CommInst: Instrumenting Process Communication

Sumit Kumar, Rajesh Rajamani, Abhishek Saxena
(sumit@cs.wisc.edu, raj@cs.wisc.edu, asaxena@cs.wisc.edu)

Abstract

We have developed a tool that can be used to automatically discover communication channels of processes for which the source code is not available. Currently, our tool reports the operations executed on file, socket and pipe communication primitives. The tool helps users in protecting their machines by limiting the execution of library functions executed by a process and hence, is a valuable tool for analyzing an untrusted application such as a virus or a worm. Our tool uses binary rewriting technology to control the execution results of library functions.

1 Introduction

Many times a user has relatively little knowledge about the functionality of an application running on her machine, and with the proliferation of spyware[13] and other software that allow third party "plug-ins", intrusion of privacy has become a major concern. The user would therefore, like to know the files that the process is accessing, the network connections it is utilizing, its relationships with other processes and the child processes that it may have running.

Such a process might be a trusted or an untrusted application. An administrator might notice a process that should not be running or one that shows suspicious activity. It might be a worm that has infected the machine or it might be a process started by another application. In any case, with viruses and worms threatening businesses and computers these days, she is interested in knowing more about the process to make sure it is not doing anything destructive. She can find information about this process by using some of the techniques she already knows such as gathering information from /proc and analyzing the system logs, but what she really wants to know is what this process is doing. Is the process sending out some sensitive information over the network? Is it writing to some files that should never be written to by an application? Has the process opened up any backdoors on the machine?

There might also be a child process that was started by another application but even though the original application terminated, the child process is still running. For example: a process loadqm.exe starts when Microsoft Money 2002 web services are accessed. This query manager (loadqm.exe) has traffic flowing out of the computer into the Internet even when the user is not explicitly accessing any data. Even after Microsoft Money is closed, loadqm.exe is responsible for the traffic outflow [2,3]. This makes a user

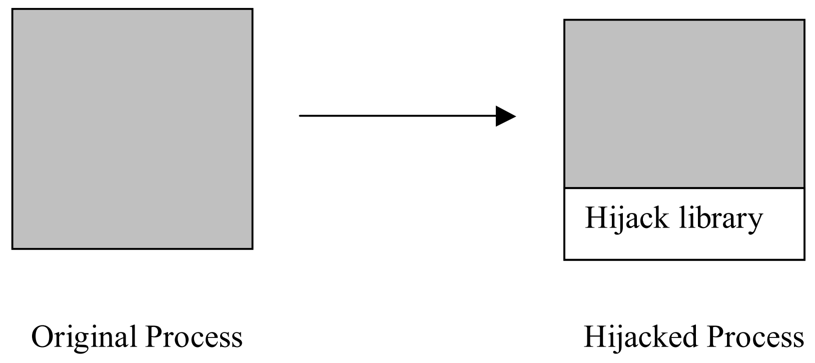


Figure 1: Hijack library is loaded into process run-time image using DynInst

wonder whether any private information is being leaked out. Is it anything to be really concerned about or is it just a harmless application trying to make the life easier by gathering more data?

There can also be applications where we do not have access to the source code or the source code is too complex and we are simply interested in understanding how the process communicates as it executes.

All of these scenarios motivate the need for a tool that can analyze a process' communication primitives especially if it is malicious without allowing it to harm the machine on which it is executing.

2 Design Issues

In our discussion we will use the terms *process* and *application* interchangeably for an application that a user is interested in analyzing. Earlier studies have talked about analyzing the file interactions of an application by sandboxing it [4]. We are interested in knowing about *all* communication channels of a process. A process can communicate using files, sockets, pipes, shared memory, message queues and semaphores. It can also enlist other auxiliary processes to manage its communication. A user would be interested in knowing which of these channels a process is using, how it is using them, the order in which it is accessing the various channels and the relationship of the process to other processes. A user would like to get information about the communication events, such as the opening of a file, as and when they happen.

After a user understands what a process is doing, she might want to control the outcome of the events on these communication channels. For example, if a process is opening a file, the user may want to disallow it from opening that file. This control over the outcome of actions performed by an application will help the user in protecting the machine from any malicious activity of the process and at the same time help her in understanding how a process uses the communication primitives. In order to control the outcome of these events fully, the user must also be able to provide false information to the process. For example, if a process is opening the file /bin/crucialapp for writing, the user should have the ability to spoof the process into believing that it is indeed opening /bin/crucialapp, when in fact it was given a handle to /tmp/bin/crucialapp. This conditional feeding of false information to the process is necessary, as we do not want an untrusted process to quit if it cannot perform a malicious task, but we want to be able to continue to observe its behavior.

We can conceptually separate these two tasks of monitoring or analyzing an application's communication primitives and controlling them into the *Analysis task* and the *Control task* respectively. Analysis task includes discovering the various communication channels that a process is using and the control task includes the ability to control these channels. A tool designed to automatically discover an application's communication channels must be able to perform both these tasks.

On Linux, we can perform the analysis task by gathering information from the /proc interface, but this gives us a rather incomplete picture about the communication channels of a process. In addition, /proc gives us information, such as opening up of a file, only after it has occurred, but we would like to obtain this information while the event is taking place so that we can control the outcome of the event. A process communicates with the operating system or other processes using system call functions. So, what we really need is a way to intercept the library function calls that a process makes, look at the arguments passed to these calls and use these to create the bigger picture. Intercepting these calls will give us the additional advantage of changing their outcome, thus allowing us to control the application as it executes. One way to find out the library calls made by a process is to use *strace* [5], but strace provides this information after the calls have been made.

