SUEZ: A Distributed Safe Execution Environment for System Administration Trials

Doo San Sim and V. N. Venkatakrishnan – University of Illinois, Chicago

ABSTRACT

In this paper, we address the problem of safely and conveniently performing ''trial'' experiments in system administration tasks. System administrators often perform such trial executions that involve installing new software or experimenting with features of existing software. Often such trials require testing of software that run on multiple hosts. For instance, experimenting with a typical client-server application requires understanding the effect of the actions of the client program on the server. We propose a distributed safe execution environment (DSEE) where such tasks can be performed safely and conveniently. A DSEE performs one-way isolation of the tasks run inside it: the effects of the client and the server are prevented from escaping outside the DSEE, and therefore are prevented from interfering with the processes running outside the DSEE. At the end of the trial execution, a DSEE allows clear inspection of the effects of running the task on all the hosts that are involved in the task execution. Also, a DSEE allows the changes to the ''committed,'' in which case the actions become visible outside the DSEE. Otherwise, they can be ''aborted'' without affecting the system in any way. A DSEE is an ideal platform through which a system administrator can perform such trials without the fear of damaging the system in any manner. In this paper, we present the design and implementation of a tool called SUEZ that allows a system administrator to create and use distributed safe execution environments. We have experimented with several client-server applications using our tool. By performing these trials in a DSEE, we have found configuration vulnerabilities in our trials that involve some commonly used client-server applications.

Introduction

System administrators and desktop users encounter various situations in their day-to-day activity that require them to download, install and run applications on their machines. One of the most common tasks is that of a ''trial'' run of a piece of software that the administrator. Such a trial is typically done if a system administrator has no prior experience in using that piece of software, but there are several other reasons for such a trial execution.

- *Understanding actions of a program.* Often, system administrators would like to study the impact of executing a particular command on their system. More importantly, they often would like to exercise a particular option in a program, and see the observable effects of exercising that option. For instance, when exercising an option in a particular program, the administrator would like to know the direct and indirect effects of using that option. The abundance of binary programs and programs equipped with graphical user interfaces (as opposed to script based installations) often compound this difficulty, as a lot of critical system changes happen ''behind the scenes.''
- *Testing compatibility with existing configurations* Often a system administrator wonders whether installation of an application will work (co-operatively) with existing packages and configurations. Another issue she is concerned about is about the security of user data that is handled by the application, and whether the data handled by the application is adequately protected through file permissions.
- *Experimenting with new software.* Often users download freeware/shareware from various Internet sources. These software may be untrusted or faulty and hence it is important to understand the effects of these software. Hence, system administrators may wish to perform several walk-throughs of these tools to ensure that they do not create any new problems related to security and/or interoperability.
- *Patch testing.* Application of software patches and updates too early may leave the system with potential interoperability issues created due to the updates. Another issue is with possible bugs in the patches/updates. (This is usually the reason updates are delayed much).

Often the user[1] performs the above tasks while facing the need for an environment that allows convenient study of the impact of such tasks. By impact on the system, we refer to issues related to general operation, interoperability with existing applications and security.

While the goal is to understand the actions of a program in a networked system, we note that users are

[1] In this paper, we use the terms *user* and *system administrator* interchangeably, unless otherwise mentioned.

not interested in every action of a program, but only those actions whose effects they perceive as relevant to system interoperability and security. This requires that we abstract away from internal actions of a program, (such as function calls and assignments to local variables), and focus on *observable* actions of a program. Some examples of such actions are a) addition of new users b) modification to user files c) changes to local configuration files d) changes to boot time scripts.

To understand the impact of such changes on a host, *Safe Execution Environments* (SEE) were proposed in [15, 23] and for the Windows platform in [25]. A SEE uses *one-way isolation* to effect containment of the tasks run inside the SEE. Processes running outside the SEE do not see the changes made by the tasks run inside the SEE. At the end of execution, one can examine the changes made to the SEE environment, and decide whether to keep them or discard them and return to the original state.

A SEE is a highly effective environment to perform system administration trials that involve a single host. However, for tasks that are distributed over a set of networked hosts it is not directly suitable. A typical example is a client-server application where any action triggered by a client may change the system state on the server host. In this case, to understand the changes on the server that were triggered by the actions of the client, we need a *distributed* environment. This paper presents the design and implementation of a SUEZ, a tool that allows for creating distributed safe execution environments(DSEE) to assist in system administration trials.

Let us consider a simple system administration example that involves remote administration of printer software. The Common UNIX Printing System (CUPS) [1] allows remote administration of printers using a specialized port on the printer server (TCP port 631). Now, on loading the printer web interface page from the print server, the user is presented with several options related to adding and managing printers and jobs. Each of these options triggers a specific change in the printer server. For instance, adding a printer requires changes in the server on the printer driver file /etc/cups/ppd, and changes to the /etc/printcap that lists the printers. All these changes take effect when the user executes the command to add a printer. In order to know the specific changes made by a command, the system administrator is left with two options. The first one is to read manuals and other forms of documentation. The second one is the use of low level system tools. While some may argue that these are be viable options in the case of a well-known application such as CUPS, they are unsuitable in the case of new/experimental software, software updates and patches.

Thus, understanding the key impacts of installing a software package/patch requires the following abilities:

1. To make the observable effects of an action on a host transparent to the user: In the above example, the action is the choice selection (through the menu displayed by the browser) to add a printer.
2. To make transparent the observable effects of these actions on other hosts in the network: This corresponds to changes to the files /etc/cups/ppd and /etc/printcap in the printer server.
3. To see the ''difference'' between the state of the system in all affected hosts before the action and after it: For the above example, this requires us to identify the above-mentioned files before and after the add-printer action, and any changes to system objects in the filesystem client (in this case there are none).
4. To correlate the above three to arrive at a complete understanding of the actions of the software under scrutiny: This requires us to log the temporal sequence of actions performed on both the client and server in a unified view.
5. In the event of the system administrator is not satisfied with the results, restore the state of the system to that before that of the start of installation (i.e., undo the effect of observable actions). The un-doing capability is needed to perform any experimentation on real systems.

