Abstract

An **intrusion detection system** (**IDS**) is a device or software application that monitors network or system activities for malicious activities or policy violations and produces reports to a Management Station. All Intrusion Detection Systems use one of two detection techniques:

- Statistical anomaly-based IDS
- Signature based IDS

Statistical anomaly-based IDS are based on modeling program behaviors (in terms of ports used, which devices generally connect to each other, **system calls it makes**) and alert the administrator when there is an anomaly. In this project I deal with system calls modelling.
A weakness of these techniques is that they focus on control flows involving system calls (the order in which calls are made), but not their arguments (data flows). This weakness makes them susceptible to several classes of attacks, including attacks on security-critical data, race-condition and symbolic link attacks, and mimicry attacks.

The main objective in my second phase of the project is to develop an approach to learn the data flow behaviors of the program. The advantage with this approach is that it learns the temporal properties of the arguments (arguments that change with time). This contrasts with previous system-call argument learning techniques that did not leverage control-flow information, and moreover, were focused on learning statistical properties of individual system call arguments.

A sample attack is done by exploiting the "***symlink vulnerability"*** in ***setuid programs*** is demonstrated and the how our approach detects it by building relationships among the system call arguments is shown.

TABLE OF CONTENTS

1. Introduction

1.1 Need for the system

Many host-based anomaly detectors have been proposed to monitor system calls. Some of these detectors monitor the sequence of system calls emitted by the application and utilize the control-flow

information of the system calls to detect intrusions. By the term "*control-flow information*" we mean the observed program's behaviour (sequence of system calls) during the training phase where we assume it is free from attacks.

Control-flow-based detectors have been shown to be effective in detecting detecting intrusions, eg., code-injection attacks, because many such intrusions change the control flow of the program to make additional system calls. However, an important weakness of the above approaches is their singular focus on control flows,with little emphasis on data flows involving system call arguments. This makes them susceptible to several classes of attacks like:

- Race Condition Attacks
- Attacks on File Descriptors
- Symlink Vulnerabilities

Hence we propose a system which not only monitors the control-flow, but also monitor the arguments to the system calls and use the data-flow information to detect attacks.

1.2 Objective

The objective of this project is to build a prototype that builds a model by capturing the system calls and its arguments that are executed by any application during normal execution (training phase) and another prototype which takes the model as input and checks for violations or deviations (if any) from this model.

We make use of ***ptrace*** system call in linux to capture the system calls and examine the register values for capturing the system arguments. The main objective is to build the unary and binary relationships among the system call arguments.

2. Literature Survey/ Background study

2.1 Existing system

Since the late 1990's there has been extensive research in the field of Intrusion Detection Systems (abbreviated as IDS) based on identifying the anomalies in the application behaviour. These Statistical Anomaly-based IDS working can be described in the following steps:

1. Building profiles of normal behaviours (training phase)
2. Using the profiles to determine the significant deviations.

These systems proved to be less effective when attacks were crafted keeping these IDS in mind.

Attacks like symlink vulnerabilities, race condition attacks calls the same system calls and hence these systems fail to identify them. Here the anomaly is not in the system calls made but it lies in the data passed to the system and thus it lies in the data-flow across the various system calls. Hence we design a prototype which makes use of data-flow across the system calls and tries to find the anomalies in the system.

2.2 Motivation for a new method

To detect the attacks mentioned above, it is necessary to reason about system call arguments. Research in this direction has so far been focused on learning statistical properties of each system call argument in isolation. In contrast, we present an intrusion detection technique that is based on learning temporal properties involving arguments of different system calls, thus capturing the flow of security sensitive data through the program.

Specifically, we try to formulate a new notion of dataflow properties of programs. Since they are defined in terms of externally observable events (by events we refer to system calls), we do not try to formulate the actual data flow that takes place inside a program during runtime. Instead we try to formulate relationships observed between the parameters of different system calls.