We can intercept the library calls by using an interposition agent using a tool like bypass [6]. Interposition agents can be easily used on dynamically linked applications and there is no need to relink or recompile

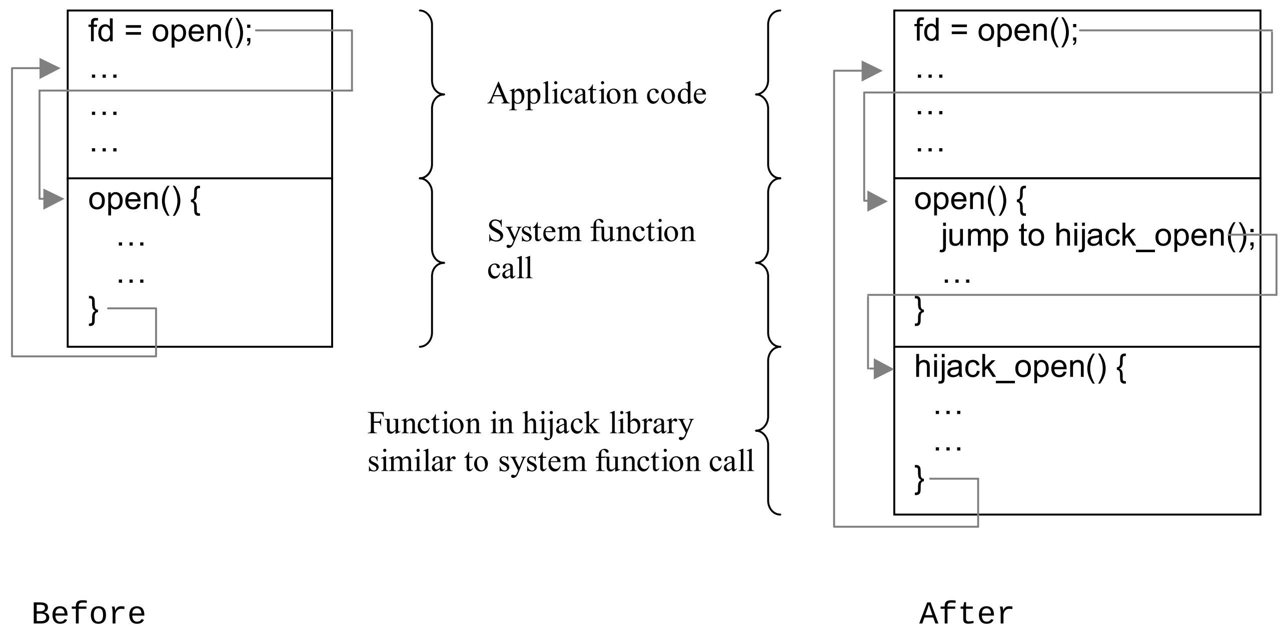


Figure 2: DynInst is used to replace system function calls to functions calls in hijack library

them. However, interposition agents do not work well in all cases and this approach cannot be taken for statically linked applications. Another way to intercept the library function calls is to relink the library calls to our own library. But, in this case, the relinking would be required every time an application loads a new dynamic library into its run-time image and this approach is not very general.

3 Our Approach

The approach we take is that when a user wants to start monitoring a process, a library is loaded into the run-time image of the process (Figure 1). We refer to this library as the *hijack* library. The hijack library has function calls with signatures (parameter types, number of parameters and return value) exactly same as the library functions that we are interested in monitoring and controlling. We then instrument the library calls present in the run-time image of the application, replacing them with calls to the similar functions in our hijack library (Figure 2). The loading of the hijack library into the application's run time image and instrumenting the library calls is achieved by using DynInst's [1] binary rewriting technology. DynInst's replaceFunction() is used to insert a jump snippet at the entry point of the library call which routes the flow of control from the library call to the similar function in the hijack library (Figure 2).

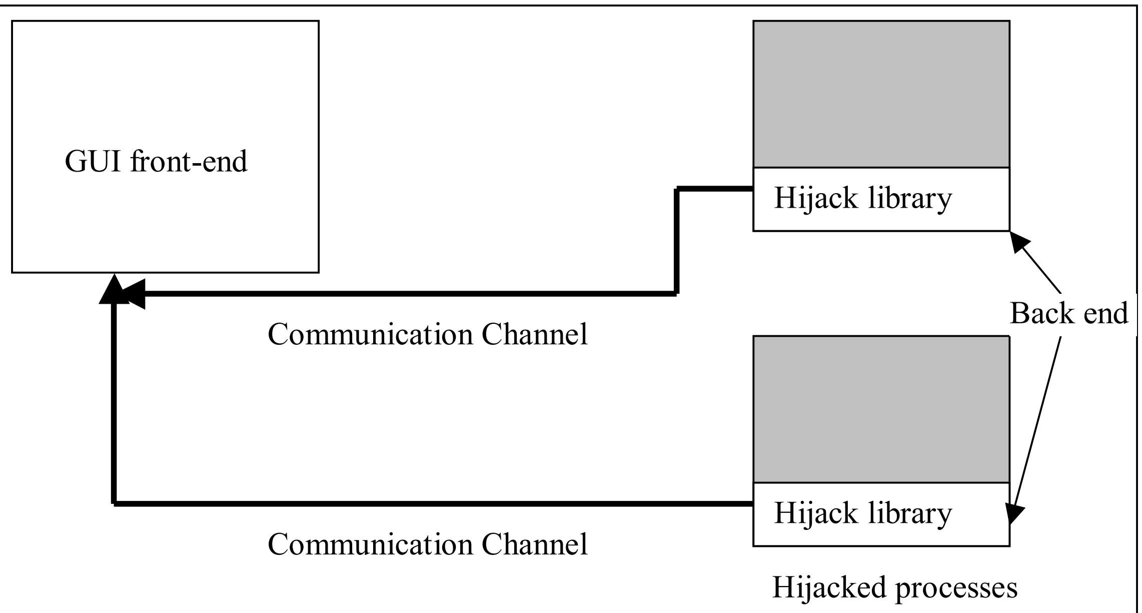


Figure 3: CommInst architecture. There is one GUI front-end and a back end for each process being monitored. Reports about communication events are sent by the hijack libraries to the front-end using Unix domain level sockets labeled as Communication channel in the figure. The front-end listens for these messages on a well-known port.

