On Preventing Intrusions by Process Behavior Monitoring[1]

R. Sekar
Department of Computer Science
Iowa State University, Ames, IA
sekar@seclab.cs.iastate.edu

T. Bowen **M. Segal**
Bellcore
Morristown, NJ
{bowen,ms}@bellcore.com

Abstract

Society's increasing reliance on networked information systems to support critical infrastructures has prompted interest in making the information systems *survivable*, so that they continue to perform critical functions even in the presence of vulnerabilities susceptible to malicious attacks. To enable vulnerable systems to survive attacks, it is necessary to detect attacks and isolate failures resulting from attacks before they damage the system by impacting functionality, performance or security. The key research problems in this context include:

- detecting in-progress attacks before they cause damage, as opposed to detecting attacks after they have succeeded,
- localizing and/or minimizing damage by isolating attacked components in real-time, and
- tracing the origin of attacks.

We address the detection problem by real-time event monitoring and comparison against events known to be unacceptable. Real-time detection differentiates our approach from previous works that focus on intrusion detection by post-attack evidence analysis. We address the isolation and tracing problems by supporting automatic initiation of reactions. Reactions are programs that we develop to respond to attacks. A reaction's primary goal is to isolate compromised components and prevent them from damaging other components. A reaction's secondary goal is to aid in tracing the origin of attack, e.g., by providing an illusion of success to the attackers (enticing them to continue the attack) while ensuring that the attack causes no damage.

Our approach to detecting attacks is based on specifying permissible process behaviors as logical assertions on sequences of system calls and conditions on the values of system call arguments. We compile the specifications into finite state automata for efficient runtime detection of deviations from the specified (and hence permissible) behavior. We seamlessly integrate detection and reaction by designing our specification language to also allow specification of reactions.

1. Introduction

Approaches to intrusion detection can be broadly divided into *anomaly detection* and *misuse detection*. Anomaly detection based approaches first create a profile that describes *normal behaviors* and then detect deviations from this profile [Fox90, Lunt88, Lunt92, Anderson95]. In contrast, *misuse detection* based approaches [Porras92, Ilgun93, Kumar94] define and look for precise sequences of events that damage the system. Anomaly detection approaches possess the advantage that learning to identify *normal behavior* can be automated, but they are prone to false positives, especially when permissible but previously unlearned behavior occurs. Misuse detection approaches are more precise and less prone to false positives. However, since misuse detection approaches require specification of damaging events, which is usually manual and based on previously known attacks, they are less effective against newly discovered vulnerabilities and attacks.

A *specification-based approach*, first proposed by Ko *et al.* [Ko94, Ko96], aims at overcoming the above drawback of misuse detection. Instead of describing the events occurring in known attacks, which may or may not occur in future attacks, a specification-based approach describes a program's *intended* behavior. Deviations from intended behavior can be flagged as intrusions, thus enabling detection of previously un-

[1] This project is supported by Defense Advanced Research Agency's Information Technology Office (DARPA-ITO) under the Information System Survivability Program, under contract number F30602-97-C-0244. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

known attacks. Our approach uses manual production of specifications, which while having the drawback of requiring a human expert, has the advantage of minimizing false positives, especially those that arise when intended but infrequently exhibited behavior is observed. Thus, we can continue to retain the precision of misuse detection and can therefore initiate defensive actions as soon as any violations are detected.

An overview of our specification-based approach for improving survivability was presented in [Sekar98]. Our approach comprises a specification language, a compiler for the specification language, and a runtime execution environment. This paper provides a more in-depth treatment of our specification language, and outlines an approach for compiling the specifications into executable modules for efficient monitoring of program behaviors at runtime. While our approach applies in principle to any modern operating system, our implementation is specific to Linux.

The rest of this paper is organized as follows. In Section 2 we give a brief overview of our approach, in Section 3 discuss related work as it applies to our specification language, in Section 4 we present our language and practical examples of its use, and in Section 5 we describe language compilation.

2. Overview of Approach

We model the survivable system as a distributed system consisting of hosts interconnected by a network. The network and the hosts are assumed to be physically secure, but the network is interconnected to the public Internet. Since attackers do not have physical access to the hosts that they are attacking, all attacks must be launched remotely from the public network. Regardless of how the attack is delivered, any damage to a target host is effected via the system calls made by a process running on the target host.[2] Thus, it is possible in theory to detect all attacks by observing only the system calls made by processes executing on the hosts comprising the system, and to prevent damage by filtering out damage-causing system calls before they are executed. Basing our techniques on system call observation has an important advantage in its ability to defend existing software applications without modifying their source code. We therefore develop a high-level specification language called Auditing Specification Language (ASL) for specifying normal and abnormal behaviors of processes as logical assertions on the sequences of system calls and system call argument values invoked by the process. ASL specifications are compiled into optimized programs for efficient detection of deviations from the specified behavior. When discrepancies are detected at runtime, automatic defensive actions, also described in ASL, to contain or isolate the damage are initiated. A simple defense is to terminate processes that deviate from specified behavior, but this approach may not be desirable since it may alert attackers that the attack has been detected. Instead, we may want to entrap attackers into continuing their activities so that we can observe and document their actions. This can be accomplished using *isolation techniques* that enable the compromised process to continue to run, while ensuring that the process cannot damage the rest of the system. As a result, the attackers may believe that they are succeeding, while in reality, they are simply wasting their time and resources. Our defensive reactions are also written in ASL, which enables a close yet flexible coupling between detection and reaction capabilities.

