communicate with the center module using a file descriptor whose number is specified in the command line argument.
- Every submodules MUST understand the request packet and MUST be able to talk the response packet in the figure 2.3.

- Every submodules MUST send some message to the center module at least one for a few seconds.
- Every submodules MUST be restartable at any time.

A submodule can be implemented using arbitrary programming language, as long as it complies all of these conventions. In addition to the affixed submodules, of course third-party submodules can also be used.

The module first reads the policy file and then enter the main loop. It waits notifications and process them if received, until the supplied file descriptor closes. Format of the packets which they read and they should write is discussed before.

The last two conventions are for supporting the self-repairing feature. They have to send heartbeat messages (it must be one of invalid response) to the center module periodically, unless they have received policy query. We can use `select()` systemcall for this loop. To meet these conventions, they must either be stateless or have their states rather on persistent storage, such as hard disk, than on memory.

We designed a library that helps this common processes. It implements the main loop, policy file parser and some useful subroutines. A programmer has only to write a few functions that are called from inside the main loop. The routine included in the library automatically reads the policy file, negotiates with the center module and sends heartbeat messages (one per second, in the prototype implementation).

Reading the target's memory is often required for the check. The library includes helper functions for this purpose. This can also be said as an encapsulation of platform-dependent code, since such functions may also be platform-dependent. Currently it uses process file system (a.k.a. procfs) for memory access, and so it cannot check processes that have setuid bit set.

Other kernel support will be needed for some checks. One good example is file access control. When specifying file or directory, we can use either absolute path or relative path. Whereas absolute paths can be checked easily, relative paths have to be translated into absolute paths before the check. To translate them, we have to know which directory the target process is currently working on. We could trace `chdir()` systemcall or `fchdir()` systemcall, but if a module is rebooted before it stores the new working directory as its persistent data, it loses the consistency and the check will not work well. Moreover, to support `fchdir()` systemcall, we have to maintain file descriptor table of all target processes and also have to trace more extra systemcalls, such as `fork()`, `execve()` and `fcntl()` (to detect use of `FD_CLOEXEC`). Of course, doing like this will surely slow down the whole system and we should take the more simple solution.

Due to this difficulty, a file access policy module in the prototype system cannot check working directory and does not work properly.

2.6 Design of the center module

The center module is the most important part of our system. It first creates systemcall monitor context, runs all submodules listed in the policy file and then executes the target program.

In our system, the center module itself does not have any security policy.

It just dispatches notifications from the systemcall hook module to its all submodules and puts together the responses from them. There are several strategies in putting together the responses, we are now taking the safest one: A systemcall is allowed only when all submodules permit it.

Alternatively it is responsible to administrate submodules. When some submodules seems failed, it has to reboot them to get the whole system back to work properly.

During the normal time, every submodules send heartbeat messages to the center module in their idle time. If the center module does not receive any message from a module for a definite period of time (3 seconds in the prototype implementation), it assumes that the module is dead and needs to be rebooted. Rebooting a module is done by sending `SIGKILL` signal and then executing the module again. The same action happens when it detects termination of submodule, which can be notified as `SIGCHLD` signal. After the reboot, the center module sends unanswered notifications again. But if reboot does not repair the system (in other words, the module still dies continuously in spite of reboots), the center module gives up and finishes execution, stopping the target.

This policy may sound a bit unsuitable for our goal. However we cannot simply desert a submodule, because security checks for the protection domain of the module become bypassable if we take this policy.

So the policy we currently choose here is one that is considered to be the safest strategy. Further research will be needed about this point.

In this design, the center module is still a single point of failure. But it will be smaller and have simpler functions than complex submodules, the probability of crash will also be lower than them. Or, we may be able to prove that it is bug-proof by using some proving support tools.

2.7 Format of policy file

Figure 2.4 is an example policy file for SeRene system.

Policy file is comprised of some sections, each of which begins with the section name followed by a colon.