Our formulation is decoupled from control-flow models, but can leverage control-flow context to significantly improve the precision of dataflow models. Dataflow properties are represented as relationships among the system call arguments.

2.3 Establishing relationships

We implement two kinds of relationships which represent the data flow across the system calls:

- Unary relationships: Unary relations that involve properties of a single system call argument.
- Binary relationships: Binary relations that involve arguments of two different system calls.

Binary relations turn out to be powerful and versatile. They enable models to be parameterized, e.g.,they can capture how a system call argument is derived from a command-line parameter or an environment variable. They enable detection of several recently reported stealthy attacks that would be undetectable without them. Binary relations also support formal reasoning about non trivial security properties of programs.

2.4 Advantages

- The system can be This approach can be layered over existing techniques for learning

control-flows thus enhancing the accuracy of the host based-intrusion detection models.

- Such a system which works on whitelisting is proved to be more effective against zero-days as it does anomaly detection.
- The overhead caused is not very high.

Ptrace

Introduction

ptrace() is a system call which provides a means to observe and control the execution of another process and also examine the changes in the core image and registers. It is primarily used to implement breakpoint debugging and ***system call tracing***.

Usage

```
long ptrace(enum __ptrace_request request,
            pid_t pid,
            void *addr,
            void *data)
```

General Execution flow

The parent can initiate a trace by calling fork system call and having the resulting child do a PTRACE_TRACEME, followed (here) by an exec. Alternatively, the parent may commence trace of an existing process using PTRACE_ATTACH.

While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. (The exception is SIGKILL, which has its usual effect.) The parent will be notified at its next wait and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

When the parent is finished tracing, it can terminate the child with PTRACE_KILL or cause it

to continue executing in a normal, untraced mode via PTRACE_DETACH.

The values of request used in this implementation are:

- PTRACE_TRACEME
- PTRACE_PEEKUSER
- PTRACE_GETREGS
- PTRACE_SYSCALL

Pros and Cons of Ptrace:

Pros**:**

- Works with all the *nix kernels (including OpenBSD, Solaris) with ptrace support.

Cons:

- It has a performance overhead. Hence applications run a bit slower.
- Some "addresses" are processor architecture dependent.
- The interface is not clean (some signals cannot be used, SIGSTOP/SIGCONT, it overrides the natural semantics of the wait system call).

Note: Even though the cons are more in number, the most popular debugger in Linux GDB (The GNU debugger) is implemented using ptrace. Which shows the overhead is not high.

3. System Study

3.1 Proposed System

The proposed system is as follows:

- The initial step is modelling the normal work flow.
- For this we have a parent process which forks a child and populates it with the executable which we are going to model.
- In the child process we trace the processes register values and system calls using ptrace.
- For each system call it executes we capture and build relationships among the system call arguments capturing the system calls flow and store it in a file which is the model.
- This step is considered to be free from attacks.
- Further usage of that executable must be done by the validation process which works similar to the above but is provided an extra argument which is the model file.
- Thus it checks the model and any anomalies are reported and the process will be forced to exit.
- Thus the system shows how anomalies in the data flow can prove to be a potential area which can be used to catch stealthy attacks on file descriptors.

3.2 Hardware Specification

There is no hardware specification involved in this project as the whole implementation is a terminal based (CLI) program.

3.3 Software Specification

The implementation is done in Ubuntu 11.04 (Natty Narwhal) with its native kernel.

- Operating System: Ubuntu 11.04
- Kernel: 2.6.38-8-Generic
- Languages: C (gnu-gcc)
- Libraries: Standard libraries in C, libraries for system programming like (ptrace, user.h,

unistd.h)

4. IMPLEMENTATION DETAILS

4.1 Defining Data Flow Behavior:

4.1.1 Events, Traces and Behavior Model

We formalize program behaviors in terms of externally observable events generated by a program. Since our interest is mainly confined to system call events, we will use the terms "event" and "system call" interchangeably. We begin with a series of definitions:

- An execution trace (or simply, a trace) for a program P, denoted T(P), is the sequence of all the system calls executed by P during its execution. A trace typically includes information about system call arguments, and/or information about the program's runtime environment,

  e.g., the program location from where a system call is made ("PC" information).