Our behavioral assertions are divided into two categories; similar to correctness properties of distributed systems:

- local correctness assertions involving the actions of a single process in isolation, and
- non-interference assertions that ensure that the concurrent actions of multiple processes do not interfere with one another

To illustrate the concept of local correctness, consider a privileged program with a buffer overflow vulnerability (such as the `fingerd` program exploited by the Internet worm) that allows an attacker to execute the data input to the program. Since the input data can be constructed to be a machine language program, the vulnerability allows execution of arbitrary programs with the authority of the attacked program, which in the case of fingerd is root. A popular attack to cause execution of a program using `execve()` to execute "/usr/bin," thus providing an interactive shell with root privilege, although other options are possible. The popular attack can be prevented by the specification shown below, which prevents the program from `execve`'ing arbitrary programs, while still permitting it to execute the program(s) that it may need to execute in order to provide its normal function

[2] This observation does not hold for some denial-of-service attacks such as ping-of-death that exploit errors in operating system kernel implementations. We monitor network packets to deal with this class of attacks, but this approach is not discussed further in this paper.

```
  execve(f) | f != "/usr/ucb/finger"
      -> exit(-1)
```

As explained in Section 4, the example reads as follows. Whenever `fingerd` attempts an `execve()` system call, if the name of the file passed as the first argument to `execve()` is not `/usr/ucb/finger` then an `exit(-1)` is performed before the `execve()`.

To illustrate the concept of non-interference, consider an attack that exploits a race condition in a privileged program. The typical race condition exists because in an attempt to correctly manage file permissions in programs whose effective user and real user are different (for example setuid to root programs), programmers use two system calls, `access()` and `open()` when opening files. Both `access()` and `open()` check file permissions, but `access()` performs the check with respect to real user, while `open()` checks with respect to effective user. Therefore, to ensure that the privileged program does not open a file for which the real user does not have permission, the `access(), open()` pair is locally sufficient. However, the sequence is insufficient when interference is possible. Another process can change the underlying file in between `access()` and `open()`, so that the real user has permissions for the file checked by `access()`, but not for the file checked by `open()`. While this appears complicated, from a practical point of view the second process merely needs to execute two UNIX® commands, `rm` and `link`, to accomplish it. For correct permission checking, we need to ensure that `access()` and `open()` are executed without interference by other processes. This requires that the data read by `access`() is not modified by another process before the completion of the `open()`. We capture the non-interference requirement using the notion of an atomic sequence, which has the semantics that if any other process issues system calls that modify the data in the atomic sequence, we detect the modification as violation of the specification. In the example shown below, the notation "`a..b`" stands for the occurrence of an event `a` followed by event `b`.

```
 nonatomic (f) in
   (access(f,mode) .. open(f)) -> exit(-1)
```

2.1. System Overview

Our intrusion detection/prevention system consists of an offline and a runtime component as depicted in Figures 1 and 2.

The offline system generates detection engines based on the ASL behavioral specifications, and the runtime system executes the generated engines. For each program P to be defended, a specification M is developed by a system security administrator who is familiar with intended behavior of the P (as can be determined from its manual pages or other documentation) as well as specific known vulnerabilities obtainable from sources such as attack advisories. The ASL compiler translates `M` into a C++ class definition, called `C`. `C` is then compiled by the C++ compiler and linked with a runtime infrastructure to produce a detection engine. The runtime infrastructure provides the mechanism for intercepting system calls; delivering them to the detection engine and providing functions the detection engine uses to take responsive actions.

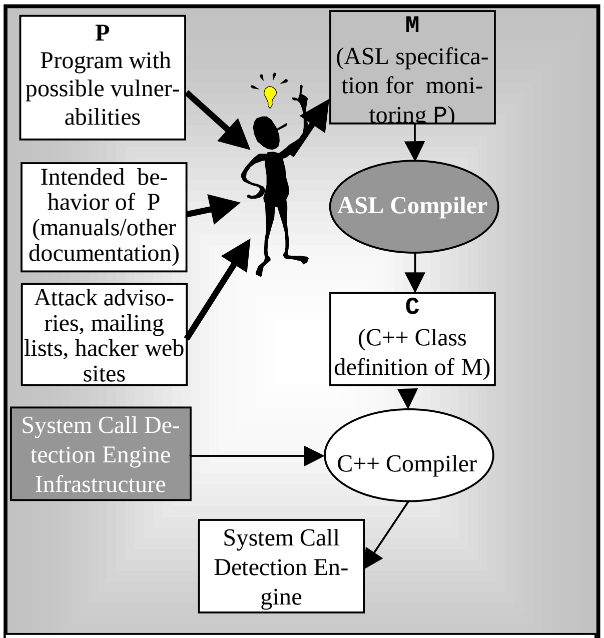
P
Program with possible vulnerabilities
M
(ASL specification for monitoring P)
Intended behavior of P (manuals/other documentation)
ASL Compiler
Attack advisories, mailing lists, hacker web sites
C
(C++ Class definition of M)
System Call Detection Engine Infrastructure
C++ Compiler
System Call Detection Engine

Figure 1 - Offline system for production of detection engines

Figure 2 shows how the detection engines generated by the offline component are used at runtime. When program P executes as process V_j, it is monitored using object O_j, which is an instantiation of `C`. For simplicity, we assume `j` is the process ID. System calls made by V_j are intercepted by the system call interceptor just before, and just after the system call's kernel level functionality is executed. At each interception, the system call information is passed to O_j through method invocation. The interception enables the system call detection engine's infrastructure and O_j to detect se-

UNIX is a registered trademark licensed exclusively through X/Open Company Ltd.

quences of system calls requested by V_j which deviate from expectation, and to modify system call execution to prevent detected deviations from causing damage.