In `modules` section, modules to use are specified. This section must exist in any policy file. Otherwise, the system does not start.

Rest of the policy file is for submodules. How they are processed and matched against the target's actions depend on each submodules. Our current submodules takes the action specified in the policy definition that lastly matched to each notifications. For example, the `net` section, which is for *net* submodule, means:

- Allow `connect()` systemcalls that connects the socket to `127.0.0.1`.
- Allow `socket()` systemcalls that create TCP, UDP or UNIX socket.
- Deny all `socket()` and `connect()` systemcalls that do not meet the two conditions above.

```
typedef enum reqtype_t {
  RT_SYSCALL_ENTER,  /* enter a syscall */
  RT_SYSCALL_EXIT,   /* exit from a syscall */
  RT_PROCESS_SIGNAL, /* caught a signal */
  RT_PROCESS_EXIT,   /* process exited */
} reqtype_t;
typedef enum reqexittype_t {
  RET_EXITED,
  RET_KILLED,
} reqexittype_t;

/* Request packet */
typedef struct request_t {
  pid_t          pid;
  reqtype_t      type;
  unsigned int   request; /* Syscall/Exit type */
  union {
    unsigned int args[4];   /* Syscall args */
    unsigned int retval[3]; /* Syscall retval */
  } data;
} request_t;

/* Response packet */
typedef enum response_t {
  RESP_ALLOW,
  RESP_DENY,
  RESP_KILL,
} response_t;
#define RESP_MAX         RESP_KILL
#define RESP_HEARTBEAT   (RESP_MAX + 1)
```

Figure 2.3: Notification packet structure

```
monitor:
module syscall
module net

syscall:
deny    fork
allow   getpid

net:
deny    all
allow   protocol        tcp,udp,unix
allow   connect         127.0.0.1
```

Figure 2.4: Example policy file

Chapter 3

Experimental results

We implemented a prototype system on FreeBSD 5.2-CURRENT running on PentiumM 1.3GHz.

For precice comparison between our system and the traditional system, we also derived the traditional-design system which does not have userland module divided from the self-repairing system. By comparing their results, we can measure the difference which is purely caused by the design.

3.1 Test of systemcall monitoring and policy checking

First we show how security checks hook work. In this example, we are using the policy file in the figure 2.4.

We try to connect `127.0.0.1`(authorized host) and `192.168.1.101`(unauthorized host) using `telnet`.

```
% ./serene test.plc telnet 127.0.0.1
SeRene System  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

DEBUG: Executing command: telnet 127.0.0.1
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Trying SRA secure login:
User (tossy-2): ^C%

% ./serene test.plc telnet 192.168.1.101
SeRene System  ver 0.01
```

```
          Copyright(C) 2003 Toshihiro Yoshino

DEBUG: Executing command: telnet 192.168.1.101
Trying 192.168.1.101...
telnet: connect to address 192.168.1.101: Operation not permit
ted
telnet: Unable to connect to remote host
```

As mentioned before, the policy file means these three conditions:

- Allow `connect()` systemcalls that connects the socket to `127.0.0.1`.
- Allow `socket()` systemcalls that create TCP, UDP or UNIX socket.
- Deny all `socket()` and `connect()` systemcalls that do not meet the two conditions above.

Thus `telnet` succeeded as usual when we tried to connect to `127.0.0.1`. On the other hand, we could not connect to `192.168.1.101`, because connection to this host does not match with the first two conditions and the third action is applied.