- A system call tracer (or simply, a tracer) is responsible for intercepting and recording system calls made by P, thus generating a trace of system calls and its related information T(P) and a behavior model which consists of the information related to the system call arguments.

Thus we capture the following data:

a. All-trace properties: Properties which hold for every trace.
b. Sequencing relationships among events: This is what we meant by ***Control-flow information***.

Besides the above we capture the ***Data-flow properties.***

4.2 Data-flow properties:

By dataflow properties, we still refer to properties of execution traces, but these properties relate to the values of event argument data and their flow from one system call to a subsequent one.

4.3 Making control-flow context available for learning

We use the term control-flow context to refer to the event context information that can be provided by a control-flow model. We use labeled traces to encode control-flow context into traces. The label here is the counter which actually tells the number of the system call.

Example: If a file operation happens in a loop (for the sake of simplicity we do not mean a buffered write) each system call write is followed by a number (f9 refers to file-9)

4.4. Possible Dataflow Relationships:

Our focus is on learning all-trace dataflow properties. These properties will be formulated as relationships on event arguments. It is natural to specify these relationships by referring to argument names in a labeled trace. We limit our attention to unary and binary relations on event arguments, as they can be learnt more efficiently, and seem adequate for our purposes.

Unary Relations: Unary relations capture properties of a single argument. They can all be represented using the form X R c, where X is an argument name, R denotes a relation, and c is a constant value. Examples of unary relations include:

- **Equals** relationship is applicable to all types of arguments. For instance X equal v indicates that the value of argument X is always v.
- **Element of** relationship is used to capture the fact that an argument can take one of several values. X ***element of*** S, indicates that X can take any of the values in the set S.
- **Is sym_link relationship** is used to capture the fact that an argument is a file or is a symbolic link to a file. Since symbolic links pose a definite threat to almost all of the setuid programs we include this.
- **subsetOf** is a generalization of elementOf, and is used when an argument can take multiple values, all of which are drawn from a set. For instance, M ***subsetOf*** {RD, WR} represents the fact that M is a set-valued argument whose value is a subset of the set {RD, WR}.

Binary Relations: Binary relations capture relationships between two event arguments. These may be arguments of the same event, or arguments of different events. Our focus is mainly on the latter, since such relationships naturally capture the flow of data from the arguments of one system call to another.

- **Equals** captures equality between system call operands,

  e.g., the file descriptor returned by an open operation equals the first argument of a subsequent write operation.

An Example program to illustrate the modeling procedure:

```c
int main (int argc, char * argv []) {
struct stat st;
FILE * fp;
if (argc != 3) {
        fprintf (stderr, "usage : %s file message\n", argv [0]);
        exit(EXIT_FAILURE);
}
if (stat (argv [1], & st) < 0) {
        fprintf (stderr, "can't find %s\n", argv [1]);
        exit(EXIT_FAILURE);
}
if (st . st_uid != getuid ()) {
        fprintf (stderr, "not the owner of %s \n", argv [1]);
        exit(EXIT_FAILURE);
}
if (! S_ISREG (st . st_mode)) {
        fprintf (stderr, "%s is not a normal file\n", argv[1]);
        exit(EXIT_FAILURE);
}
// this sleep creates a time gap to change the file and create a symbolic link
sleep (10);
if ((fp = fopen (argv [1], "w")) == NULL) {
        fprintf (stderr, "Can't open\n");
        exit(EXIT_FAILURE);
}
fprintf (fp, "%s\n", argv [2]);
fclose (fp);
fprintf (stderr, "Write Ok\n");
exit(EXIT_SUCCESS);
}
```

We

We illustrate the concepts developed above with an example program that is a highly simplified version of the tar program. Figure 1 shows this simpletar program, which takes a command-line argument describing the source directory, and another command-line argument that specifies the name of the archive. It traverses the directory, which may contain subdirectories, and copies all the files into the archive. For simplicity, we ignore many aspects of archiving such as maintaining file boundaries, directory structures, and so on. In addition, we have abstracted away some details such as the use of lstat system call, and replaced them with more descriptive names such as isdirectory. In the example, all system calls are underlined.