We implement the system call interceptor within the operating system kernel. Other alternatives include interception of system calls as they pass through the system call library, libc, or using the system call tracing and process control facilities of many UNIX variants. However, these approaches do not offer the same level of security as our kernel-based approach, since they can be easily bypassed. It is also doubtful that either approach can be made as efficient as the kernel approach since the kernel approach alone allows interception and modification without process context switching.

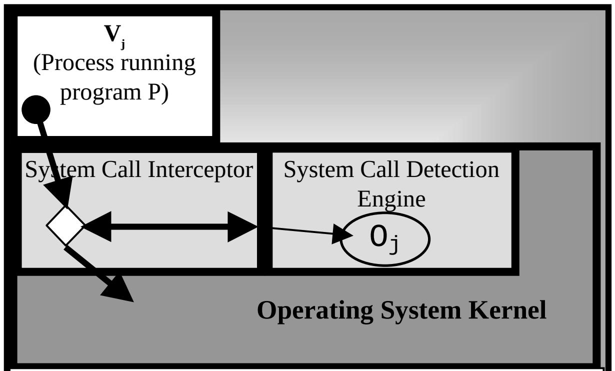

Figure 2 - Runtime system for execution of detection engines

2.2. Salient Features of Our Approach

- *Prevention.* The preventive ability makes it feasible to continue to allow the execution of programs that are known to contain exploitable vulnerabilities. Without preventive abilities, the potential of damage is so great that use of vulnerable programs must be prohibited until the program is repaired. The same reasoning even applies to programs from untrusted sources. Without assurance of damage prevention, the danger of damage from untrusted programs precludes their execution, but with damage prevention, even untrusted programs can be executed.
- *Programmability* enables a system administrator to respond quickly to a newly discovered vulnerability, without having to wait for a vendor-supplied patch.
- *Automated response.* Unlike previous approaches that focussed mainly on intrusion detection, our approach integrates detection and reaction within a uniform framework, since both are contained in the same specification. Automation reduces the need for constant involvement of teams of human experts, thus providing a more cost-effective solution.
- *Deception.* Our approach allows the development of reactions that both isolate the attacked process to prevent damage, and deceive the attacker into believing that the attack is successful. Deception enables us to observe and document attacker behavior, either for apprehending attackers or to gain a better understanding of the system vulnerabilities.
- *Dynamically tunable monitoring.* Our technique allows the granularity of monitoring to be changed on the fly at runtime. We can use a low-level of monitoring under normal conditions, but can quickly increase the level of monitoring when errors or suspicious activities are detected.

3. Related Work

Use of a specification-based approach for intrusion detection was first proposed by Ko et al. [Ko94, Ko96]. Similar to their approach, we model the behavior of a process in terms of the system calls and their arguments. However, their approach analyzes logs of system calls to detect deviations from specification, and so are limited to post-attack detection. Our system intercepts system calls as they execute, so in addition to detecting deviations, we can *enforce* the specified behaviors at runtime to prevent damage. Runtime detection demands efficient execution of specifications, so our specification language design emphasizes efficiency. [Ko96] uses a specification language based on context-free grammars augmented with state variables, while our specification language is closer to regular languages augmented with state variables. Use of regular languages allows the compilation of specifications into an extended finite-state automaton (EFSA), which is a finite-state machine that is augmented with state variables. Such an EFSA permits efficient runtime checking, while using bounded resources (CPU or memory) that can be determined *a priori*. In addition, we believe that regular languages makes our specifications easier to understand and more concise. Although regular grammars are less expressive than context-free grammars, the difference is much less pronounced when these grammars are augmented with state variables.

Forrest *et al.* [Forrest97, Kosoresow97] developed intrusion detection techniques inspired by immune systems in animals. They characterize “self” for a UNIX process in terms of sequences of system calls that are made by the process under normal conditions. Intrusion is detected by monitoring for “foreign” system call sequences. Their research results are largely complementary to ours, in that their main focus is on *learning* normal behaviors of processes, whereas our focus is on *specifying* and *enforcing* these behaviors efficiently.

Goldberg *et al.* [Goldberg96] developed the Janus environment for confining helper applications (such as those launched by web-browsers) so that they are restricted in their use of system calls. Like our techniques, their techniques prevent unauthorized operations, such as attempts to modify a user's `.login` file. But their approach is more of a finer-grained access-control mechanism rather than an intrusion detection or prevention mechanism. The key distinction between the two mechanisms is as follows. Access control mechanisms restrict access rights for each process to the minimum rights required for the process' s functionality, while intrusion detection verify that a process uses its access rights in the intended fashion. For instance, attacks based on race conditions and unexpected interactions among multiple processes manifest themselves as unintended use of access rights. Consequently, our specification language must be able to express sequencing relationships among multiple system calls made by one or more processes, whereas Janus only permits restriction of access to individual system calls made by a single process.

