

Systrace Enforces Rules for Permitted System Calls

Gatekeeper

Vulnerabilities in web servers, browsers, IRC clients or audio players may allow programs to perform all kinds of malevolent tricks. Systrace protects your system from unpleasant consequences by placing it in a tightly locked jail of legitimate system calls.

BY MARIUS AAMODT ERIKSEN
AND NIELS PROVOS

Systrace, the kernel gatekeeper, forces processes to respect a policy for system calls, thus restricting access to a host. Of course this will not remove any existing vulnerabilities, but it will mitigate the consequences. If a program is not required to launch any other processes, the systrace policy will disable the syscall normally used for this purpose. An intruder will be unable to open a shell, even if she has gained complete control over an active process.

To enforce the policy, systrace intercepts system calls at kernel level and launches only those functions intended by the legitimate user. If an application attempts to step outside the bounds set by systrace, a GUI popup warns the user and prompts them to

decide to either permit or deny the action. Systrace comprises a kernel patch, a command line program, and a gtk GUI (BSD license). All three components are available from [1].

Userspace applications use system calls to access the kernel. System calls provide services in areas where security is critical, such as file handling, network connections, or the heap. Table 1 provides an overview of common syscalls. More than 200 calls are available on most UNIX type operating systems, and they provide the only way to invoke persistent changes to a system. Without them a process could not perform any useful tasks, although admittedly an attacker would not be able to get up to any mischief either.

No Attacks without System Calls

A typical attack might succeed due to a buffer overflow in a web server that allows an intruder access to a shell. The malevolent hacker would then inject exploit code that runs with the privileges of the web server process. The code will need to execute a few system calls, such as “fork()” and “execve()” for example, to launch the shell. Thus, the real damage is not caused by the security hole itself, but by a syscall it allows. As security holes are more or less inevitable, admins often monitor system calls in order to provide an extra layer of system protection.

Normally an application will have access to any system calls it requires. Nothing would prevent the web server from launching a shell and serving it up to any user connecting to the web server. This action is undesirable and the server was not programmed to perform it, but a



H&A Ramm free Visuals, visipix.com

software bug allows the attacker to trick the application into behaving in way the authors did not envisage.

Each application needs access to a subset of the syscall interface functionality. A simple web server listens on TCP port 80, responds to HTTP requests, and serves up files from a standard directory structure. The web server is not required to provide any other services, and it particularly does not need to launch an interactive shell or read “/etc/passwd”. The system calls a program that allows you to describe its legitimate functions. Systrace makes use of this fact: it monitors the system calls and develops a policy based on these calls. Any application that is controlled by the systrace program can only work within the bounds of the policy.

Policies

A systrace policy comprises a set of rules. Each rule controls a syscall and its parameters, specifying whether or not the call is allowed. The simple rules outlined in the following example allow for the “fchdir()” and “fstat()” system calls:

```
linux-fchdir: permit
linux-fstat: permit
```

A rule containing the “deny” keyword instead of “permit” would prevent these

Table 1: Common System Calls

Syscall	Function
fork	Creates a new process
execve	Executes a file
open	Opens a file
read	Reads from a file descriptor
write	Writes to a file descriptor
connect	Uses a socket to open a connection to a remote host
bind	Binds a socket to a name
unlink	Deletes a directory entry

system calls. Rules can also apply to a specific parameter of a syscall:

```
linux-fsread: filename eq ⚡
"/tmp/foo" then permit
linux-fsread: filename match ⚡
"/etc/*" then deny[enoent]
```

Based on these rules, the program is allowed to read the file “/tmp/foo”, but files that match the “/etc/*” will lead to an “ENOENT” error. Instead of running the syscall, systrace informs the application that the file does not exist.

Policy Grammar

The policy grammar is identical for all system calls. Each rule begins with the name of an emulation and the syscall, e.g. “linux-fsread”. This is followed by a list of conditions, and an action (“deny” or “permit”) to be taken by systrace. As Linux does not support syscall emulation, each rule starts with the “linux” string. Systrace also supports OpenBSD and NetBSD and both these systems can emulate various syscall variants.