States in the model are labeled with 'n' where 'n' is the system call number. Transitions are labeled with system call and argument information. Argument names are based on the nature of the argument and the transition they are associated with, e.g., a file descriptor argument to the close system call made on the transition from state call '14' to call-'8' is labeled as FD14, while FD`14 refers to the same argument when the transition is from call-'14' to call-'18'. Relationships are shown as annotations on the transitions.

To illustrate how some of these relationships are learnt, we consider an execution trace generated when the program is run to archive the directory ***/opt/proj*** into a tarball ***/tmp/proj.tar***. The operations of the trace are shown in the first column of Figure 2. The second column shows corresponding control-flow transitions learnt in the FSA. The third column shows values of system call arguments, and the fourth column shows some of the relationships learnt. In the generation of the trace, we have introduced a synthetic event start to capture command line arguments, and converted all the files names into absolute path names. Some of the relationships, such as those capturing absolute values of the file descriptors, are not shown in the above example. In addition, absolute values of various file arguments will be learnt from the above trace, but these are not shown

A sample trace of the program in Figure 1 and the observed argument

relationships:

Operation Traces	Control-Flow Transition	Argument Values	Satisfied Data-Flow Property
Program started with arguments "/opt/proj","/tmp/proj.tar"		$\{I$ = "/opt/proj", O = "/tmp/proj.tar"$\}$	
ℓ_3 : open("/tmp/proj.tar", WR)=3	$start \rightarrow \ell_3$	$\{F_3$ = "/tmp/proj.tar", M_3 = WR, $FD3 = 3\}$	F_3 **equal** O, M_3 **elementOf** {WR}
ℓ_6 : opendir("/opt/proj")	$\ell_3 \rightarrow \ell_6$	$\{F_6$ = "/opt/proj"$\}$	F_6 **isWithinDir** I
ℓ_8 : isdirectory("/opt/proj/README")	$\ell_6 \rightarrow \ell_8$	$\{F_8$ = "/opt/proj/README"$\}$	F_8 **isWithinDir** F_6
ℓ_{11} : open("/opt/proj/README", RD)=4	$\ell_8 \rightarrow \ell_{11}$	$\{F_{11}$ = "/opt/proj/README", M_{11} = RD, $FD_{11} = 4\}$	F_{11} **equal** F_8 M_{11} **elementOf** {RD}
ℓ_{12} : read(4)	$\ell_{11} \rightarrow \ell_{12}$	$\{FD_{12} = 4\}$	FD_{12} **equal** FD_{11}
ℓ_{13} : write(3)	$\ell_{12} \rightarrow \ell_{13}$	$\{FD_{13} = 3\}$	FD_{13} **equal** FD_3
ℓ_{14} : close(4)	$\ell_{13} \rightarrow \ell_{14}$	$\{FD_{14} = 4\}$	FD_{14} **equal** FD_{11}
ℓ_8 : isdirectory("/opt/proj/src")	$\ell_{14} \rightarrow \ell_8$	$\{F'_8$ = "/opt/proj/src"$\}$	F'_8 **isWithinDir** F_6
ℓ_6 : opendir("/opt/proj/src")	$\ell_8 \rightarrow \ell_6$	$\{F_6$ = "/opt/proj/src"$\}$	F_6 **isWithinDir** I
ℓ_8 : isdirectory("/opt/proj/src/a.c")	$\ell_6 \rightarrow \ell_8$	$\{F_8$ = "/opt/proj/src/a.c"$\}$	F_8 **isWithinDir** F_6
ℓ_{11} : open("/opt/proj/src/a.c", RD)=4	$\ell_8 \rightarrow \ell_{11}$	$\{F_{11}$ = "/opt/proj/src/a.c", M_{11} = RD, $FD_{11} = 4\}$	F_{11} **equal** F_8 M_{11} **elementOf** {RD}
ℓ_{12} : read(4)	$\ell_{11} \rightarrow \ell_{12}$	$\{FD_{12} = 4\}$	FD_{12} **equal** FD_{11}
ℓ_{13} : write(3)	$\ell_{12} \rightarrow \ell_{13}$	$\{FD_{13} = 3\}$	FD_{13} **equal** FD_3
ℓ_{14} : close(4)	$\ell_{13} \rightarrow \ell_{14}$	$\{FD'_{14} = 4\}$	FD'_{14} **equal** FD_{11}
ℓ_{18} : close(3)	$\ell_{14} \rightarrow \ell_{18}$	$\{FD_{18} = 3\}$	FD_{18} **equal** FD_3
ℓ_{19} : exit(0)	$\ell_{18} \rightarrow \ell_{19}$		