Our approach to isolation has some similarities with the approach used in the Deception Toolkit (DTK) [Cohen98]. In particular, when an intrusion is detected, our approach enables defenses that deceive the attacker with the illusion of success. The DTK employs a similar strategy. However, with DTK, deception depends upon enticing the attacker to use phony versions of the attacked service. The real service is no longer available at the DTK server, which contrasts with our approach, where standard server functionality is still present for legitimate uses.

As compared to our earlier work in [Sekar98], this paper presents a significantly improved version of ASL. It also outlines an approach for compiling the high-level specifications into finite-state automata that perform efficient runtime monitoring of process behavior. Improvements to ASL described in this paper are as follows. We have developed a more elegant approach for dealing with race conditions and other similar errors that result due to interference in data access by multiple processes. The pattern language for behavioral specification has also been improved by separating different classes of patterns. To further improve conciseness of specifications, the notion of event abstractions has been introduced. Another important improvement is the introduction of an interface definition component to ASL so as to decouple the ASL compiler from the specifics of the events monitored by the detection engine. As a result, we can now write ASL specifications that model system behaviors in terms of any observable events, as opposed to being limited to observation of system calls. Moreover, the ASL compiler need not be changed to deal with these new event types — we simply need to link the code produced by the ASL compiler with appropriate runtime infrastructure that can deliver these new events to the detection engine.

4. Auditing Specification Language (ASL)

We model the behavior of a process in terms of the system calls the process makes. We treat these system calls as *events*, which have the general form $e(a_1, \ldots, a_n)$, with e denoting the event name and $a_1, \ldots, a_n$ denoting the event arguments. Two events are associated with each system call, namely the entry to the system call and exit from the system call. We distinguish system call entry events from system call exit events by prefixing the $-symbol to exit events.

4.1. Interface Declarations

The interface between the detection engine and the monitored processes supports the conveyance of events from the process to the detection engine, and the conveyance of response functions from the detection engine to the monitored process. The functionality of the interfaces is realized via a set of interface functions that deliver events to the detection engine and provide mechanisms for invoking response actions. For generality, the functionality provided by the interface is specified in ASL via interface declarations. These declarations specify

- datatypes that can be exchanged over the interface
- events delivered over the interface in terms of their names, arguments and types
- external functions[3] provided by the interface that can be invoked by the detection code

We describe each of these components below.

4.1.1 ASL Data Types

Built-in types in ASL include `bit, byte, short`, `int`, `long`, `double`, and `string`. All of the integral types excluding `bit` and `byte` are either signed or unsigned. Their sizes coincide with the norm for the specific host for which the ASL specification is being

[3] We call the response function *external functions* to differentiate them from internal functions that are built into ASL.

applied. The string type is a variable length byte array prefixed with a 2-byte length field. ASL supports multi-dimensional arrays of built-in types.

Foreign types, correspond to data that can be exchanged on one or more of the interfaces, but whose representation is opaque to ASL. Foreign types are designed with the intent of modeling data within the virtual memory space of a monitored process. Depending on the particular implementation approach used in the detection engine, it may or may not be easy (or even possible) to access such data directly. To address this problem, we have developed *class* types that cannot be directly accessed in ASL, but can only be accessed using member functions defined on the type. Class types correspond to abstract data types. A sample class definition corresponding to a C-style string is:

```
class CString {
            string getVal() const;
            void setVal(string s);
}
```

A more complex definition suitable for manipulating data associated the stat system call is given below.

```
class StatBuf {
            int getDev()const;
            int getIno()const;
            int getMode()const;
             ⋮
            int getAtime()const;
            int getMtime()const;
            int getCtime()const;
}
```

Note that the return type of a member function could itself be a foreign type. Whether a member function changes the value of the object or not is given by the presence or absence of the `const` keyword in the declaration of the function. This fact is used by the ASL typechecker to ensure that expressions in ASL do not cause unexpected errors when evaluated at runtime.

Since ASL specifications may be compiled into detection engines that run within an operating system kernel, safety and reliability are especially important. Two important language mechanisms in ASL that promote safety and reliability are strong typing and the absence of pointer types.

4.1.2. External Functions

External functions are functions that are defined outside of the detection engines, but can be accessed from the detection engines. Semantically, they are no different from member functions associated with foreign types. In other words, member functions are simply external functions that use a different syntax.

The primary purpose of external functions is to invoke support functions needed by the detection engine or reaction operations provided by the system call interceptors. For instance, when an event for opening a file is received by a detection engine, the detection engine may need to resolve the symbolic links and references to "." and ".." in the file name to obtain a canonical name for file. The detection engine may use a support function declared as follows to find the canonical file name:

```
string realpath(CString s);
```

The detection engine may also need to check the file's access permissions, which may be done using a support function declared as follows:

```
StatBuf stat(const Cstring s);
```

In ASL system call names either represent an event (i.e., invocation of a system call by a monitored process) or are a component of a reaction taken by the detection engine (i.e., a statement in the a reaction program). We use the same syntax for system calls in both cases, since the context resolves any ambiguity.