We refer to these instrumented library calls as *hijacked* and the process is referred to as hijacked or mutated [10]. So, in effect, the standard library calls have been routed to the hijack library. Now, we will know when an application attempts to execute an operation on a communication channel and the value of arguments passed to the library calls. Using the knowledge of the library calls made and the arguments passed to them we can monitor and report application's behavior.

Users must also be able to protect the system from malicious activities of a process. In our framework, users do this by specifying the policies to control the outcome of process actions. These actions distill down to library functions called by a process (for example, files are opened using the open() libc call). Since we have instrumented the standard library calls, in order to control an application's behavior, all we have to do is control the outcome of the hijacked function calls. If we want the hijacked call to succeed, we make that call in our hijack function or if we want it to fail we simply return an error. We can also change the values of the arguments passed and then make a system call to execute the operation that the hijacked call would have executed. This way we have the ability to stealthily control a mutated application's communication channel.

Communication Channel	***Library calls instrumented***
File	open, close, read, write
Socket	socket, socketpair, connect, bind, accept, send, recv, sendto, recvfrom, sendmsg, recvmsg, read, write, shutdown, setsockopt , close
Pipe	pipe, read, write, close
Spawning processes	fork, execve

Table 1: Library calls that CommInst instruments and the communication channels that they are used for. Fork and execve though are not related to a communication channel they are important in capturing spawning of processes

We now describe our tool that can be used to understand and control an application's communication channels

4 Architecture

We call our tool CommInst - communication instrumentation. We currently have a working prototype for Linux (kernel version 2.4.10) to monitor a process and control its interactions with respect to these communication channels: files, sockets and pipes. Currently we monitor only the applications that use the standard C libraries in Linux.

The architecture of our tool is shown in Figure 3. Our tool consists of a GUI front-end and one back-end for each process being monitored. The GUI front-end is used to monitor processes and to set policies which are sent to the back end. When the administrator wants to start monitoring a process she specifies this to the front end, which then mutates the process, as specified in section 3, thus creating the back end. The back end then reports back to the GUI front-end whenever an interesting event takes place in the mutated process. We use the terms hijack library and back end synonymously.

4.1 Back end

After the GUI front-end creates a back-end in the process to be monitored, as explained in section 3, using DynInst it also replaces the library function calls that perform operations on files, sockets and pipes.

The library functions that CommInst currently instruments are listed in Table 1. Some of the functions like read(), write() and close() work on both files, sockets and pipes. Pipes are created in a different way using pipe(), mkfifo() or fifo() but once they have been created read(), write() and close() are used to read data from a pipe, write data to a pipe or close a pipe, respectively. By monitoring the library calls in Table 1, CommInst is able to monitor all file and socket operations. For pipes, except limited monitoring for creation of pipes, all other operations are monitored. Creation of pipes is reported only if a pipe is created using the pipe() library call as CommInst does not instrument the other two pipe creation calls - mkfifo() and fifo(). A process can also spawn other processes and a user investigating a process would be interested in knowing that too. To capture this CommInst also replaces fork() and execve() library calls. If CommInst had created a back end in a process before it forked, after fork(), the new child process also has the back end and there is no need to load the hijack library into the child process (this is a feature of DynInst). So, by default we start monitoring spawned processes.

Since the original library calls have been replaced by the hijack library calls (as explained in Section 2), the back end can now handle these calls based on user policies. The calls can be returned with an error value making the application think that the call did not succeed, the back end can make these calls (using SYSCALL) and return the value of the syscall back to the application or the calls can be made with the different argument values than the ones passed by the application.

The appropriate action taken by the hijack library is determined by user policies. These user policies are read by the back end from a file and stored in its memory space. When the front end creates the back end first time, it specifies the file from which the user the policies must be read. The hijack library parses the file and remembers the policies so that it can take appropriate action(s) when the hijacked process calls library functions. If no file is specified at the time of creation of the back end, a default policy file is loaded. After that if the policies need to be changed, front end sends a signal to the hijack library which on receipt of this signal reloads the policies from the file specified in the signal. This way CommInst allows changing of policies while the process is being monitored.

We will now specify what the various policies are and what actions hijack library takes for these policies.

4.1.1 File Policies

A user can specify the action to be taken by the hijack library when operations (open, read, write and close) are performed on files. A file policy consists of two fields, file(s) or directories that this policy applies to and the type of policy.

The type of policy informs the hijack library about the appropriate action to be taken. Allowable values for this field are ALLOW_REPORT, ALLOW, DENY_REPORT and DENY. ALLOW_REPORT instructs the hijack library to allow the operations on files to be completed. The hijack library function that replaced the original library call (section 3), does this by making a syscall() to complete the operation. ALLOW_REPORT also instructs the hijack library to send a report message to the front end about the library function called by the process. ALLOW instructs the hijack library to simply allow the call to proceed and not report it to the front end. DENY_REPORT as the name suggests, disallows the call, returning an error to the application and also sends a report message to the front end specifying the library call attempted by the application. DENY is same as DENY_REPORT except that the report message is not sent to GUI about the library call. ALLOW and DENY help in decreasing the overhead of sending the report message to the front end. Instead of sending a report message for every instrumented library call, back end has to send a report message to the front end for only those operations in which a user is interested.

Policies that appear higher or earlier in the list (section 4.2.2) are given priority over the ones that appear lower. When an operation is performed on a file by the application, the hijack library goes through the list of file policies stored in its memory space and checks to see if a policy is defined for that file. It uses the first match and takes the appropriate action of either allowing or denying the operation attempted on that file (ALLOW or DENY). As explained earlier, it also reports to the front end if the policy was set to report (ALLOW_REPORT or DENY_REPORT)

4.1.2 Socket Policies

A socket policy like a file policy consists of two fields, first field specifies an IP address and port number pair and the second field specifies the type of policy. An IP address and port number can be defined using wildcards (section 4.2.2). The type of policies are same as the ones defined for files, ALLOW_REPORT, ALLOW, DENY_REPORT and DENY.