Related Work

In this section, we discuss related work that are available as options to the system administrator. We first state the requirements of any system that would satisfy our objectives 1) to 5) given above.

1. *Allow the task to execute to completion.* In order to study the effects of a trial execution, we must allow the application to execute to completion. This will ensure that the results of a trial execution match the results of a real execution when the application is actually installed and deployed.
2. *Track the effect of the task on multiple hosts.* During execution, a task may further trigger changes to system objects in other hosts, as given in the CUPS example above. This suggests that any approach that addresses this problem must have support for *distributed monitoring*, thereby tracking and correlations the actions of a program on other hosts on the network.
3. *Support customizable unified logging.* The temporal sequence of operations that result in changes to the objects in various hosts need to logged in a central location, where they can be analyzed. In addition, to focus on events of interest to the system administrator, the logging system must be simple enough to support customizable filters to reduce the size and complexity of evaluating them.

4. *Ability to undo the effects of actions of a program.* This is required to ensure that the system can be restored to its original state before the program was executed.

Below, we discuss the related work by grouping them into various categories. At the end of this section, we discuss the suitability of each approach category in matching the the above requirements.

Logging based approaches A typical approach way to understand the effects of executing a particular software is through the use of logging [2]. The system administrator can enable the logging options present in the software, and then inspect the logs after the operation to have an understanding of the actions of the software. The problem with this approach is that it is completely dependent on the developer of the software system/patch to log its actions. Thus this is is not very dependable option as many software systems are written without logging features. Of course, with experimental software this approach clearly will not work. Also, an approach purely based on logging will make the job of reverting the system back to original state quite tedious, error-prone and in some cases, impossible.

Use of program tracing tools A second approach is to use tools such as *ltrace* [10] and *strace* [4] to study the actions of a piece of software. While this approach may reveal the effects of running or upgrading an application, one sees the effects of the software *after* it has finished execution, when the applications actions have already affected the system. It may be too late, as recovery actions may involve clean-up actions such as restoring files from backups, or removing user-ids created by the application. Approaches such as sandboxing [14, 13, 18, 22, 7, 19] do not work too, as they simply restrict the execution of the software, rather than allowing it to run completely in order to study its actions. Use of package managers such as `RPM` and `dpkg` may simplify the problem of uninstallation; but they do not offer any help in understanding the effects of software that are already installed. Furthermore, package managers are inapplicable if the software is distributed in binary or source forms.

VM based approaches A third approach is to use special machines [16, 12] or even virtual machines [5, 11, 24] for studying the effects of a particular piece of software. In order to correctly track the effects of the system, machines and special hardware have the problem of accurate environment reproduction, where the system configuration on the virtual machine environment needs to accurately reflect the one on the production environment. Such accurate environment reproduction is crucial to ensure that the system behavior on the VM is same as that on the production system. Another possibility is make use of snapshot features in modern virtual machines such as VMware. However, these snapshots tend to give the difference of the actions of the entire set of processes running on the system and not the programs the user wishes to focus on.

Recovery-oriented approaches Although recovery from failures is not the primary goal of our approach, we do provide facilities for recovery in case of a task failure. The Recovery-Oriented Computing (ROC) project [20] is developing techniques for fast recovery from failures, focusing on failures due to operator errors. [8] presents an approach that assists recovery from operator errors in administering a network server, with the specific example of an email server. The recovery capabilities provided by their approach are more general than those provided by ours. The price to be paid for achieving more general recovery capabilities is that their approach is application specific. In contrast, through a DSEE we provide a general task-independent framework for troubleshooting and recovery.

Discussion Note that sandboxing based approaches do not fully support the objective of allowing a task to run to completion (point a) above), as they block actions of a program based on the policy. So using sandboxing, we have no way of learning the complete effects of a piece of software. Logging based systems allow the applications to run with complete freedom, but do not support undoing of actions (point d) above). File versioning systems [17, 21] and virtual machine based snapshot approaches may satisfy undoing at a more general level, but not based on a program or specific actions of a program and therefore do not satisfy point d) above. Furthermore, they do not directly support point b) and c) above. On the other hand, executing a task in a DSEE will address all the objectives a) to d) above.

Paper Organization This paper is organized as follows. In the next section, we discuss the concept of one-way isolation that serves as the basis for our approach for building DSEEs. We then discuss the design details of our framework for building DSEEs followed by the routing enhancements to automatically provide the redirection facility for network operations. We explicate a message handling subsystem that we implemented for communication between various DSEEs. We present a system evaluation by performing various trials using our system and discuss the performance costs followed by a conclusion.

One-way Isolation

Our approach builds on the one-way isolation approach presented in [15, 23]. We briefly review the one-way isolation approach that we employ to create distributed safe execution environments (DSEE).

Isolation of a set of tasks refers to the property that disallows the effects of such tasks from being made available until its completion. In database systems, isolation is one of the ACID properties. The main objective in using isolation in our approach is to effect containment of the trial execution task performed inside the isolated environment. Any operation that is only ‘‘reads’’ the system (i.e., one that reads the

system state but does not write/modify it) may be performed by SEE processes. It also means that ''write'' operations should not be permitted to change the state of the system. There are two options to implement the environment such that isolation is achieved: one is to *restrict* the operation, i.e., disallow its execution. The second option is to *redirect* the operation to a different resource that is invisible outside the safe execution environment. To maintain the correctness of the resource access operations, it is important to maintain the redirection for subsequent operations (such as writes) from the program. Below, we discuss both restriction and redirection for performing system administration trials.

Through *restriction,* an operation initiated by a process is prevented from completion. When this happens, an exception may be returned to the process. To implement restriction, we need to know the set of operations that may affect the state of the system. However, in the context of performing trial executions, an approach purely based on restriction is not likely to be very successful as it will prevent applications from running successfully to completion. For instance, a program may intend to perform a network operation by opening and socket and listening to messages on that socket. If this operation is restricted, this program will not be able to successfully receive messages. Most non-trivial client-server applications will fail for similar reasons. Hence, in our approach we resort to restriction only if the other redirection option is not likely to provide successful results.