An optional error code can be appended to the action (this defaults to “EPERM”, operation not permitted). The user can optionally choose to have systrace log specific activities by adding the “log” keyword at the end of the rule. The BNF specification (Backus Naur Form) of the policy syntax is shown in Listing 1.

A few predicates are available to restrict the validity of rules. They define additional conditions for the actions and currently apply to users or groups on the system. Predicates are appended to the rule following a comma, for example:

```
linux-fsread: filename eq ⚡
"/etc" then deny[eperm], if ⚡
group != wheel
```

This rule only restricts users who are not members of the “wheel” group.

Arguments are defined for the majority of system calls. For example, “open” expects to be passed the name of the file to be opened. Systrace translates these parameters into a human readable format, displaying them as strings and comparing them with the rules. Systrace offers a range of operators for this comparison (see Table 2).

Implementation: Setting Up a Base Camp

When implementing systrace functionality, you first need to find an appropriate place to insert control mechanisms. Looking at the path of a syscall reveals several potential candidates. Applications initiate system calls by writing to specific registers and invoking soft interrupts (the “int” instruction on i386 processors). The standard C library (libc) is typically responsible for setting up and initiating syscalls. A large proportion of the C library functionality derives from system calls, for example “open()”, “read()” and “write()”. The system call path is shown in Figure 1.

Syscalls can be intercepted and modified in each of these layers. Intercepting system calls in the library layer (libc) would be trivial: You could use the “LD_PRELOAD” environment variable to preload a library on top of libc. The new library would provide all of libc’s system call functionality.

Unfortunately, an attacker would easily be able to sidestep this mechanism by making an application invoke the system call itself, instead of using libc. And the method would not work for statically linked programs. Additionally, there are a few systrace functions that cannot be run in userspace.

Gatekeeper – Syscall Gateway

So it would seem that the kernel layer is the natural place to intercept system calls. This is the only place where you can be sure to catch every syscall, no matter where or how it was initiated. Every system call enters the kernel via the syscall gateway, which acts as an interrupt handler for the soft interrupt used by system calls.

The gateway reads a register (“eax” for i386 processors) to ascertain the system call number, which is a simply index into the system call table containing function pointers to individual kernel functions. The gateway parses the values of the syscall number and then initiates the correct function which performs the task specified by the system call. In order to reject a system call, systrace must intercept it before it is executed. Systrace hooks into the call gateway to do so.

Most of syscall’s functionality is implemented in a userspace program. The kernel hook is provided by device: “/dev/systrace”. The userspace section of systrace reads kernel messages via the device and invokes “ioctl” calls for the device in order to return messages.

Systrace Takes the Helm

An application must be launched by the “systrace” userspace utility to initialize

Listing 1: Systrace Policy Syntax

```
01 filter = expression "then" action errorcode logcode
02 expression = symbol | "not" expression | "(" expression ")" |
03           expression "and" expression | expression "or" expression
04 symbol = string typeoff "match" cmdstring |
05           string typeoff "eq" cmdstring | string typeoff "neq"
cmdstring |
06           string typeoff "sub" cmdstring | string typeoff "nsub"
cmdstring |
07           string typeoff "inpath" cmdstring | "true"
08 typeoff = /* empty */ | "[" number "]"
09 action = "permit" | "deny"
10 errorcode = /* empty */ | "[" string "]"
11 logcode = /* empty */ | "log"
```

Table 2: String Matching with Systrace

Operator	Function
match	Is true if the file name for a glob-pattern matches “fnmatch(3)”
eq	Is true if the syscall argument exactly matches the string following the operator
neq	Logical negation of “eq”
sub	Looks for matches in a substring of the system call argument
nsub	Is the logical negation of “sub”
inpath	Is true if the syscall argument is a subpath of the string following the operator
re	Looks for matches for a regular expression in the syscall argument