Detection of attacks

Note that in principle, a range of attacks are detectable by our approach. Even though we can test for the various exploits from CVE (Common Vulnerability database), however this would not be very helpful because we would primarily be testing with very easy-to-detect attacks such as code-injection attacks that alter control flows in obvious ways. Indeed, many of those attacks can be detected by control-flow models. However, the real problem is that an attacker can easily adapt his attack to evade detection by these techniques. The main advantage of our approach is that due to the improved precision offered by it, it can block such stealthy attacks designed to evade existing IDS. To establish this, we:

- **Demonstrate detection of a stealthy attack like symbolic link vulnerability**

One of the famous race condition attack in the file system, but can occur in any concurrent system. The basic attack is defined by the following abbreviation **TOCTTOU.**

TOCTTOU stands for Time of Check to Time of Use.

- Check: Establish some precondition (invariant), e.g., access permission
- Use: Operate on the object assuming that the invariant is still valid.

Here we assume that the process is atomic. But that is not the case. Here there is a Race Condition involved which is often exploited.

The following is a vulnerability detected in the fingered utility in BSD (older version)

> **Fingerd symlink** vulnerability: Some programs assume that file names given to them are regular names and do not contain symbolic links. Attacks can be crafted by violating this assumption. We describe an example of symlink vulnerability in old versions of BSD fingerd [8]. This server uses a local finger client program to serve remote requests. The server runs with root privileges, and executes the client without dropping these privileges.
>
> This allows the following attack: A user can create a symbolic link called .plan in his home directory that points to a file readable only to root (e.g., the shadow password file). Now, by running a finger on himself from a remote site, he can see the contents of this file. The vulnerability arises in the following code snippet in show text() function, which verifies the As essential aspect of this attack is that the file name that is actually read isn't within the directory of the user. This is detected in our approach as a violation of the relationship between the command-line argument, which specifies the name of the user to be fingered, and the directory of the filename opened at L1. (Recall that we resolve symbolic links in file names before using them for learning or detection.) The attack could potentially be detected by observing that the resolved filename is something other than .plan, but this would raise a false alarm if the user were to use the symbolic link in a benign way, say, by linking .plan to another file named schedule.