The other choice for implementing isolation is through redirection. In *redirection,* any operation that accesses a resource is redirected to another resource that is unavailable to the rest of the system. For instance, when a file modification operation is performed by a SEE process, a copy of the original file may be created in a ''private'' area of the filesystem, and the modification operation is performed on this copy. Redirection does not suffer from the same problem as restriction and the SEE process is likely to successfully run to completion under redirection.

Two forms of redirection are possible: *static* or *dynamic.* Static redirection requires the source and target objects to be specified in advance of the operation, in fact before the SEE process is executed. For instance, one may statically specify that operations to bind a socket to a port p should be redirected to an alternate port p'. Similarly, one may specify that operations to connect to a port p on host h should be redirected to host h' (which may be the same as h) and port p'. However, such static redirection becomes hard to implement when the number of possible targets is too large to be specified in advance or if a SEE process performs a large number of such operations that are distinct. For instance, it may be hard to predict the number and location of files on a server that may be accessed or modified by a client operation. Moreover, such modification operations have indirect side effects that involve dependencies between such object, e.g., the file operations on the server involve changes to the directories these files reside in. A redirection operation that ignores the effect on these directories simply will not work. In such case, *dynamic redirection* where the target for redirection is determined dynamically during execution.

In this paper, by using such redirection, we show how to build *distributed SEEs* (DSEE), where processes executing within SEEs on multiple hosts can communicate with each other. Such distributed SEEs are particularly useful for safe execution of a network server application, whose testing would typically require accesses by nonlocal client applications. (Note, however, that this approach for distributed SEEs works only when all cross-SEE communications take place directly between the SEE processes, and not through other means, e.g., indirect communication through a shared NFS directory.)

In our current implementation, system call interposition is used to implement restriction and static redirection. We restrict all modification operations other than those that involve the file system and the network. In the case of file operations, all accesses to normal files are permitted, but accesses to raw devices and special purpose operations such as mounting file systems are disallowed.

In terms of network operations, we permit any network access that can be dynamically redirected. This entails any local network operation such as a service request from a host in the network. Dynamic redirection is currently supported in our implementation for a number of commonly used network services.

After the trial execution is over, the system administrator can examine the results of the trial execution. If the results are satisfactory, she can commit the results back to the file systems on the respective hosts that run the DSEE. Commit criteria for such executions have been developed in [23]. In this paper, we do not discuss criteria for committing. Instead, our focus is solely on construction of DSEEs and performing system administration experiments with them.

Our Approach

Figure 1 shows the a network-level overview of SUEZ. There are two main components in SUEZ that are responsible for creating a DSEE. They are a) a host level monitor that runs on each SUEZ host and b) a network redirector that runs on the main router. Each host under SUEZ has a *host monitor* component. This host monitor is responsible for isolating any local operation or remote operation. Such host-level isolation component resides on all the other hosts that are similar in the network, and the isolation environments in all these hosts collectively form a DSEE isolation context. The host monitor also runs a messaging service that it uses to communicate with other DSEEs.

The router has a component of SUEZ that performs transparent network level *host and service redirection*. The use of transparent host and service redirection allows the user of the system to run experiments without having to know the network and service requirements of the task to be performed in advance. Each host monitor logs its actions, and these logs are integrated in a log server. The log server presents the temporal sequence of operations performed during the trial execution.

Host Monitor

Figure 2 presents a detailed view of the host monitor. Each host-level monitor is built on top of the isolation module present in [15]. These monitors are used for tracking observable behaviors of programs running on their hosts and tracking changes to file-system state. As shown in Figure 1, similar monitors run of every host used in our system, and communicate with each other for the purposes of logging software actions.

In a typical client-server interaction, an action from a client triggers an action in the server. Hence these monitors communicate with each other to precisely track the commands executed in the server in response to the actions of the client. We therefore have two broad components in a monitor. The first one that addresses isolation of processes running locally under the monitor, corresponding to *host-level* isolation. The other component is for communicating with similar monitors running on other hosts such that *network level* isolation is achieved. This is shown in Figure 2 by the division of host and network level components.

In the reminder of this section, we describe the host monitor.

The objective of our monitoring system is to identify observable events that are triggered by the execution of a program across the entire system administrative boundary. At the level of a host system, this requires us to monitor the observable actions of a set of processes. These actions are ultimately effected through system calls, and hence, *system call interposition* is our primary monitoring approach. Each host level monitor intercepts the system calls of the applications that are running under its purview.

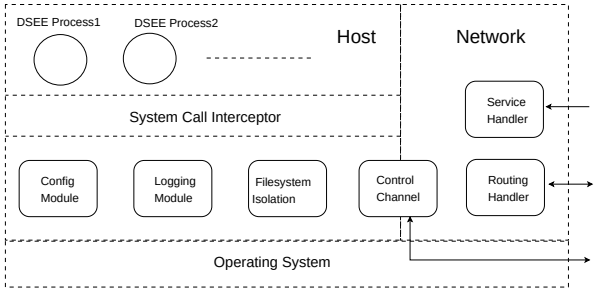

Figure 2: A host view of a DSEE.

The file system module tracks changes made by the software that is run under the DSEE. The file system module is based on our past work on one-way isolation [15, 23]. Isolation is achieved by intercepting and redirecting file modification operations made by the process running on the host so that they access a ‘‘modification cache.’’ This modification cache is invisible to other processes in the system. (This ensures that in the event the system administrator does not like the changes made by the software, it can be safely removed from the system without any side effects.) To ensure a consistent view of system state, the results of file read operations made by the process are modified to incorporate the contents of the modification cache. On termination of the process, the system log contains entries from the modification cache for user to inspect these files to determine if the modifications are acceptable. Otherwise, they can completely undo the changes through the trial execution.

Managing network connections When a process is being monitored, it may make connections to other hosts on the network. Once such a connection is initiated, the *Control Channel Module* (CCM) initiates the monitoring required at the other end of the connection.