When a socket operation is attempted by the application, the hijack library looks for a matching IP address and port number in the socket policy list. It uses the first found match and takes the appropriate action specified in that policy of allowing the socket operation to be performed (ALLOW, ALLOW_REPORT) or not performed (DENY_REPORT, DENY). As in files, it also reports to the front end about the library call attempted by the application if the policy was set so (ALLOW_REPORT or DENY_REPORT).

4.1.2.1 Hijack IP

In order to further control the socket communication channel of an application we added a global socket policy. The global socket policy allows a user to specify an IP address to which all the outgoing socket connections will be redirected. We call refer to this IP address as the hijack IP. Hijack IP address allows CommInst to hijack an application's outgoing socket connections redirecting it to any IP address that the user may wish, though the application thinks that it is communicating with the IP address that it specified. This can be helpful in the case when the user is investigating a worm and does not want the worm to carry on real attacks, instead wants to divert the attacks to an IP address that belongs to a machine which she can monitor. For example, user can specify the hijack IP to be 127.0.0.1, which is the local loop back interface, and all the outgoing socket level communications will now be diverted to 127.0.0.1. This can also help the user to find out if her system has any vulnerabilities and how it will respond if attacked by the worm from a different machine.

To implement this functionality CommInst replaces the value of IP address passed as arguments to connect() or sendto() library calls with the hijack IP address and then completes these calls by making a syscall.

We only instrument the outgoing socket operations. For incoming operations it is rather difficult for the hijack library to trick the application. For example, in recvfrom(), the IP address of the remote end point is filled in one of the arguments by the library call when it returns. Since, an application might be expecting the message from any IP address it is hard to determine what address CommInst might fill. Even if CommInst can somehow determine this IP address, the harder part is to construct a legitimate message to be passed back to the application when we never allowed the connection to be established in the first place. In this case, hijack library will have no idea about what a legitimate message might be, since a message is never received.

4.1.3 Pipe policies

A pipe policy consists of just one field - the type of policy which like files and sockets is ALLOW_REPORT, ALLOW, DENY_REPORT and DENY. The operations that CommInst monitors for pipes are opening of pipes, reading and writing to pipes and closing of pipes. The pipe policy tells the hijack library the action to perform when these operations are attempted on pipes. For example, if the policy is set to ALLOW_REPORT pipe operations, all pipe operations are allowed and are also reported to the front end.

4.1.4 Reporting to GUI

A ‘report message’ is sent by the hijack library to the front end if a policy was set to ALLOW_REPORT or DENY_REPORT. This report message has detailed information about the library call attempted by the application and the value of arguments passed to the library call. For example, when connect() library call is made by the application and the policy for sockets is set to ALLOW_REPORT or DENY_REPORT, a report message is sent to the front end specifying the file descriptor of the socket on which this call was made and the IP address and port number to which this call was attempting a connection.

The report messages are sent using UNIX domain sockets. The front end listens for these report messages on a specific port to which all back ends send their report messages.

4.1.5 Controlling communication channels

Currently the control of the communication channels is very limited. The channels are controlled using the policies specified above. Though the control discussion accompanied the policies, we are again presenting it here for brevity.

For files: CommInst can either allow or disallow the file(s) that an application attempts to open. It does not redirect an application’s file access to another file. Though we believe this is a nice feature to have in CommInst it is not of utmost importance. The sandboxing of file accesses is discussed in detail in [4]. Also users can mount file systems and restrict an application’s access to the mounted file system. Some of the sensitive files these days are already protected. For example, shadow password files are used instead of /etc/passwd file. Also most of the network services run with restricted privileges to modify crucial files.

For sockets: Like files, CommInst can either allow or disallow the socket connections based on IP addresses and ports of the remote end. Unlike files, CommInst can redirect the outgoing socket transmissions to another IP address (hijack IP). We present our results in section 5 using the hijack IP.

For pipes: CommInst only allows or disallows opening, reading and writing to pipes.

4.2 Front end

Graphical User Interface (GUI) of the front-end has been designed using the Qt Designer from Trolltech[7]. This GUI front end is used to monitor and control multiple processes. The front-end runs as a stand-alone application. A user specifies the process she is interested in by:

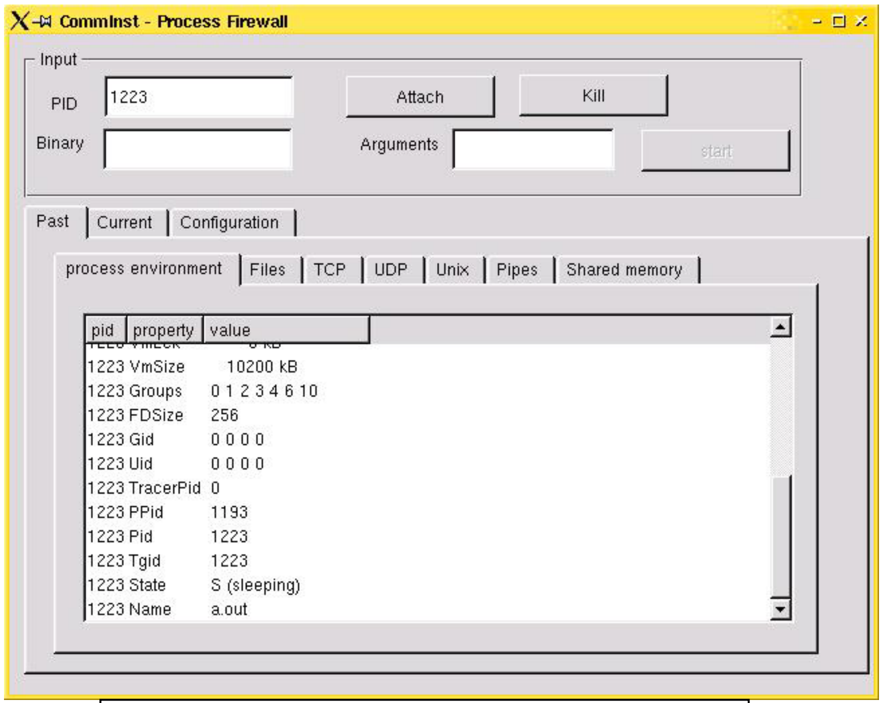
1) Specifying the process id of a running process - comminst starts monitoring and controlling the process from that point onwards (using 'Attach' button in Figure 4)

2) Specifying the name of an executable - comminst starts the process and monitors and controls it as soon as it starts executing (using 'Start' button in Figure 4)

There is no constraint on the type of process that a user can specify except the ones mentioned in section 6 (current limitations and future work).