By specifying debug option `-d 3` on the command line, SeRene displays some information for each events and we can look them through.

```
% ./serene -d 3 test.plc telnet 192.168.1.101
SeRene System  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

Boot submodule 'syscall'
Module syscall booting...
Boot submodule 'net'
Module net booting...
pid9350 ENTER(close) arg1=0x3
pid9350 EXIT(close) retval=0x0
pid9350 ENTER(write) arg1=0x2
DEBUG: Executing command:pid9350 EXIT(write) retval=0x0
...
pid9350 ENTER(write) arg1=0x1
Trying 192.168.1.101...
pid9350 EXIT(write) retval=0x0
```

```
pid9350 ENTER(socket) arg1=0x2
pid9350 EXIT(socket) retval=0x0
pid9350 ENTER(getuid) arg1=0x0
pid9350 EXIT(getuid) retval=0x0
pid9350 ENTER(setuid) arg1=0x3e9
pid9350 EXIT(setuid) retval=0x0
pid9350 ENTER(setsockopt) arg1=0x3
pid9350 EXIT(setsockopt) retval=0x0
pid9350 ENTER(connect) arg1=0x3
pid9350 ENTER(write) arg1=0x2
telnet: pid9350 EXIT(write) retval=0x0
pid9350 ENTER(write) arg1=0x2
connect to address 192.168.1.101pid9350 EXIT(write) retval=0x0
pid9350 ENTER(write) arg1=0x2
: pid9350 EXIT(write) retval=0x0
pid9350 ENTER(write) arg1=0x2
Operation not permitted
pid9350 EXIT(write) retval=0x0
...
pid9350 ENTER(exit) arg1=0x1
pid9350 Process exitting
DEBUG: All target process exited.
```

In this case, the network policy module denies `connect()` systemcall, it fails with `EPERM`. Notice that, since the control does not enter the real `connect()` systemcall, exit notification for `connect()` systemcall is lacking.

3.2 Secutiry

Next we intentionally implemented a buggy module *buggy*, which crashes after it processed 100 notifications, in order to demonstrate whether self-repairing feature works. Source code of *buggy* module is presented at appendix A. With this module, we inspected how our system recovers from crash.

We used *bug.plc* (see figure 3.1 for its contents) for this test.

First we tried to monitor a program with the traditional system. Here we used `-d 2` debug option, which tells the system that messages about submodules should be output.

```
monitor:
module  syscall
module  buggy
```

Figure 3.1: *bug.plc*: policy file used for the test of self-repairing feature

```
% ./trad -d 2 bug.plc ../test/getpid 1000
Traditional Reference monitor  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

Boot submodule 'syscall'
Boot submodule 'buggy'
DEBUG: Executing command: ../test/getpid 1000
Oh, no!
Segmentation fault (core dumped)
Process killed according to the security policy.
```

As easily expected, the traditional reference monitor crashed due to a module's bug. Then the target process is killed because the monitor context is closed on termination.

Next we tried the same test with our system.

```
% ./serene -d 2 bug.plc ../test/getpid 1000
SeRene System  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

Boot submodule 'syscall'
Module syscall booting...
Boot submodule 'buggy'
Module buggy booting...
DEBUG: Executing command: ../test/getpid 1000
Oh, no!
Module buggy died
Module buggy rebooting...
Module buggy booting...
Oh, no!
Module buggy died
Module buggy rebooting...
Module buggy booting...
```

```
Oh, no!
...
Module buggy rebooting...
Module buggy booting...
pid10666 Process exitting
DEBUG: All target process exited.
```

Debug message indicates that the *buggy* module indeed crashed. But since each modules in our system are distinct processes, this crash did not strike any other modules. The center module then properly detected the module's death and started rebooting it. After a reboot, the *buggy* module begins to process notifications again as if nothing had happened to it.

The *buggy* module crashes again after processing the next 100 messages, so this procedure has to be repeated until the target program exits.

3.3 Overhead to programs

Then we measured the overhead added to target programs. Programs used for this test and output scripts are present at appendix B. The result is shown in table 3.1.

Program	No monitor	Traditional RM	Our system
getpid			
(100,000 times)	0.05	1.43	9.86 (6.90)
(500,000 times)	0.29	7.04	49.47 (7.03)
socket			
(500 times)	0.08	0.15	0.34 (2.27)
(5,000 times)	—-	1.56	3.74 (2.40)

Note: Parenthesized numbers are the ratio of execution time, between our system and traditional system.