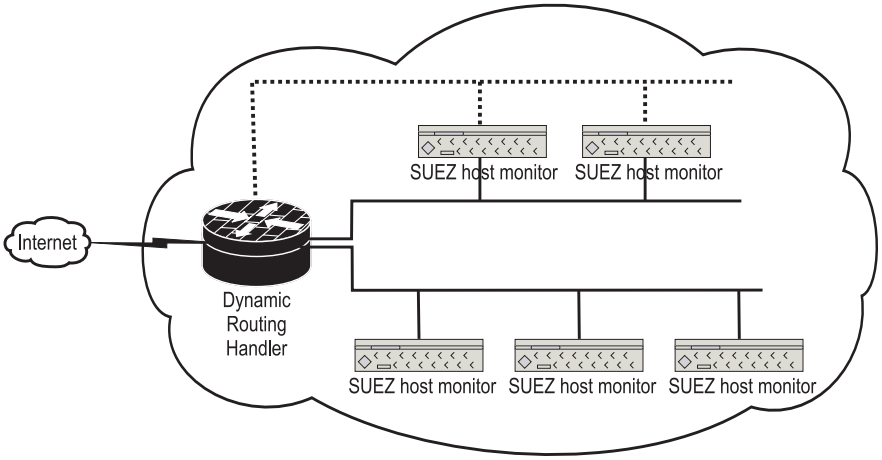

Figure 1: A network view of SUEZ.

Based on the nature of network connectivity (client/server), this module will communicate with its counterpart on the other end of the connection. If the program tries to connect to the network, CCM informs the router of this event which will result in creation of new routing path to the other hosts. There is no global network state stored at a single point for network actions since all other distributed monitors handle them co-operatively. CCM just passes appropriate control messages to the relevant components. We describe the routing module in more detail in the next section.

Dynamic service start/monitoring Recall the CUPS example, where the actions from a browser affect the configuration settings on the print server. In this case, the monitor on the remote host needs to be alerted to monitor the service that receives this request. If the service is not already running and if the SERVICE_UP message is received, then the system allows the service to be started on demand. This is accomplished by using the database of services available in the system. In case the service is already running (i.e., started through the previous step), then the monitor detects this and dynamically attaches itself to the service process. If the service is not already running, it starts the service process.

Log Module The log module generates logs depending on various configuration options and filters. These logs reside on the individual hosts. Using the system call output information itself as the log is not very useful as it may contain excessive information. The log module transforms the system call log information to a more user friendly form. Since the logs can be output can be quite long, customizable filters can be written for the logs to inspect specific actions. For instance, the log can be customized retain information only about filesystem operations and network operations. For filesystem operations, it contains the file object name. For network operations, the service type and address related to the connection is retained. A log generator can be used to merge logs from various hosts to produce unified view logs.

Routing Module

A process that is run may connect to a network service on the local network. Isolation of this operation can be done statically or dynamically. Performing network-level isolation using static redirection requires that the system administrator knows the requirements of the software system that she is experimenting with. Guessing the requirements can soon become tedious or can impact the usability of the approach. Instead, our approach involve dynamic redirection of network service requests. Such dynamic redirection is configurable for specified network services. One question that arises in the same context is that of an application contacting an Internet host. In this case, providing complete isolation while allowing the application to run is not possible, as it is hard to emulate the functioning of an arbitrary network service. In this case, there are two options. One is to disable such requests, for the sake of security. Since our approach is built using system-call interposition, this is feasible. The other option is to only isolate the actions of the client at the host level. Of course, the disadvantage is this option is that reproduction of the entire behavior of the application is not possible, as the server side behavior is not reproduced accurately. This is acceptable as there is generally no easy solution to

```
network-op-isolation-module() {
    switch(new-route){
        case ROUTE-UP:
            client-addr = get-address-of-client();
            target_addr= get-requested-address();
            if (target-addr) already on  network break
         else
            new-host = find-available-host();
            map new-host to  client-addr;
            send new-routing-up message to  new-host;
            get network-portion of the requested address.
            new-device = get-available-devcie();
            boot new--device.
           break;
        case DEL-ROUTE:
            client = get-address-of-host();
            find list of hosts assigned for client.
            send del-routing-path message to the host.
            new-device= get-device-name(routing-path);
            release host resources;
            release network device();
            shutdown device ();
            break;
    }
}
```

Figure 3: Algorithmic sketch of the routing module.

the problem of studying an experimental/untrusted software that tries to connect to an outside host.

Dynamically setting up routes and services requires redirection of network service requests, that are established using *dynamic route generation* and *dynamic service redirection*. We will describe the route generation in this section, and service redirection in the next subsection.

Dynamic route generation is established using a specialized *route handler* module, that dynamically establishes a routing path between the host running the program and the target host. Such dynamic redirection has several possible options – the use of forwarders that do IP masquerading such as IP tables and IP chains. However, if the application specific functionality (such as any internal tables) is dependent on the target IP address, then such forwarding mechanisms may break programs. Open source redirectors are available, however, they do not support every kind of TCP/UDP connection. Also, using a redirector requires the same to be installed on the all the target hosts. The approach we have taken is to modify the routing table dynamically on the router to forward the connections to the target network/host.

To enable redirection of connections, the host needs to configure the IP address of the target host (that runs the network service) dynamically. In our implementation, this is accomplished by establishing a virtual network interface on the target host. This virtual network interface is enabled using IP aliasing.

For a minimal set up for testing client-server implementations, our system needs one router and at least two machines, one that initiates a service request and the other that accepts such requests. (These can be set up in an inexpensive fashion using virtual machines, a topic we will discuss below.) If each of the machines needs to be on a different subnet, then the router should have a network interface on each subnet. Furthermore, IP forwarding needs to be enabled in the kernel state of the router. Our router module is required to be running on the router and on the host accepting service requests. This is needed to change routing tables dynamically.

Let us look at a typical client-server interaction between a client and a web server on our system.