4.2.1 Reporting communication events

After loading the hijack library into an application, CommInst collects all information about the process once from the /proc interface and displays it under the 'past tab' (Figure 4). This information is further sub divided into -Process environment, Files, TCP, UDP, Unix domain sockets, Pipes and Shared memory. Though CommInst collects information about shared memory once when it starts monitoring a process from /proc, it does not support monitoring or controlling access to shared memory.

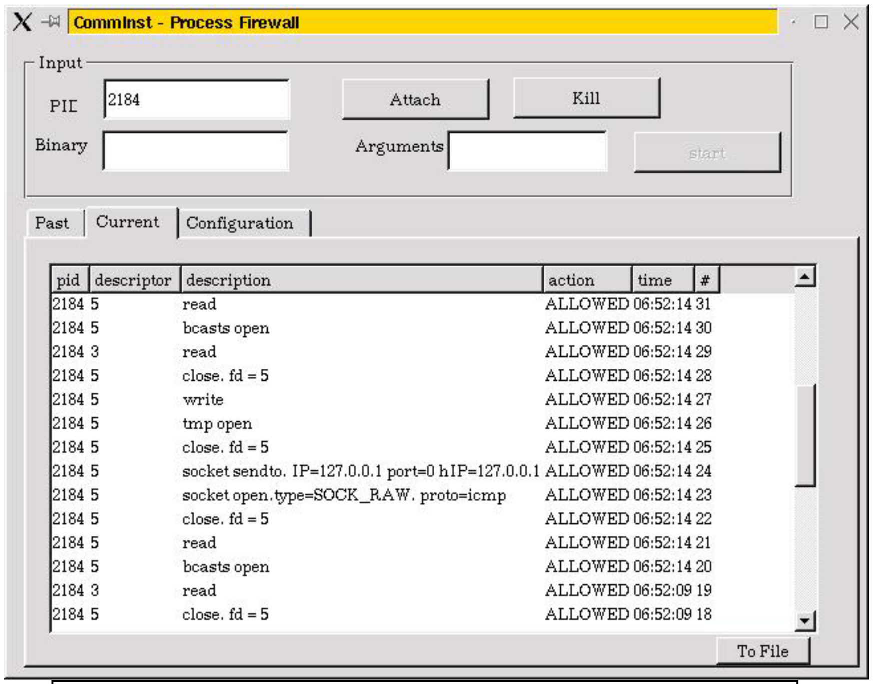


PID 1223

pid	property	value
1223	VmSize	10200 kB
1223	Groups	0 1 2 3 4 6 10
1223	FDSize	256
1223	Gid	0 0 0 0
1223	Uid	0 0 0 0
1223	TracerPid	0
1223	PPid	1193
1223	Pid	1223
1223	Tgid	1223
1223	State	S (sleeping)
1223	Name	a.out

Figure 4 – CommInst attached to process 1223

The front-end also gets the report messages (section 4.1) from the back ends in hijacked processes. On receiving these, it timestamps them, and displays them to the user in the order which they were received. This order is same as the order in which these messages originated from the hijacked application as we are using unix domain sockets to send these messages from the back end to the front end. The report messages are displayed to the user under the 'Current' tab (figure 5). User also has an option of writing these messages to a log file.



PID 2184

pid	descriptor	description	action	time	#
2184	5	read	ALLOWED	06:52:14	31
2184	5	bcasts open	ALLOWED	06:52:14	30
2184	3	read	ALLOWED	06:52:14	29
2184	5	close. fd = 5	ALLOWED	06:52:14	28
2184	5	write	ALLOWED	06:52:14	27
2184	5	tmp open	ALLOWED	06:52:14	26
2184	5	close. fd = 5	ALLOWED	06:52:14	25
2184	5	socket sendto. IP=127.0.0.1 port=0 hIP=127.0.0.1	ALLOWED	06:52:14	24
2184	5	socket open.type=SOCK_RAW. proto=icmp	ALLOWED	06:52:14	23
2184	5	close. fd = 5	ALLOWED	06:52:14	22
2184	5	read	ALLOWED	06:52:14	21
2184	5	bcasts open	ALLOWED	06:52:14	20
2184	3	read	ALLOWED	06:52:09	19
2184	5	close. fd = 5	ALLOWED	06:52:09	18

Figure 5 – CommInst displaying the report messages received from the back end in process 2184

4.2.2 Controlling communication events

User can control an application's file, socket and pipe operations by specifying policies. These policies are saved to a file and a signal is sent to the hijack library in the process being monitored. On receiving this signal, the hijack library reads the policies from the saved file and loads them into its memory space.

4.2.2.1 Policies

The policies specify what actions must be taken when the hijacked application performs operations on the communication primitives. Using these policies a user is able to control the hijacked application's communication channels. These policies can be changed on the fly making the control more flexible. For example, a user who was earlier allowing the process to open any file can now decide that she wants the application to be able to open only certain files and can change the policy without having to restart monitoring.

Section 4.2.1 describes the actions that the back ends takes for these various policies. Here we describe in detail how users specify and enter policies using the GUI front end.

4.2.2.1.1 Entering Policies

Policies are specified on the 'Configuration' tab of CommInst (figure 4). The 'Configuration' tab has following four fields to add a new policy:

1. *Descriptor type*: This field specifies the communication primitive for which this policy will be used; current options are FILE, SOCKET or PIPE.