Table 3.1: Execution time with/without a reference monitor system (in seconds)

The *getpid* program simply calls `getpid()` systemcall repeatedly. Since this is one of the most lightweight systemcall, we can say that this is the worst case in which overhead is very high; Approximately it took 25 $\sim$ 30 times longer than usual in traditional system. In comparison between traditional system and self-repairing system, overhead in the latter system is about 7 times larger than the former.

However the overhead of inter-process communication (IPC) can be cancelled by some other factor. For example, if systemcall takes long time before completing its operation, the difference of execution time between two systems becomes relatively lower. The *socket* program creates a socket and tries to connect it to the *discard* port (port number 9) on `127.0.0.1` (in other words, `localhost`) by using `socket()` and `connect()` systemcall many times. We measured overhead again, and this time our system took only 2.3 times longer than traditional system.

From these results we can conclude that the overhead for real programs depends on which systemcalls they use and how frequent they call systemcalls. Generally speaking, the overhead will decrease further from the results above, because real programs perform many calculations and userland operations between two subsequent systemcalls which these test programs do not perform.

Overhead is also subject to the number of modules used. Table 3.2 shows how overhead changes if we use more submodules.

Program	Default	5x Default	10x Default
getpid(10,000 times)	1.28 (1.0)	4.82 (3.8)	9.21 (7.2)
socket(250 times)	0.22 (1.0)	0.64 (2.9)	1.23 (5.6)

Note: Parenthesized numbers are the ratio of execution time, in that case and in default case.

Table 3.2: Execution time with our system adding modules more (in seconds)

Execution time written in ”Default” column is that when we use the policy file on the figure 2.4, in which there are only two submodules. ”5x Default” and ”10x Default” are the environment where we use 10 and 20 submodules each. For example in ”5x Default”, we duplicated `module` declarations for 5 times.

The result shows that overhead is almost proportional to number of submodules. This is because the notification of *every* systemcall is sent to *every* submodules in the current implementation.

Of course overhead also depends on the complexity of policy checks. Thus we cannot easily say how much overhead is added to the real programs. But if we dare to conclude something from this result, overhead of a reference monitor increases to approximately twice the original when we add self-repairing feature to it. Due to rapid progress on hardware speed, this overhead would not be too large to apply to the real programs which emphasize security.

Chapter 4

Related works

First of all we have to say that we are deeply inspired by the idea of "recursive recovery" and "crash-only software", introduced by Candea et al.[1, 2].

Although we used only very simple part of this technique, their idea as a whole is much more complex but solid. They suggest multi-level modularized system in which an acyclic graph that expresses the dependency of modules can be constructed, and the recovery module should use this graph for determining which modules to be recovered. They applied their technique to a Java-based software system Mercury, a prototype ground station system for communicating with data satellites.

It is also very common in other fields to divide a system into some independent modules in order to achieve higher security and robustness. qmail[8], a widely used mail transport agent (MTA), is an example of such systems. It consists of some program modules which do not believe each other, and the whole system remains safe even if several modules are hijacked.

In designing reference monitor and interfaces, we referred Janus[4, 9]. This is an orthodox style reference monitor. Currently it is implemented on Linux running on the x86 architecture[5].

Its kernel module interface gave us many hints for designing the systemcall hook module.

Chapter 5

Future work

5.1 Limitations of the current system

As shown by experiment, under the worst case that the target program simply repeats issuing very lightweight systemcalls, current implementation of SeRene system slowed down the program execution time up to about 200 times the normal.

This is mainly because every submodule gets the full notification about systemcalls in the current system, despite that each of them are interested on very small subset of systemcalls. We may be able to reduce the overhead if we change the implementation so that each submodules tells the center module on which systemcalls they are interested.