4.2. Modules

The ASL specifications are structured as a collection of parameterized modules, each of which consists of a collection of *state variables* and *rules.* State information can be retained across multiple rules within a module via the state variables.

As an aid to programmability, modules may be parameterized. Parameterization enables specification of abstract behaviors that can be customized by providing values for these parameters. A typical use of parameterization is to allow a general-purpose module to be used in nearly identical situations that differ only in a few minor details. The process of generating a compilable module from a parameterized module is known as module *instantiation.*

Another important role of modules is that they provide a mechanism for dynamically altering the degree of monitoring, possibly in response to suspicious events. In particular, the action `switch` *`ModuleName`* can be used to start monitoring with respect to a module named *`ModuleName`*. It is also useful when a process uses the `execve()` system call to overlay itself with a new program. The `switch` action can then be used to perform monitoring that is appropriate for the new program. Finally, if a process is discovered to be compromised, we can alter the behavior of future system calls made by the process in such a fashion as to isolate the process from the system. This may also be accomplished by switching to a new specification.

4.3. Event Patterns

ASL *general event patterns* are used to specify valid or invalid behaviors. An *atomic pattern* is of the form $e(a_1,\ldots,a_n)\,|\,C$, where e denotes an event and C is a boolean-valued expression on $a_1,\ldots,a_n$. C may contain standard arithmetic, comparison and logical operations. C may also contain comparisons of the form x = *expr* where x is new variable, with the semantics being that of binding the value of *expr* to *x*. A *primitive pattern* is obtained by combining atomic patterns with the disjunction operator `||`, and possibly preceding the entire expression with the complement operator '`!`'. As an example of a primitive pattern, consider:

```
!((open(f)|realpath(f)=/home/*/.plan)
    || (close(f))||(exit(f))
```

In this pattern, a shorthand notation `/home/*/` is used to refer to any directory that is immediately contained within `/home`. The above primitive event pattern captures all system calls other than those for opening ".plan" files, closing files or terminating processes. (For illustrative purposes this example is simplified, it does not, for example, permit the opening of some necessary files, such as dynamically loaded libraries.)

General event patterns are obtained by combining primitive patterns using temporal operators. Such operators enable us to capture sequencing or timing relationships among system calls:

- *Sequential composition:* $p_1; p_2$ denotes the event pattern p_1 immediately followed by pattern p_2.
- *Alternation:* $p_1 \,\|\, p_2$ denotes the occurrence of either p_1 or p_2.
- *Repetition:* $p\{n_1,n_2\}$ denotes at least n_1 repetitions and at most n_2 repetitions of p. $p\{n_1\}$ and $p\{,n_2\}$ are shorthand for $p\{n_1,\infty\}$ and $p\{0,n_2\}$ respectively. The notation $p*$ is shorthand for $p\{0,\infty\}$.
- *Real-time constraints:* p **within** $[t_1,t_2]$ denotes the occurrence of events corresponding to pattern p occurring over a time interval. The shorthand for [0,t] is [t], whereas the shorthand for [t,∞] is [t,].
- *Atomicity:* **nonatomic** d **in** p corresponds to an occurrence of pattern p within which the data item d is not accessed atomically.

For convenience, we define the operator ".." that can be applied only to primitive patterns. $p_1 \,..\, p_2$ is equivalent to $p_1;(!(p_1 \| p_2)*);p_2$, i.e., p_1 followed by p_2 with possibly other events occurring in between. The restriction that ".." be applied only to primitive patterns is imposed since the operator has unintuitive semantics on general event patterns.

We illustrate the use of temporal operators using several simple examples below. Note that in general, we wish to take reactive action when the behavior of a monitored process fails to satisfy certain properties. Hence, we typically develop patterns that are the negation of assertions describing normal behaviors.

- `e1;!e2*;e1` asserts that `e1` must occur twice with no intervening `e2`. This corresponds to the negation of the property that `e1` must always be followed by `e2` before a second occurrence of `e1.`
- `(e1;!e2*) within [t,]` captures violation of property that `e1` is followed by `e2` within time t
- `e1;!e2*;e3` captures violation of property where `e2` must always occur between `e1` and `e3`
- `e{k} within [t]` captures violation of property that `e` occurs less than `k` times within time `t`

4.4. Event Abstractions

An event abstraction is a convenience mechanism allowing programmer definition of abstract events comprising arbitrary event patterns. Event abstractions allow the programmer to name and treat complex event patterns as if they were primitive events. To illustrate the use of event abstractions, note that many UNIX system calls have overlapping functionality. When we write behavioral specifications, it is cumbersome to write several variants of the specification based on the exact system calls used by a particular program. For convenience, we group similar system calls so that all of the calls in one group can be viewed as implementations of a higher level abstract system call. For instance, the `creat()` and `open()` system calls can both be used to open new files, so we define the abstract event `writeOpen` which captures this commonality. Then, a single behavioral specification using `writeOpen` can be used to monitor processes that open new files using either `creat()` or `open().`

```
 event writeOpen(path) =
   open(path, flags) |
     flags&(O_WRONLY|O_APPEND|O_TRUNC)||
   open(path, flags, mode) |
     flags&(O_WRONLY|O_APPEND|O_TRUNC)||
   creat(path, mode);
```

Code Example 1 - Definition of `writeOpen()` Abstract Class

Different levels of abstraction may be desired in different contexts, and hence there may be overlaps among different user-defined abstract events. For instance, we may have an abstract event that corresponds to `readOpen`, and another that corresponds to any open, regardless of whether it is for reading or writing. For simplicity, we restrict the definition of abstract events to be primitive event patterns.