2. *Action*: This field specifies the operations or actions that this policy applies to; currently the only option is ALL.

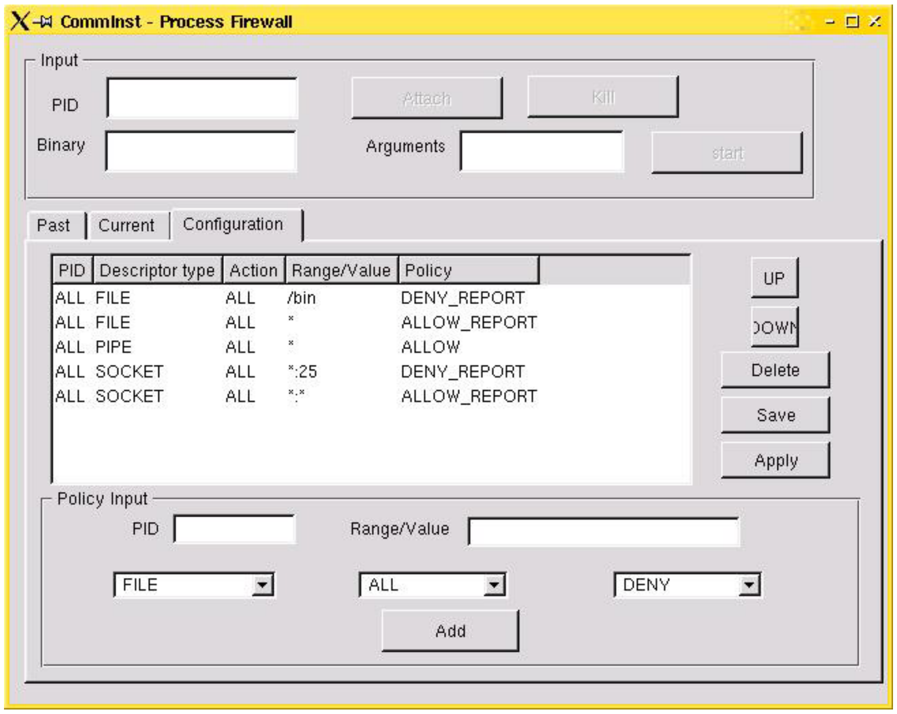
3. *Policy*: This field specifies what this policy means; options are ALLOW, DENY, ALLOW_REPORT, DENY_REPORT. ALLOW value in this field means that the current policy allows actions in the 'Actions' field on the communication primitive in the 'Descriptor type' field. DENY means that this policy does not allow operations in the 'Actions' field on the communication primitive in the 'Descriptor type' field. ALLOW_REPORT and DENY_REPORT, in addition to ALLOWing or DENYing, specify that report messages (as described in section 3.1) must be sent to the GUI front-end by the hijack library, when the operation specified in the 'action' field is performed on the communication primitive in the 'Descriptor type' field. If the user does not specify DENY_REPORT or ALLOW_REPORT for some communication primitive, then reports about actions/operations performed on that primitive are not produced (hijack library does not send report messages to the GUI front-end). This reporting option is useful because users

may not be interested in getting the reports about all actions for all communication channels. This option thus also allows us to reduce the overhead of sending the report messages from hijack library to GUI front-end every time an instrumented function call is made.

4. *Range/Values*: This field is closely tied to the 'Descriptor type' field.
For Files: users can specify a file, files (using a wildcard) or a directory. Specified actions will be allowed or denied on the file(s) or directory specified in this field.
For sockets: users can specify an IP address and a port number using this format N.N.N.N:port where N.N.N.N is an IP address like 10.2.11.102. Users can use a wildcard in place of any N or the port. For example, user can specify 10.2.*.*:*
For pipes: this field is currently not used.

4.2.2.1.2 Policy Examples

Here are some examples of policies that a user might want to specify from Figure 6:



PID	Descriptor type	Action	Range/Value	Policy
ALL	FILE	ALL	/bin	DENY_REPORT
ALL	FILE	ALL	*	ALLOW_REPORT
ALL	PIPE	ALL	*	ALLOW
ALL	SOCKET	ALL	*:25	DENY_REPORT
ALL	SOCKET	ALL	*:*	ALLOW_REPORT

Figure 6 – CommInst’s configuration tab is used to specify policies

Example 1 (ALL, FILE, ALL, /bin, DENY_REPORT): a user may specify to disallow a process to either open a file, read a file or write a file from the directory /bin and also specify to report (DENY_REPORT) the operation that the process was trying to execute (open, read, or write).

Example 2 (ALL, FILE, ALL, *, ALLOW_REPORT): a user may specify to allow all types of actions on all files and also specify that these actions be reported. So, in this case, if a file is being opened, read or written to, the hijack library will send report message to the GUI front-end and allow the system function calls (open, read or write) to be executed normally.

Example 3 (ALL, PIPE, ALL, *, ALLOW): a user can specify to let a process create pipes.

Example 4 (ALL, SOCKET, ALL, *:25, DENY_REPORT): a user can specify to deny any sockets that can communicate with port 25 (sendmail port) on any machine. Even though denied, the socket operations (connect, bind, send, recv, sendto, recvmsg, read, write) must be reported to the GUI front-end (DENY_REPORT).

Example 5 (ALL, SOCKET, ALL, *:*, ALLOW_REPORT): for all sockets allow all actions (connect, bind and so on) and report these actions.

As seen in figure 6, user can add a number of these policies on the 'Configuration' tab of CommInst. Policies that are higher in the list take precedence over the policies that are lower in the list. Here, for sockets using line 4 and line 5 a user has specified to deny any socket connections to port 25, but allow connections to all other ports.