Writing a sample exploit for sym-link vulnerability

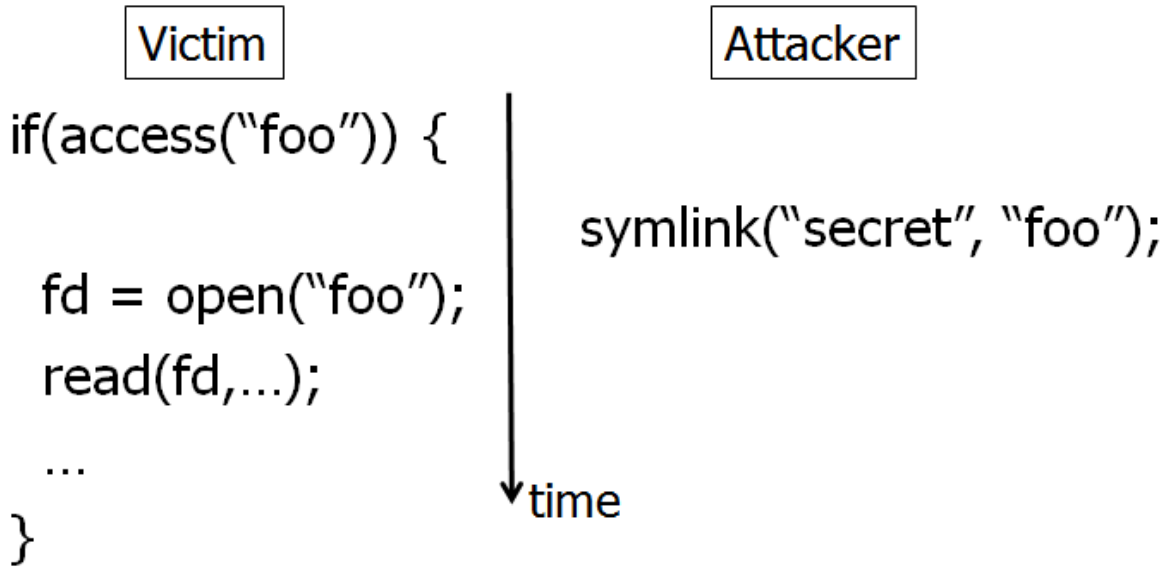
Access control: User should only be able to access a file if he has the permission to do so.
But if the same user is running a setuid program he is accessing files as root.

E.g., A printing program is usually setuid-root in order to access the printer device runs "as if" the user had root privileges

But a root user can access any file!

the printing program know that the user has the right to read (and print) any given file by making a system call called access().

UNIX has a special access() system call which checks for the permission of a particular user on that file.



Thus the attacker makes sure there is a race condition and try to win the race. And if he does then the attacker is making the changes to the sym link which he has access to (but it would be pointing to a file which he doesn't has access).

5. System Testing

Here we list out the testing performed during the modeling.

- Tried modeling an various executables like ls, find etc.

For the sake of simplicity we implemented only system calls like open, read, write, close and limited the number of open files to 10 so that the model would be small but clear enough to understand the workflow.

5.1 Test Cases:

Wrote a sample attack which makes use of the above mentioned vulnerability. We make use of a sleep() system call to mess around with the files involved which is done in another terminal window when the program is under sleep.

6. Experimental Results and Benchmarking

My Implementation works in all the *nix versions with Ptrace support using Intel Architecture. Theoretically there would be a performance overhead when an application is running with the tracer as child process stops on generating any signal waiting for the Ptrace to execute.

7. Conclusion and Future Enhancements

Even though the system is working we need to test it with many more attacks. And by including more relationships. There is also need to check the integrity of the model before actually making use of the model file. For this md5 integrity checking would be enough.

There is also scope for improvement on including sys_socket system call and thus turning socket programming more secure. Also we can observe a multitude of possibilities if we can port this application the virtual layer thus can monitor how the program interacts in a VM.

8. Resources:

- Sandeep Bhatkar, Abhishek Chaturvedi, R. Sekar. Dataflow Anomaly Detection, by Department of Computer Science, Stony Brook University, Stony Brook, NY 11794.
- Theuns Verwoerd, Ray Hunt. Intrusion detection techniques and approaches, by University of Canterbury, New zealand. 2011
- Ptrace Man page

- Sourceforge project ***strace (http://sourceforge.net/projects/strace/***) sourcecode.
- Stackoverflow.com