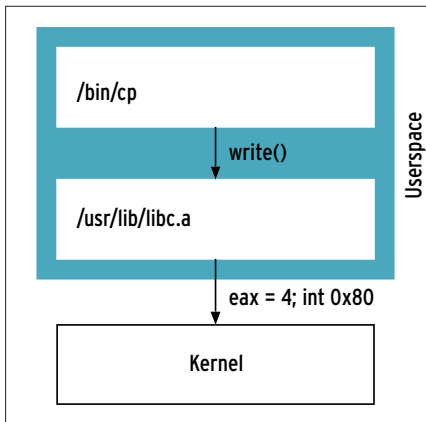


Figure 1: Userspace processes use libc to initiate system calls. "/bin/cp" calls the "write()" library function, which selects the appropriate syscall via the "eax" register

systrace. The command opens a session to the kernel portion of systrace by opening the "/dev/systrace" device. It forks a new process, uses an "ioctl" command to tag the process, and uses "execve()" to run the application it needs to monitor.

The modified call gateway checks each system call to discover whether or not the process has been tagged. If so, control is passed to the systrace hook. Systrace looks up the system call number in its policy cache to ascertain whether or not a simple rule exists for the call (that is "permit" or "deny" without any additional arguments).

If systrace discovers a simple rule, it performs the action described by the rule. If there is no cached action, systrace turns to its userspace counterpart to ask for a decision.

Systrace and Monkey.org

Monkey.org is an example of systrace in a production environment. The private UNIX shell provider runs the processes of its approximately 200 users on systrace. The admins have defined policies for every program installed at monkey.org for this purpose.

Every user's login shell is set to "stsh" (systrace shell). "stsh" spawns the user's real shell as a systraced process allowing every process a user starts to be monitored. Systrace runs in enforcement mode and thus denies any syscall not envisaged by a policy, and logs any contraventions. The administrators can parse their logs and change their policies accordingly, if required.

Listing 2: Sample lines from the XMMS policy

```
01 linux-fsread: filename eq "/etc/ld.so.preload" then permit
02 linux-fsread: filename eq "/etc/ld.so.cache" then permit
03 linux-fsread: filename eq "/lib/libpthread.so.0" then permit
04 linux-fsread: filename eq "/usr/X11R6/lib/libSM.so.6" then permit
05 linux-fsread: filename eq "/usr/X11R6/lib/libICE.so.6" then permit
06 linux-fsread: filename eq "/usr/lib/libxmms.so.1" then permit
07 [...]
08 linux-fswrite: filename eq "/dev/dsp" then permit
09 linux-fsread: filename eq "/home/marius/.xmms/menurc" then permit
10 linux-fsread: filename eq "/dev/mixer" then permit
11 linux-fsread: filename eq "/home/marius/.xmms/xmms.m3u" then permit
12 linux-fsread: filename eq "/home/marius" then permit
13 [...]
14 linux-pipe: permit
15 linux-clone: permit
16 linux-rt_sigsuspend: permit
17 linux-poll: permit
18 linux-getppid: permit
19 linux-kill: pidname eq "/usr/bin/xmms" and signame eq "<unknown>: 32"
then permit
```

To do so, the kernel component queues a message which is then forwarded to the userspace systrace component via the "/dev/systrace" device. The message contains the number and any parameters for the system call. The userspace component looks up matches for the syscall and parameters in the policy for the current application and tells the kernel what action to perform if a match is found. If it cannot find an appropriate rule, systrace will interactively prompt the user for a decision. In enforcement mode any actions not defined in the policy will be prevented and logged.

Decisive Users

Systrace uses either the console or a GUI to prompt the user for a decision, displaying the syscall and any parameters in both cases. The user can decide to permit or deny the action, or create a new rule. If the user chooses to "deny", the error message defined in the "deny" request is returned (this defaults to "EPERM"). Systrace will allow the system call to be dispatched if the user chooses "permit".

As an additional security measure, the kernel kills any processes currently being monitored by systrace if the monitoring process (that is "systrace") terminates in an unexpected fashion.

In some cases the userspace systrace component wants to know the return value of the system call, and the kernel component indicates the value after the call has been processed. This is particularly useful for "execve()" calls. In the case of successful system calls, systrace will use the policy assigned to the new program in future.