Currently, we are not using the PID field on the 'Configuration' tab. This field is for future versions to specify a policy for a specific process. Also, only allowable value for 'Action' field (figure 6) may seem very restrictive. For example: a user might want to specify that she wants to allow reading of all files in /bin, but does not want to allow writing to any file in /bin. Adding both of the above-mentioned features is straightforward but they do not currently exist.

5 Results

Since CommInst was written with mainly a security concern in mind, we wanted to see how it performs on real world malicious applications. We wanted to test it with viruses and worms that run on Linux and are written using standard C libraries. One such worm that we were able to find was Mscan worm. The Mscan worm is completely written in C and as such is not harmful as it looks for exploits for which patches and upgrades are already available. But it can be easily used to compromise machines that are still running older versions of the exploited network services without appropriate patches.

5.1 Mscan worm's propagation mechanism

Mscan worm like other worms spreads from one machine to another by probing machines and looking for exploits in their network services. The exploits that it looks for a CGI hole, wu-ftp buffer overflow exploit [11] and rpc.statd buffer overflow exploit [12]. Though the current Mscan worm is not deadly as these exploits are quite old and the parts to gain shell access are commented out in the source code, anyone with a working knowledge of C can easily make changes to it and add new exploits to it.

Mscan worm generates random IP addresses one by one and attacks these IP addresses looking for the above exploits. If it is able to find a vulnerable machine, it tries to copy itself to the remote machine by

```
1  15:55:05  1817  ALLOWED  9  socket open.type=SOCK_STREAM. proto=ip
2  15:55:05  1817  ALLOWED  9  socket connect. IP=196.101.122.1 port=80.
                                 hIP=127.0.0.1
3  15:55:05  1817  ALLOWED  9  close. fd = 9
4  15:55:05  1817  ALLOWED  9  socket open.type=SOCK_STREAM. proto=ip
5  15:55:05  1817  ALLOWED  9  socket connect. IP=196.101.122.2 port=21.
                                 hIP=127.0.0.1
6  15:55:05  1817  ALLOWED  9  close. fd = 9
7  15:55:05  1817  ALLOWED  9  socket open.type=SOCK_STREAM. proto=ip
8  15:55:05  1817  ALLOWED  9  socket connect. IP=196.101.122.3 port=111.
                                 hIP=127.0.0.1
9  15:55:05  1817  ALLOWED  10  /etc/resolv.conf open
10 15:55:05  1817  ALLOWED  10  read
11 15:55:05  1817  ALLOWED  10  read
12 15:55:05  1817  ALLOWED  10  close. fd = 10
13 15:55:06  1817  ALLOWED  10  socket open.type=SOCK_DGRAM. proto=udp
14 15:55:06  1817  ALLOWED  10  socket bind. IP=0.0.0.0 port=721
15 15:55:06  1817  ALLOWED  10  socket setsockopt
16 15:55:06  1817  ALLOWED  10  socket sendto. IP=196.101.122.3 port=111
                                 hIP=127.0.0.1
17 15:55:06  1817  ALLOWED  10  socket recvfrom. IP=127.0.0.1 port=111
18 15:55:06  1817  ALLOWED  10  close. fd = 10
19 15:55:06  1817  ALLOWED  10  socket open.type=SOCK_STREAM. proto=tcp
20 15:55:06  1817  ALLOWED  10  socket bind. IP=0.0.0.0 port=722
21 15:55:06  1817  ALLOWED  10  socket connect. IP=196.101.122.3 port=32768.
                                 hIP=127.0.0.1
22 15:55:06  1817  ALLOWED  10  write
23 15:55:06  1817  ALLOWED  10  read
24 15:55:06  1817  ALLOWED  10  close. fd = 10
25 15:55:06  1817  ALLOWED  9  close. fd = 9
```

Figure 7

starting a shell for wu-ftp and rpc.stad exploits. It is not very clear what it does when it encounters a CGI exploit.

5.2 CommInst and Mscan worm

The results from monitoring and controlling of Mscan worm are shown in figure 7. Since, we had access to Mscan's binary, we specified the binary name to CommInst for starting the worm and monitored it from then onwards (section 4.2). Figure 7 shows the report messages that were sent by the back end (hijack library) to the front end.

The first column in Figure 7 shows the serial number of the report message it received. Whenever the front end receives a report message it assigns a serial number to it before displaying it to the user. The second field is the time stamp when the front end received this message. The third column is the PID of the process being monitored, fourth column shows the policy action taken by the hijack library (socket operation was ALLOWED by the hijack library) and fifth column shows the file descriptor on which the communication event in the last column took place. Last column shows the communication event that happened and more details related to. In the last column 'proto' stands for protocol, 'fd' stands for file descriptor and 'hIP' stands for hijack IP as described in section 4.1.2.1.

From figure 7, lines 1 to 3 (serial numbers 1 to 3), we can see that Mscan worm attempts to open a connection to IP address 196.101.122.1 port 80. This connection is hijacked by CommInst and redirected to IP 127.0.0.1 which is the local loop back interface. Though we had 'httpd' running on port 80, due to a bug in CommInst implementation the connection was not made and since the connect() call failed Mscan worm closes the socket (line 3). In lines 4 to 6, Mscan worm attempts to open the connection to IP address 196.101.122.2 port 21 on which wu-ftp service runs. CommInst again redirected this connection to the local loopback interface. We did not have a wu-ftp server running on port 21 and thus the worm closes the connection (line 6). Lines 7 to 25 show the worm's attempt to exploit the rpc.statd service that runs on port 32768. In order to get the port number on which rpc.stad runs, an application first needs to contact the 'portmap' service on port 111. This can be seen in lines 8 to 21 which were the result of making the library call clnttcp_create(). The worm tries to exploit the buffer overflow by writing to the connected socket in line 22 and then waits on line 23 to see if it gets a response back. If it does get a response back then it means that it was not able to exploit the bug, if it does not get a response back within a certain time period, then it assumes that the payload that was sent on line 22 has been able to open up a backdoor. In figure 7, we were running the version of rpc.statd that did not have this exploit and thus, the worm receives a response from the rpc server and closes the connection (line 24)

This is one of the tests that we performed and it shows that CommInst is quite effective in helping a user understand how the worm works and the network services it tries to exploit. Most importantly, we were able to gather this knowledge without actually allowing the worm to probe someone else's machine but redirecting connections to a machine that belonged to us (local loopback interface in this case).