1. A client invokes connection request to the service that either runs on the network or is not yet available.
2. The Control channel module on the client intercepts this event and notifies the routing module (running on the router) of the address for this connection request.
3. Upon receiving this request, the router checks whether there exists an already running web server on the network. If so, it returns and the CCM informs this service-related information to a the service handler on a machine running server. If the service is already running on the server, the client can start exchanging messages. If not, the service handler starts the service. If the network path is not established it proceeds to the next step.
4. The routing module on the machine receiving the message from the router boots up a new virtual network interface with the address.
5. The router chooses appropriate address for a new routing path and boots this new interface.
6. From this point onwards, all communication is transparently redirected through this newly established path between client and server.

The routing module is explained in Figure 3. The state maintained in the router consists of available Ethernet devices and addresses of hosts running. During the initialization of the router module, devices' name need to be given to the module as parameters. When a new host comes up on the network, it registers its address with the router module. The router module maintains a vector of such addresses. Whenever a task is complete, the network interfaces allocated for the routing path are brought down, and the returned to the pool of resources for future use.

Message Handler

Often, the focus of attention on a particular trial execution is in executing one or more features of an application. In this case, a user may want to only focus

```
message-loop() {
    while(true) {
        waitfor-command();
        dispatch-command();
    }
}
dispatch-command {
    case NEW-ROUTE-UP:
        /* set up new route */
        break;
    case DEL-ROUTE:
        /* delete route and release resources */
        break;
    case SERVICE-UP:
        /* bring up the network service */
        break;
    case SERVICE-DOWN:
        /* shutdown network service */
        break;
    case NEW-HOST-UP:
        /* add host info to host list */
        break;
    case QUERY-HOST:
        Query host list ;
        break;
    case START-TRACING:
        /* start recording operations */
        break;
    case STOP-TRACING:
        /* stop recording operations */
        break;
}
```

Figure 4: Various messages received by the message handler.

on this operation and ignore other operations of the system. A message handler is made available on each client to start and stop tracing the operations made by the trial execution. A typical use scenario is as follows: When the user would like to focus on exercising a feature in the application, before exercising this feature, she can instruct the client DSEE to send a START_TRACING message. All the DSEEs will record the subsequent operations made by the task. After the user is done, she can send a STOP_TRACING message that will stop recording the operations of the task. When tracing is stopped, the set of actions that were recorded between the START_TRACING and STOP_TRACING messages capture observable effects of the operations in this window.

Additionally, the message handler also deals with messages from other DSEE components. These messages are about routing information and services registration. On receiving these messages, the message handler invokes the appropriate handlers. The responses to messages received are shown in the commands exercised by the message handler in Figure 4. For example, when it gets NEW-ROUTE-UP or DEL-ROUTE messages, it invokes routing module to boot up or shutdown routing paths respectively.

If the application running in the DSEE is untrusted, it may send false messages to the message handlers on the other hosts. For this purpose, the default policy enforced by the system call interceptor is to disallow any such messages on the control channel that is maintained by the host monitors.

Support for virtual machine hosts Virtual machines can result in creation of inexpensive hosts on demand, and our approach is designed to take advantage of the use of virtual machines. Our prototype implementation uses VMware virtual machines [5], where creation/loading of virtual network interface and virtual network groups can be easily done on demand.

Experimental Evaluation

Before describing the experiments performed with SUEZ, we describe our experimental set up. We also describe the configuration options available to the user.

Setup

- *Virtual network setup* The network set up has one router and two subnets. Since we used VMware to create hosts on the network, this required creation of three virtual machines.
- *Router setup* To act as a router, the kernel value for IP_FORWARD should be 1. This router has three network interfaces, one on the physical network, and the other two for subnets A (192.168.1.X) and B (192.168.2.X).
- *Message handler setup* The message handler on the router is set up with available device names and addresses. Above case, available device on subnet B is bound to 192.168.2.1.
- *Server Host monitor setup* A SUEZ host monitor (with its associated message handler) is launched on a machine on subnet B to act as host available for service. To this, HOSTMODE value need to be set in the config file. At the starting of this host information will be sent to the message handler.
- *Client Host setup* A SUEZ monitor with ROUTEMODE value set in the appropriate config file for the a machine on subnet A. ROUTEMODE config variable is explained below.

From this point onwards, if the client program tries to connect to a service, with SUEZ with ROUTEMODE set, the connection will be transparently forwarded to host in subnet B.

Configuration Parameters

The following configuration flags need to be set on the hosts in the network.

1. ROUTEMODE – If this value is set, SUEZ will intercept all network connections before the client program get connected to its original destination. Eventually, the connection will be transparently forwarded to a machine that hosts the corresponding service.
2. HOSTMODE – To automatically configure an ip address and start the required service dynamically as on host, one would set this value in SUEZ. If this flag is set, the host monitor in SUEZ will send host address to the appropriate message handler.
3. REMOTELOG(ULOG) – In order to make a unified log, each host monitor traces and collects the events of interest. If this value is set in config file, each event of interest will be logged. When these events are merged into to one log, only events of interest will be made viewable in the unified log.

In addition, a list of available devices available to setup new routing paths on the router is provided as input to the router module through a config file.

In the following section, we present an experimental evaluation of using our approach. Our evaluation consists of two parts: the first is a *system evaluation*, which was about applying the system to study the execution of several system and application software tools. The second part is a performance evaluation of our system.

We analyzed the installation and execution of several applications in DSEEs created using our system. Below we describe four candidates from our experiments.

Address Book leak in SquirrelMail Squirrelmail is a Mail User Agent (MUA) package written in PHP4. Being a web based user agent, it interacts with a webserver in addition to a mail server. The functionality of the program is triggered through many links and buttons

on the web page interface. For SquirrelMail, since the interface is web-based, we tried to understand the functionality that interacts solely with the web server, as opposed to that which also interacts with the mail server. Understanding the nature of information stored in a web server is critical as the protection of data stored in a web server is an important issue. So we installed Squirelmail with its default configuration in a DSEE, and observed the actions during installation.