4.5. Rules

A rule is of the form *pat* → *action*, where *pat* is a pattern of the form described above, and *action* is a sequence of responsive steps to be initiated when an event matching the pattern occurs. Actions may be empty, variable assignment, function invocation, or switch. Function invocation causes the specified function to be executed by the runtime infrastructure, and thus may be used by the detection engine for purposes such as reading or writing data in the monitored process, or executing arbitrary system calls in the monitored process. The `switch` *`SpecName`* action enables switching to the behavioral specification named *`SpecName`* for monitoring.

5. Example Behavior Specifications

In this section we illustrate ASL using several example specifications.

5.1. Finger Daemon

The following specification restricts the `finger` daemon[4] so that it can open only specific files for reading, cannot open any file for writing, cannot execute any file, and cannot initiate a connection to any host. If any specified behavior is attempted, the system call associated with the attempt does not execute. Instead, an error code is returned or the process terminated. For events whose arguments are not of interest, it is not necessary to specify the arguments. We make use of a support function, `inTree,` which determines whether a file resides within a directory or its descendents. The example shows only a subset of those system calls that must be disallowed for an adequate defense.

```
 open(file, mode)|
   ((f = realpath(file)) &&
    ((f != “/etc/utmp”) &&
     (f != “/etc/passwd”) &&
     !inTree(f,“/usr/spool/finger”)) ||
    (mode != O_RDONLY))
 -> fail(-1,EACCESS)
 execve || connect || chmod || chown
   || chgrp || create || truncate
   || sendto || mkdir
 -> exit(-1);
```

Code Example 2 - ASL Specification for Monitoring `fingerd`

5.2. Race Conditions

We illustrate two approaches to protect against race condition attacks. Our first approach monitors for an `access()` followed by an `open()` and ensures that both use identical conditions for checking permission. Identical in this case means that the effective user at the time of `open()`is the same as the real user at the time of `access().`

Rprog1 defines two state variables and an event abstraction for use in the rules defined subsequently. The event abstraction simplifies the structure of the rules. In the first rule, the comparisons in `acc1` event definition bind the temporary variable `ruid`. Whenever the monitored process performs an `open()` following an `access()`on the same file, we temporarily set the effective user ID of the monitored process to the value of the real user ID before the `open()` executes. Before doing this, we save the current value of the effective user ID in the state variable `savedEuid`, and set a flag `changedEuid` to record that we have temporarily changed the effective user ID. When `open()` completes, we use the values stored in the state variables to restore the original effective user ID.

```
 int savedEuid;
 bit changedEuid;

 event acc1(name, ruid) =
  access(name, mode)|(ruid = getuid());

 acc1(name, ruid)..open(name1, flags)|
   (name = name1)
 -> changedEuid = 1;
    savedEuid = geteuid();
    setreuid(-1,ruid);

 $open(f, fl)|(changedEuid = 1)
 -> changedEuid = 0;
    setreuid(-1,changedEuid);
```

Code Example 3 - ASL Specification `rProg1` for a Race Condition Vulnerability

[4] The specification pertains to the GNU finger program, and in particular, the finger daemon running as the master server. Note that GNU finger is implemented differently from the BSD finger daemon, and does not need to `execve` the finger program.

The second defense against the race vulnerability uses the concept of atomic sequences. The race vulnerability exists because two system calls `access()` and `open()` must be used to accomplish what is essentially a single function, that is, opening a file with respect to real user’s permissions. We can execute a sequence of system calls as if they were all a single system call by placing them in an atomic sequence as follows:

```
nonatomic (f) in
    (access(f,md) .. writeOpen(f))
  -> fail(-1,EACCESS)
```

An atomic sequence is a sequence of system calls executed by process P whose execution appears not to be interleaved with the system calls of any other concurrently executing process. Atomic sequences are similar to transactions in databases. Atomic sequences depend on the definition of read and write sets for all system calls. We also note that runtime checking of atomicity requires coordination among the monitors for different processes, since it depends not only on the system calls performed by a process being monitored, but also the calls made by other process.