Although we have mainly aimed to cope with crashes caused by system bugs in this thesis, there is still another possible pattern of exploitation: highjacking. Since the current implementation of SeRene system does not suspect submodules, the system also believes a module regardless it is highjacked or not, as long as it complies the conventions listed in the section 2.5. Thus specially crafted highjacking can allow the attacker to call arbitrary systemcalls.

Unfortunately detection of highjacking is itself a hard problem, which is a very famous problem "Byzantine General's problem" in distributed systems. There is a well-known algorithm for this problem, but it requires very high technical level and high implementation cost. First we must prepare more than one submodules for each protection domains, and they interact with each other to determine if there is a malicious module. These submodules must not be the same, because the same submodule is highjacked in the same manner.

5.2 Future work

Implementing other submodules, especially file access policy module, is the most important task for us. Due to lack of modules, we have to admit the current prototype is still far from practical system. To make the system more useful, it is needed to add functions to the current system.

Moreover, what the current prototype can do for check is only watching the arguments and return values of systemcalls. Substituting such values will be useful. For example, it allows the system to map the file to another, just like *chroot* utility on UNIX systems.

In spite of our efforts, the center module still remains a single point of failure. Proving that it is bug-proof or reducing the probability of crash is also important task for us.

SeRene system is a collection of modules which process their tasks with communicating each other. Thus it is, in these respect, very similar to distributed systems. The prototype is considered to be a very common server-client model, which inevitably includes some single points of failure. Methods to create fault-tolerant distributed systems may be applicable to our system. For example, peer-to-peer(P2P)-like system structure, which does not have any center module, will increase the system's robustness, but at the same time the complexity, too.

Alternatively, we may be able to adopt so-called "higher-order reference monitor", a reference monitor that watches another reference monitor. This method is based on the assumption that a reference monitor accesses only very small resources on the machine. Of course heterogeneous reference monitor set would be feasible, but it requires multiple-stage systemcall hook by some kernel modules at the same time. This might cause the OS to crash, or at least to slow down. Therefore, we could say that the homogeneous reference monitor set would rather be better, in which we can use the same kernel module for all reference monitors.

Arranging strategy is also an open question. Although we said that the current prototype takes the safest strategy in arranging the policy, choosing other strategy, such as majority vote, can increase the system's flexibility, but this may sacrifice security a bit.

Chapter 6

Conclusion

We designed and implemented a prototype system of self-repairing reference monitor. In this way, the possibility that reference monitor crashes caused by bugs can be reduced.

And we have shown that there are some advantages to modularize the system structure.

1. Single point of failure can be reduced.

   If a submodule crashes, the whole system is not affected by it. Then the center module detects it and reboots the dead module.

2. The system becomes flexible and extendable.

   The user can easily extend a system by implementing his/her own submodules. Submodules can be programmed in any programming language, as long as they conforms all conventions.

On the other hand, this modularization also increases overhead. However it is feasible in the case where security is important than speed, such as the Internet services.

Although our result is very preliminary, we feel that it would be a milestone on the road toward the method to secure reference monitors from bugs.

References

[1] George Candea, James Cutler, and Armando Fox. Improving Availability with Recursive Microreboots: A Soft-State System Case Study. *Performance Evaluation Journal*, 56(1–3), 2004.

[2] George Candea and Armando Fox. Crash-Only Software. In *In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, 2003.

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[4] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, July 1996.

[5] Janus. http://www.cs.berkeley.edu/˜daw/janus/releases.html.

[6] Kazuhiko Kato and Yoshihiro Oyama. SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation. In *Software Security – Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 112–132, 2003.

[7] David A. Patterson. Recovery Oriented Computing: A New Research Agenda for a New Century. In *International Symposium on High Performance Computer Architecture*, page 247, 2002.

[8] qmail. http://www.qmail.org/.

[9] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, 12, 1999.

[10] 大山恵弘. ネイティブコードのためのサンドボックスの技術. **コンピュータソフトウェア**, 20(4):55–72, 2003.