After the installation, we tested the various options in Squirrelmail by trying out the various options in the web interface. One such interface is the address book interface that allows a user to add or remove entries from his address book. Once that interface was tried, the results of the system pointed to file modifications on the web server. We observed that the default configuration resulted in placing the `data` subdirectory that holds the address book information under the top-level Squirrelmail directory. If this URL is known, an arbitrary user can access the (private) address book information of any other user. The URL is normally known to any user of the system, and is easily guessable if one knows the presence of a Squirrelmail installation on a server. This directory needs to be protected from being directly accessible in order to protect the privacy address book information.

Our tool enabled us to correlate the action of creating an address book entry on the client to the location that it was stored in the server and therefore uncover the vulnerability of address book information leak. Changing the access permissions for the directory subsequently solved this problem.

Remote web server upgrading Several systems exist that perform upgrades/installation from a remote machine. For instance, Webmin [9] is one such tool. The primary purpose of such tools is to simplify desktop administration. Although this purpose is achieved by such tools, they do not provide a way to recover from any problems during installation. For instance, if the installation of a package using a remote administration tool is not successful it is difficult to recover from such errors. Configuration files that are overwritten may be lost during the installation process. (Using a backup procedure, the system administrator can save these files, but this requires knowing in advance which files are being overwritten). Using our approach, we can perform the installation without the fear of damaging the system in any way, and then finally inspect the system to see if the changes made by the installation are desirable, and then commit these changes. If the installation does not proceed as expected, they can go back to the original state of the system.

To study the use of the Webmin administration tool, we upgraded the apache web server program from a remote client machine (in a DSEE). We upgraded the http package (that contains the apache web server) version 2.0.55 to http-2.0.58 through the Webmin tool remotely. After the installation, we tested whether the installation process worked fine by testing the new version of the web server. Also, we observed for any modifications to existing files. In both cases, there were no problems resulting from the installation. Hence these changes were successfully committed into our system.

Debugging Mgboard Configuration Mgboard [6] is web-based message board on apache with php. Mgboard uses an internal flat-file database rather than an external SQL database such as mySQL. Using Mgboard, a user can post articles and upload files to the website. In interactive programs such as Mgboard, it would be a tedious task to identify misconfigurations. For example, the files that store system configuration data for Mgboard needs to be group-writable and other when create database file (for public writing). As server-side scripts are hard to debug, any misconfigurations in Mgboard (which is indirectly powered by the Phpadmin program) are hard to detect. Also it is difficult to know which actions (such as addition/update of posts) are affected by this misconfiguration. However with the use of a DSEE, it is easy to know which actions were triggered (specifically, which files were executed) and thereby reason through the CGI script operations.

To check this, we performed two experiments. In the first experiment, we intentionally planted certain misconfigurations in the remote client, by introducing file permission errors. In the second experiment, using a web client, we created a new page on the web server. While posting an article from the client on to the server, using our tool, we observed creation of temporary files in the `/tmp` directory of the server. This helped us to investigate the possibility of a local race condition vulnerability that could result through the creation of this temporary file. Such a race condition could happen if arbitrary users can overwrite this file. Such reasoning is possible with our system as the unified log presents the temporal sequence of such actions and writing custom filters can identify and 'zoom in' on the error.

Configuration errors in Proftpd ProFTPD [3] is a ftp server written for use on a UNIX-like operating system. A ftp server allows the remote user to perform operations on remote file systems, and even send site-specific commands. It is therefore important to test the software installation and check for the potential actions of the server when it interacts with a client. Occasionally the settings may dictate account users or even anonymous users can inquire or change file systems explicitly by using remotely issued commands.

We tried this as a candidate example for testing the ProFTPD server. We observed that on installation, the system executes an init script, which results in two main actions: creation of user and group ids for the server. Another configuration file that was created was the `/usr/local/etc/proftpd.conf`. Also during our installation, the server was configured to restrict users' access

with DefaultRoot in proftpd.conf but an accidental system configuration error resulted in the option #Default Root . When the service was started, we exercised the gftp client to change the working directory to the root directory and store a file. The usual expectation on part of the server administrator was to have the file stored in user home root, but due to this misconfiguration it triggered a permission error. In this case, the unified view log shows actual action sequence with operations on the file system starting from operations on the client to the server. It pinpointed the source of the error to the DefaltRoot option. Such configuration errors can be debugged effectively with our approach, which allows one to focus specifically on the results of a particular action in a program.

Performance Evaluation

The second part of our evaluation is the of the performance of our system with several client-server applications.

We describe the experimental setup first. We used several virtual machines hosted in machine for all our experiments. The machine is an AMD Sempron 2400+ CPU with 2 GBytes memory, running the Red Hat Enterprise Linux distribution. The virtual machines also run the RedHat Enterprise 4. Each virtual machine instance runs SUEZ with router and service handler, and any associated client or server programs.

We classify the performance measurement experiments into three categories:

- *Client-side overheads.* These overheads in the client side may result from the monitoring overhead through SUEZ.
- *Server side overheads.* These overheads result from the monitoring overhead at the server.
- *Network delays.* These are overheads introduced due to the routing and service isolation in SUEZ.
- *Service and program launching overheads*. Since the service redirection and service program startup are done on demand, this may introduce additional overheads during the start of a session.

We discuss all the overheads in detail below.

Client-side overheads We recall that our system uses system call interposition at the client to track the actions of the client, and any possible communication messages to the server. Such interception facilities are implemented using *ptrace* mechanisms available in the Linux operating system. We have measured the performance as a ratio of the combined system and user time and compared them for the following situations: a) without any monitoring b) with the use of our monitoring mechanism. Figure 5 shows the performance overheads at the client. The performance numbers show overall execution times with and without the monitor.

In the figure, we have measured the performance of four desktop clients while performing the experiments mentioned in the previous section. The results show that the overhead due to system call interposition vary for various clients ranging from 68 to 325%. The difference in overhead is due to the frequency of system calls invoked by these different clients. In addition, an entirely a user-level mechanism such as *ptrace* suffers from moderate to high overheads [15]. These high overheads due to the context switching associated with the process that performs the monitoring. A kernel level mechanisms typically has overheads in the range of 10-15% as evidenced by earlier work on kernel level mechanisms for one-way isolation [23].