First Training – then Production

To use Systrace on a process, it has to be started with the "systrace" utility, for example, to run netscape under systrace:

```
% systrace netscape
```

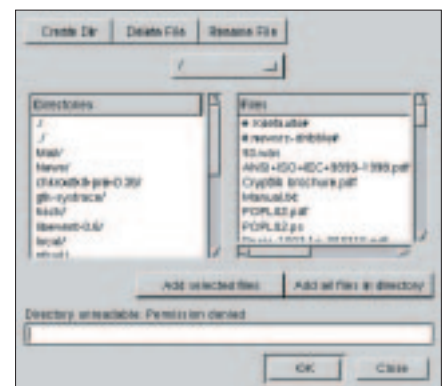


Figure 2: In interactive mode systrace warns the user when a program contravenes policy rules. In our example, XMMS has attempted to read the root directory "/"



Figure 3: A systrace policy denies access to “/” and returns an “EACCES” error message. The file dialog in XMMS reacts by displaying the message “Directory unreadable: Permission denied”



Figure 4: Systrace catches the configure script in a trojaned version of fragroute: The source package has been manipulated by malevolent hackers and attempts to open a TCP connection to port 6667 on IP 216.80.99.202

This will cause the tool to launch a Netscape process which it tags for monitoring. If a policy already exists for the application, it will simply be applied, if not, systrace will create a new policy. Systrace notifies the user whenever it encounters a system call that does not match an entry in the policy.

Systrace also provides a training mode launched by the “-A” flag. In this mode the behavior displayed by the application is defined as normal. Systrace monitors the system calls initiated by the application and generates an appropriate policy from them. Let us look at XMMS:

```
% systrace -A xmms
```

In training mode you should launch everything that is considered normal for the program, such as play a few songs, in the case of XMMS.

Taming XMMS

After quitting the XMMS application systrace will store the new rules in “\$HOME/.systrace/usr_bin_xmms”.

Listing 2 provides a few examples. The policy comprises about 100 entries that mainly refer to file system access for libraries and plug-ins, additionally the sound device is opened and used. It makes sense to check the generated policy for any unusual parts – just in

case you were attacked while going through the training stage. Systrace would classify the attacker’s activities as normal and allow them in future.

Using the policy, systrace can now monitor the application: “systrace xmms”. This should allow XMMS to run normally, unless the user tries something not envisaged by the policy. A user might attempt to access the root directory “/” by selecting it from the file selection dialog box in XMMS. This would provoke a systrace error as can be seen in Figure 2. The following policy entry would then prevent this kind of access permanently:

```
filename eq "/" then ⤵
deny[EACCES]
```

The entry also specifies that syscall should return an “EACCES” message when denying access, informing XMMS that it does not have the required permissions. XMMS then informs the user that it cannot read the directory (see Figure 3). If XMMS contains a bug that allows an attacker to access a user’s private files, systrace would notice this abnormal behavior and warn the user. XMMS does not normally need access to these files, and the policy has no rules on them.

Once policies have been defined systrace can be run in enforcement

mode. In this mode, systrace will not prompt the user if it notices abnormal behavior, instead denying the syscall and writing a message to the syslog.

Conclusion

Systrace places applications in a policy jail, thereby restricting the damage a security hole can cause (see Figure 4). Effectively the policy describes an application’s intended usage of system calls. When systrace is running, it informs the user about system call activity not covered by the policy. The user can then decide whether systrace should permit or deny the call. ■

INFO

[1] Systrace home page: <http://www.citi.umich.edu/u/provos/systrace/>

THE AUTHORS

Marius Aamodt Eriksen is an open source developer and a computer engineering undergraduate student at the University of Michigan in Ann Arbor, Michigan. He also ported systrace to Linux.

Niels Provos has developed numerous Open Source Programs, systrace being one of them. He is currently working on his doctorate at University of Michigan in Ann Arbor, Michigan. His research topics are computer and network security. He is also interested in steganography.