5.3. Program from Untrusted Source

To ensure that a program from an untrusted source does not damage the host executing it, we want to ensure that the program can read only world readable files, can write only within the `/tmp` directory, cannot execute any programs, and cannot perform network operations.

```
open(file, mode) |
  [(!inTree(realpath(file), “/tmp”) &&
   (mode & (O_WRONLY|O_APPEND|
            O_CREAT | O_TRUNC)))||
   !accessible(realpath(file), mode,
               “nobody”))
-> fail(-1,EACCESS);

exec || connect || bind || chmod ||
 chown || chgrp || create ||
 truncate ||  sendto || mkdir
-> exit(-1);
```

Code Example 4 - ASL Specification `sandbox` for Untrusted Programs

5.4. Using Specification for Isolation

When we detect an attack on process V_j, we can use the `switch` action to switch to a specification that contains ASL rules to isolate V_j. The isolation specification contains rules that modify the behavior of system calls made by V_j in such a way that V_j is prevented from executing operations that can damage the survivable system. For example, the isolation specification can perform one or more of the following:

- return faked return value. When a system call that can potentially damage the system is invoked by the isolated process, we can prevent the system call from being completed, and instead return a faked (but legitimate) return value.
- log the activity for later analysis.
- reduce limits on resources that the isolated process can consume.
- restrict access to files. We can use the `setuid()` system call to change the effective user ID of the process to that of a user with very few access rights and we can use the `chroot()` system call to change the root directory of the compromised process.

To illustrate this idea, consider the modification to the previous specification for the `finger` daemon which implements isolation. In particular, we introduce the rule:

```
execve ->
  chroot(“/altroot”); setuid(-1);
  nice(100); switch genericIsolate;
```

This rule changes the root of the monitored process to a decoy file system (called altroot), changes the user ID to nobody, reduces the priority of the process, and finally switches to a new monitoring specification called `genericIsolate`.

```
module genericIsolate
  connect
    -> sleep(60); fail(-1,ETIMEDOUT);
  bind
    -> sleep(5); fail(-1,EADDRINUSE);
  recv
    -> sleep(1);
  open
    -> sleep(1);
end
```

Code Example 5 – ASL Specification for Damage Prevention

As shown, `genericIsolate` gives only a few of the rules that would be needed for isolation. Since the isolated process is operating in a decoy file system, file system operations are allowed. However, network operations are restricted. Most operations are slowed down using `sleep()`, so that the CPU and resource usage of the attacked host are reduced, but the attacker will probably attribute the delay to normal host or network congestion.

6. Compilation of ASL

The main task in translating an ASL specification into a C++ class definition is to translate the patterns into an extended finite-state automaton (EFSA). An EFSA is similar to a finite-state automaton, with the following differences:

- In addition to the control state of an FSA, an EFSA can make use of a fixed set of state variables.
- The EFSA makes transitions based on events, event arguments and conditions on event arguments and state variables. The transitions may assign new values to state variables.

An EFSA may be deterministic (DEFSA) or nondeterministic (NEFSA). For the sake of efficiency, we always prefer to generate a DEFSA rather than a NEFSA. However, this is not always possible as conversion of NEFSA into a DEFSA can cause unacceptable explosion in space requirements. For traditional FSA, every nondeterministic automaton can be converted into an equivalent deterministic automaton with at most an exponential increase in the number of (control) states. For performance critical applications (e.g., lexical analysis phase of a compiler), this increase in state space is quite acceptable, especially because the worst case behavior is unusual. For EFSA, the explosion in size is exponential in the product of the number of control states and the range of values that can be assumed by each of the auxiliary state variables. For instance, a deterministic EFSA that is equivalent to a nondeterministic EFSA with one integer (32-bit) state variable and N control states can have $2^{N*2^{32}}$ states! This problem leaves us with two choices:

- restrict the class of ASL patterns so that they can be compiled into DEFSA, or
- do not convert an NEFSA into an EFSA, and simulate the NEFSA at runtime.

Note that at runtime, the transitions of an EFSA are represented in code, whereas its current state (which includes the control state and the state variables) is stored in a data structure. Since we plan to combine all patterns in one ASL specification into a single EFSA, there is only one instance of the transition relation at runtime. To support nondeterminism, we permit multiple instances of the dynamic state of the EFSA. These multiple instances capture all of the states the NEFSA could have reached after examining its input up to this point.

If an EFSA needs to make a two-way nondeterministic transition on an event e, we perform a “fork” operation on the EFSA, i.e., replicate its current state. The replica follows one of the non-deterministic choices, while the parent follows the other choice. This approach can lead to an unbounded increase in the number of instances of EFSA, but unbounded growth should happen only when certain unusually repetitive sequences of system calls are observed at runtime, and hence is not a serious issue in practice. We are currently working on techniques that can avoid unbounded growth by restricting the class of patterns permitted in ASL.

The starting points for our algorithm for generating EFSA from ASL patterns are the seminal papers by Brzozowski [Brzozowski64] and Berry and Sethi [Berry86]. However, these papers address regular expressions and classical FSA, whereas we must address conditions on event arguments and state variables that can be complex data structures. Our earlier work on first-order term-matching [Sekar95] provides the starting point for addressing this aspect. By combining and extending these two techniques, we developed an algorithm for generating EFSA from a restricted class of ASL specifications. A detailed description of this algorithm is beyond the scope of this paper, so we only provide a description of how we map an NEFSA into C++ code.

At code generation time, the EFSA generated from ASL specifications is turned into a C++ class. Specifically, one class is generated from each ASL specification. This class has one member function for each event, and these member functions have the same number and types of arguments as the event. When the runtime infrastructure intercepts an event, it delivers it to the appropriate detection engine by invoking the corresponding member. For instance, the runtime infrastructure invokes the `open_entry` method when a monitored program enters an `open` system call, and the `open_exit` method when the process is about to exit this system call.