Server-side overheads For servers, we measure the overheads differently. Since most servers continue to run even after servicing client requests, it is not possible to measure the overheads in a manner similar to that of

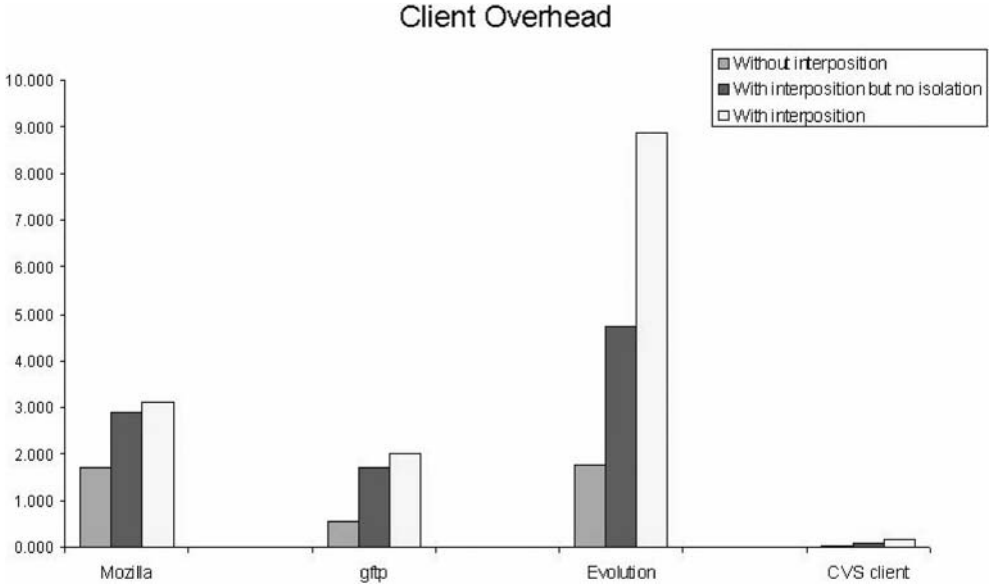

Figure 5: Client-side overheads.

the clients. To measure the overheads on the server, we have measured the mean response time of the server at each client. Recall that our system monitors the system calls made from the client, and on a connect system call, sets up routing paths and starts the corresponding services. Therefore, the response time is measured (at the client) as the difference from first connect system call to first or last recv call on each client's log.

Figure 6 shows the mean response times at the client. This response time indicates the steady state overhead, i.e., the overhead without the following two causes 1) any (one-time) routing overhead in the computation of the virtual routing path and 2) overheads from automatic start of the corresponding network service. As shown in the figure, the response times for various server programs are within a factor of two. For trial installation purposes we consider such overheads acceptable. Moreover, a kernel level patch to the isolation mechanism will reduce the response time overhead.

Route Computation Overhead

In order to create a dynamic routing path, new network interfaces are needed to be initialized on the router and the server host respectively. Delays are introduced at the router (in the control channel module implementation) to find any available hosts for assignment of new IP address.

We compute the overheads for CUPS, Webmin, Proftpd, Sendmail and CVS. overhead also as the difference between mean server response time at the client, with and without route computation. In order to measure this overhead introduced by the dynamic route computation and the service handler, we obtained the following measurements of the programs used in our experiment. These are a) the relative mean response time without the system, b) with the use of isolation but not using routing and service function on servers and c) with host-level isolation and the use of dynamic routing and service handling. The performance for all the seven applications that were used in the server side overhead experiments (described above) were measured. The time stamps on the client were recorded in the log for each network and file related operation. For web based programs, the mean time difference from first connect to first recv system call was measured. For sendmail, the difference between the first connect to last socket write was measured. For proftpd the difference between the first and second connect system calls made (the first call is made for getting the data channel for the file transfer). For CVS, the mean time to create the .cvspass was measured.

We measured the delays introduced due to the routing process. This delay does not depend on the specific application that was used. We measured this delay as a average delay as perceived by the client. The mean delay introduced due to the router (as perceived by the client) was measured to be 0.125 seconds.

Routing and Service Launching Overhead

Once a host receives a request for a service, it needs to start the service and subsequently the server responds to the request. We measured the service launching time for each of the server applications tested. We measured this as an average delay perceived by each client. Also, to avoid routing delays from entering the picture, we set static routes from the client to the server. These are the delays for the server applications: sendmail (3.8 s), apache (3.8 s), Proftp (2.2 s), and Webmin (1.6 s).

We note that services can be started using inetd, and may not result in overheads when the client contacts the server host for the first time. Such service programs can be traced dynamically (i.e., attached to the monitoring process). This will result in much lesser overheads. We also note that at the time of interception of the original network operation, the route is created and the service is launched transparently before the actual connection request from the client is sent.

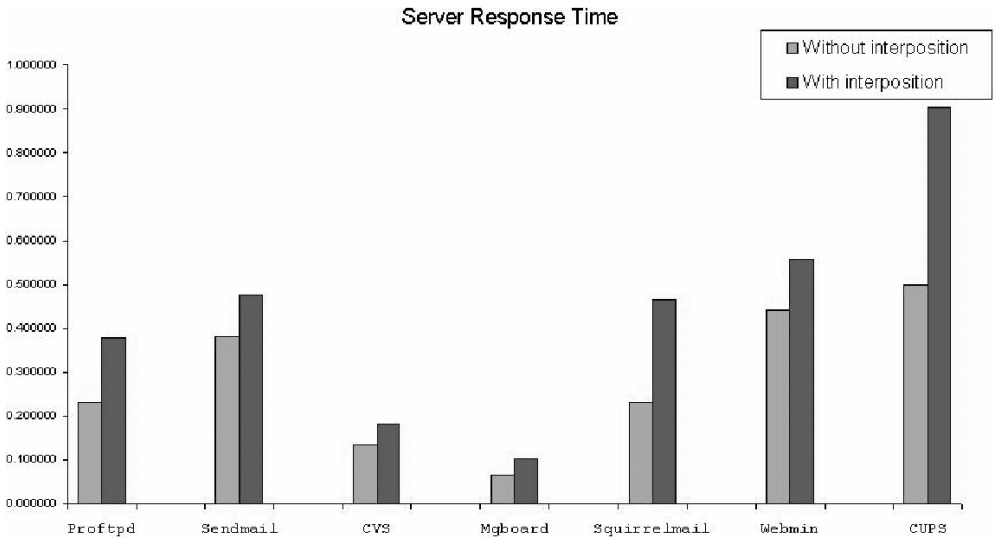

Figure 6: Server response times.