[11] 大山恵弘, 神田勝規, and 加藤和彦. 安全なソフトウェア実行システム SoftwarePot の設計と実装. **コンピュータソフトウェア**, 19(6):2–12, 2002.

Appendix A

The *buggy* module sourcecode

```
#include <stdio.h>
#include "common.h"
#include "protocol.h"
#include "libmodule.h"

#define LIMIT 100
static int counter = 0;
static char *null = NULL;

static response_t
reqproc(const request_t *req)
{
  if(++counter >= LIMIT){
    fprintf(stderr, "Oh, no!\n");
    *null = '\0';
    counter = 0;
  }
  return RESP_ALLOW;
}

static BOOL
config_handler(const char *secname, const policy_t *pol)
{ return TRUE; }
static BOOL
mod_init(BOOL is_init, pid_t monitor)
{ return TRUE; }

MODULE(buggy, mod_init, config_handler, reqproc);
```

Appendix B

Overhead test programs and results

B.1 *getpid*: calls getpid() repeatedly

B.1.1 Sourcecode

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  unsigned int num = 1000;
  if(argc > 1) num = atoi(argv[1]);

  while(num-- > 0) getpid();
  return 0;
}
```

B.1.2 Result

```
% time ../test/getpid 100000
0.000u 0.059s 0:00.05 100.0%    6+211k 0+0io 0pf+0w

% time ./trad test.plc ../test/getpid 100000
Traditional Reference monitor  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

DEBUG: Executing command: ../test/getpid 100000
0.107u 0.796s 0:01.43 62.2%     42+285k 0+0io 0pf+0w

% time ./serene test.plc ../test/getpid 100000
SeRene System  ver 0.01
```

```
          Copyright(C) 2003 Toshihiro Yoshino

DEBUG: Executing command: ../test/getpid 100000
1.959u 7.860s 0:09.86 99.4%     31+250k 0+0io 0pf+0w
```

B.2 *socket*: connects to localhost repeatedly

B.2.1 Sourcecode

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

const static char           *host = "127.0.0.1";
const static unsigned short  port = 9;
static struct sockaddr_in    sin;

void check()
{
  struct protoent* pe = getprotobyname("tcp");
  int fd;

  if(!pe){
    fprintf(stderr, "getprotobyname() could not fine tcp proto
col\n");
    exit(255);
  }
  if((fd = socket(PF_INET, SOCK_STREAM, pe->p_proto)) < 0){
    perror("socket()");
    exit(255);
  }
  if(connect(fd, (struct sockaddr *) &sin, sizeof(sin)) < 0)
    perror("connect()");
  close(fd);
}

int main(int argc, char *argv[])
{
  unsigned int i, num = 100;
  if(argc > 1) num = atoi(argv[1]);

  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_len    = sizeof(sin);
  sin.sin_port   = htons(port);
  if(inet_aton(host, &sin.sin_addr) != 1){
    fprintf(stderr, "Error in address conversion\n");
  }else
    for(i = 0; i < num; i++){
      printf("Try%d\n", i);
      check();
    }
  return 0;
}
```

B.2.2 Result

```
% time ../test/socket 250
Try0
Try1
Try2
...
Try247
Try248
Try249
0.000u 0.010s 0:00.04 25.0%     12+456k 0+0io 0pf+0w

% time ./trad test.plc ../test/socket 250
Traditional Reference monitor  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

DEBUG: Executing command: ../test/socket 250
Try0
Try1
Try2
...
Try247
Try248
Try249
0.006u 0.012s 0:00.07 14.2%     192+1248k 0+0io 0pf+0w

% time ./serene test.plc ../test/socket 250
```

```
SeRene System  ver 0.01
          Copyright(C) 2003 Toshihiro Yoshino

DEBUG: Executing command: ../test/socket 250
Try0
Try1
Try2
...
Try247
Try248
Try249
0.018u 0.116s 0:00.17 70.5%     37+310k 0+0io 0pf+0w
```