6 Future Work

CommInst currently runs only on Linux though our framework is not architecture dependent and CommInst can be ported to other platforms. CommInst handles only files, sockets, pipes and forking of processes though a user might be interested in other communication channels like shared memory and message queues. Currently, the control of the communication channels is limited and we provide false information to control a channel only in the case of outgoing socket communications (section 4).

The biggest limitation of our current approach is that we instrument only the standard library calls and an application might make system calls directly for performing operations on communication primitives. CommInst cannot provide an insight into applications that might directly execute assembly code as it handles only those applications that are written using standard C libraries. Other limitations are due to DynInst that does not provide full instrumenting support for multithreaded applications and applications with stripped binaries [1].

Earlier we also had ideas about making the tool more interactive by implementing SUGGEST policy. Specifying SUGGEST policy for a channel would mean that when an operation is attempted by the application on that channel, the application would be blocked from executing and a message would be sent to the user asking him to suggest the action that must be taken. User would then have to decide whether to allow the application in completing that operation on the channel or disallow the application from completing it. This decision would then be sent back to the hijack library that would then take the action specified by the user. We were not able to implement this functionality due to time constraints.

Almost all of the viruses and worms try to hide and reduce the information available to the kernel about their processes. They might do this by clobbering the arguments passed to them by the shell when their binaries are launched, by becoming a daemon thread or by installing a kernel module to hide the process. We did not investigate these aspects of a malicious application.

It would also be nice to know the process at the other end of a communication channel. For files, users might want to know if a producer and consumer relationship exists between processes, i.e., if one process is writing to a file is there another process reading from that file. For pipes the users might want to know the processes on the opposite ends of a pipe. CommInst currently does not support these functionalities. A way to get this information would be, when an operation is performed on a file descriptor (it can be a file, socket or pipe), to search in /proc for that file descriptor and see if any other process also has a file descriptor pointing to the same i-node. This approach can add considerable overhead to the tool.

7 Related Work

Janus[8] is a framework for running untrusted applications and preventing them from harming the machine on which they are running. Janus does this by controlling the execution of system calls which is accomplished by using the /proc interface on Solaris. Users give the framework, a policy file, which is used to determine the system calls that will be blocked and the ones that will not be blocked. The blocking/unblocking decision is based on the system call and the arguments that are passed to a system call. Janus' main aim is to run untrusted applications whereas our approach is to discover a process's communication channels. CommInst reports to the user the library calls made by a process whereas Janus does not. Janus' approach to restricting a process from performing malicious activity is very similar to ours conceptually, but is different in implementation. CommInst instruments the system function calls or the library functions where as Janus monitors the system calls using Solaris' /proc interface. We do not use the /proc interface to control the untrusted activities as Linux does not provide this sophisticated control of a process using /proc to control another process' system calls. Using ptrace was not an option as most of the viruses and worms quit, if they find out from running small tests, that they are being debugged using by ptrace. In addition, Janus has an extra overhead when checking whether a system call should be executed or not, as this decision is taken by the process that is controlling the untrusted application and hence there is a context-switch before the decision can be taken.
Other approaches have been to sandbox the application by restricting its file access [4] or use *program shepherding* to monitor its control flow and use policies to restrict the execution of data as code [9]. These

approaches, like Janus, concentrate on being able to run untrusted applications rather than attempting to understand process's communication primitives.

8 Conclusions

Our tool CommInst can be used to discover a process's communication channels. CommInst reports the exact sequence in which a process tries to use its communication channels. Currently, it can report about a process's file, socket and pipe operations. CommInst can start analyzing an already running process and the process does not need to be compiled/recompiled in a certain manner. CommInst has the ability to analyze multiple processes at the same time. While analyzing a process, policies can be used to prevent malicious activity. This makes CommInst especially helpful in fast analysis of processes that are untrustworthy like viruses and worms, so that we can come up with a defense against it. If a virus/worm uses multiple alternatives to infiltrate a machine, trying one after another, CommInst policies can be used to discover these various alternatives.

Acknowledgements

We gratefully acknowledge our reviewers Nathan C. Burnett, George Kola, James Nugent and Joel Sommers for their valuable feedback. Our many thanks to Victor Zandy for helping us with numerous questions on DynInst and the DynInst team for allowing us to use their API. We also thank Barton P. Miller for coming up with the idea of a tool like CommInst.

References

[1] DynInst API: http://www.dyninst.org

[2] http://support.microsoft.com/default.aspx?scid=kb;en-us;Q309418.

[3] http://www.virusstriker.com/forum/viewthread.php?FID=4&TID=313

[4] Vassilis Prevelakis and Diomidis Spinellis. *Sandboxing applications.* In USENIX 2001 Technical Conference Proceedings: FreeNIX Track. Usenix Association, June 2001

[5] http://www.liacs.nl/~wichert/strace/

[6] http://www.cs.wisc.edu/condor/bypass

[7] http://www.trolltech.com/qt

[8] Goldberg, Ian, David Wagner, Randi Thomas and Eric A. Brewer. *A Secure Environment for Untrusted Helper Applications*, 1996 USENIX Security Symposium

[9] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. *Secure Execution Via Program Shepherding***.** MIT/LCS Technical Memo LCS-TM-625, February, 2002.

[10] Victor C. Zandy, Barton P. Miller, and Miron Livny. *Process Hijacking.* The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, August 1999, pp. 177-184.

[11] wu-ftp exploit. http://www.kb.cert.org/vuls/id/886083

[12] rpc.stad exploit. http://www.kb.cert.org/vuls/id/34043

[13] http://www.cnn.com/2002/TECH/internet/05/07/kazaa.software.idg/index.html