Conclusion

A set of questions that the system administrator typically has when performing any trial execution are:

- During the trial experiment, does this piece of software cause conflicts with other packages such as overwriting configuration files?
- After installation of a package, does it work cooperatively well with existing software within the entire network?
- Does any of the features of a piece of software malfunction, even though it may seem to work well apparently?
- Is it safely deployable in the network? Does it violate the network privacy and security policies? Does it install files in hidden locations?

The system we describe in this paper, called SUEZ, is designed to support assist a system administrator in answering these questions. To achieve this our system employs one-way isolation of local and remote operations inside a distributed safe execution environment. In addition to satisfying main goal of providing support for study and experimentation with software, our approach has numerous other benefits. It requires no access to source code of the applications that need to be studied; it is cost-effective in being able to utilize virtual machine technology for dynamically configuring hosts and routes, and customizable to various situations that one may encounter in system administration practice. We believe that our approach has the potential to be applicable in several day to operations involving system trials, reverse engineering and troubleshooting.

Acknowledgments

We thank Zhenkai Liang, Weiqin Sun and R. Sekar for many discussions about the implementation of distributed isolation operations that formed the basis for writing this paper. We also thank our shepherd Narayan Desai and the anonymous referees for reading our text and providing many useful suggestions that have improved the contents this paper. Finally, we acknowledge Rob Kolstad's help with typesetting of the manuscript.

Author Biographies

Doo San Sim is a graduate student in Computer Science at University of Illinois at Chicago. His research interests are in computer security, mainly in addressing security in software installations. He can be reached by email at dsim2@uic.edu .

Dr. V. N. Venkatakrishnan is an Assistant Professor of Computer Science at the University of Illinois at Chicago. He is currently co-director of the Center for Research and Instruction in Technologies for Electronic Security (rites.uic.edu). His main research area is computer and network security. Specific research areas include malware detection, software security and personal information privacy. He is available by email at venkat@cs.uic.edu .

Bibliography

[1] *Common UNIX printing system*, http://www.cups.org .

[2] *Controls the system log*, Man pages.

[3] *Professional FTP*, http://www.proftpd.org .

[4] *Strace*, http://www.liacs.nl/~wichert/strace .

[5] *Vmware*, http://www.vmware.com .

[6] *A web board not using Sqldb*, http://www.phpschool.com .

[7] Acharya, A., and M. Raje, "Mapbox: Using parameterized behavior classes to confine applications," *USENIX Security Symposium*, 2000.

[8] Brown, A. and D. Patterson, "Undo for operators: Building an undoable e-mail store," *USENIX Annual Technical Conference*, 2003.

[9] Cameron, J., *A web-based interface for system administration for UNIX*, http://www.webmin.com .

[10] Cespedes, J., *A library call tracer*, Man pages.

[11] Chen, P. M. and B. D. Nobl, "When virtual is better than real," *Proceedings of Workshop on Hot Topics in Operating Systems*, 2001.

[12] Chiueh, T., H. Sankaran, and A. Neogi, "Spout: A transparent distributed execution engine for java applets," *International Conference on Distributed Computing Systems (ICDCS)*, 2000.

[13] Dan, A., A. Mohindra, R. Ramaswami, and D. Sitaram,' *Chakravyuha: A sandbox operating system for the controlled execution of alien code*, Technical report, IBM T.J. Watson research center, 1997.

[14] Goldberg, I., D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: confining the wily hacker," *USENIX Security Symposium*, 1996.

[15] Liang, Z., V. Venkatakrishnan, and R. Sekar, "Isolated program execution: An application transparent approach for execution of untrusted programs," *ACSA Computer Applications Security Conference (ACSAC)*, Las Vegas, December, 2003.

[16] Malkhi, D. and M. K. Reiter, "Secure execution of java applets using a remote playground," *Software Engineering*, Vol. 26, Num. 12, 2000.

[17] Muniswamy-Reddy, K.-K., C. P. Wright, A. P. Himmer, and E. Zadok, "A versatile and user-oriented versioning file system," *Proceedings of USENIX Conference on File and Storage Technologies*, 2004.

[18] Prevelakis, V. and D. Spinellis, "Sandboxing applications," *Proceedings of Usenix Annual Technical Conference: FREENIX Track*, 2001.

[19] Provos, N., "Improving host security with system call policies," 2002.

[20] *Recovery-oriented computing,* http://roc.cs.berkeley.edu .

[21] Santry, D. J., M. J. Feeley, N. C. Hutchinson, and A. C. Veitch, ‘‘Elephant: The file system that never forgets,’’ *Proceedings of Workshop on Hot Topics in Operating Systems*, 1999.

[22] Scott, K. and J. Davidson, ‘‘Safe virtual execution using software dynamic translation,’’ *Proceedings of Annual Computer Security Applications Conference*, 2002.

[23] Sun, W., Z. Liang, V. N. Venkatakrishnan, and R. Sekar, ‘‘One-way isolation: An effective approach for realizing safe execution environments,’’ *NDSS*, 2005.

[24] Whitaker, A., M. Shaw, and S. Gribble, ‘‘Denali: Lightweight virtual machines for distributed and networked applications,’’ *Proceedings of USENIX Annual Technical Conference*, 2002.

[25] Yu, Y., F. Guo, S. Nanda, L. Lam, and T. Chiueh, ‘‘A feather-weight virtual machine for windows applications,’’ *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE’06)*, 2006.