The transitions in the EFSA are translated into code as follows. We maintain a list of active EFSA instances at runtime. When an event is delivered, we go through the list of EFSA instances and for each of them, make a transition based on its current state and the newly delivered event. If multiple transitions exist out of the current EFSA state for this event, then copies of the EFSA are made (using the fork operation mentioned earlier), so that there is one EFSA to make each of these transitions. If there is no transition for an EFSA instance, then it is “killed” and any resources used for the instance are released.

7. Conclusions and Future Work

In this paper we presented an approach for intrusion detection that is based on specifying the valid behaviors of processes in terms of system call sequences together with constraints on the argument values that the processes can make. We described our specification language and illustrated it with several examples. Based on these examples, we are optimistic that concise and clear specifications of security-related behaviors can be developed with relative ease in the ASL language. These examples also indicate that the approach can successfully prevent (or at least quickly detect) attacks. Additional preliminary evidence in this context was presented in [Sekar98] where we examined the attack advisories from CERT®over the past five years and concluded that most of them can be detected by our approach.

We are continuing to refine and experiment with our specification language. We are also developing algorithms for compiling ASL specifications into deterministic EFSA, rather than non-deterministic EFSA. In parallel, we are also in the process of developing medium to large-scale experiments designed to assess the performance impact of our online monitoring approach. Our preliminary indications are that indeed we can do such monitoring using our current, kernel-level interception approach easily, especially since our EFSA enable efficient checking of specification assertions at runtime.

References

[Anderson95] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, *SRI-CSL-95-07, SRI International*, 1995.

[Aslam96] T. Aslam, I. Krsul and E. Spafford, A Taxonomy of Security Faults, *National Computer Security Conference*, 1996.

[Berry86] G. Berry and R. Sethi, From Regular Expressions to Deterministic Automata, *Theoretical Computer Science* 48 pp. 117-126, 1986.

[Bishop96] M. Bishop and M. Dilger , Checking for Race Conditions in File Access. *Computing Systems* 9(2), pp. 131-152, 1996.

[Brzozowski64] J.A. Brzozowski, Derivatives of Regular Expressions, *Journal of ACM* Vol. 11, No.4, pp. 481-494, 1964.

[Cai98] Y. Cai. A Specification-Based Approach for Intrusion Detection. *M.S. Thesis, Department of Computer Science, Iowa State University*, Dec 1998.

[CERT98] CERT Coordination Center Advisories 1988--1998, http://www.cert.org/advisories/index.html.

[Cheswick92] W.R. Cheswick, An evening with berferd, in which a cracker is lured, endured and studied, *Winter USENIX Conference*, 1992.

[Cohen98] Fred Cohen and Associates, The Deception Toolkit Home Page, http://www.all.net/dtk/dtk.html.

[Connet72] J. Connet et al., Software Defenses in Real-Time Control Systems, *IEEE Fault-Tolerant Comp. Sys.*, 1972.

[Denning87] D. Denning, An Intrusion Detection Model, *IEEE Trans. on Software Engineering*, Feb 1987.

[Forrest97] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, *Comm. of ACM* 40(10), 1997.

[Fox90] K. Fox, R. Henning, J. Reed and R. Simonian, A Neural Network Approach Towards Intrusion Detection, *National Computer Security Conference*, 1990.

[Goldberg96] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, A Secure Environment for Untrusted Helper Applications, *USENIX Security Symposium*, 1996.

[Hlady95] M. Hlady, R. Kovacevic, J. J. Li. et al., An Approach to Automatic Detection of Software Failures, Proc. *IEEE 6th International Symposium on Software Reliability Engineering*, 1995.

[Ilgun93] K. Ilgun, A real-time intrusion detection system for UNIX, *IEEE Symp. on Security and Privacy*, 1993.

[Ko94] C. Ko, G. Fink and K. Levitt, Automated detection of vulnerabilities in privileged programs by execution monitoring, *Computer Security Application Conference*, 1994.

[Ko96] C. Ko, Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach, *Ph.D. Thesis, Computer Science, University of California at Davis*, 1996.

[Kosoresow97] A. Kosoresow and S. Hofmeyr, Intrusion detection via system call traces, *IEEE Software* '97.

[Kumar94] S. Kumar and E. Spafford, A Pattern-Matching Model for Intrusion Detection, *National Computer Security Conference*, 1994.

[Landwehr94] C. Landwehr, A. Bull, J. McDermott and W. Choi, A Taxonomy of Computer Program Security Flaws, *ACM Computing Surveys 26(3)*, 1994.

[Lunt92] T. Lunt et al., A Real-Time Intrusion Detection Expert System (IDES) - Final Report, *SRI-CSL-92-05, SRI International*, 1992.

[Lunt93] T. Lunt, A survey of Intrusion Detection Techniques, *Computers and Security*, 12(4), June 1993.

[Mukherjee94] B. Mukherjee, L. Todd Heberlein, Karl N. Levitt. Network Intrusion Detection, *IEEE Network*, pp.26-41, May/June 1994.

[Porras92] P. Porras and R. Kemmerer, Penetration State Transition Analysis - A Rule Based Intrusion Detection Approach, *Computer Security Applications Conference*, 1992.

[Sekar95] R. Sekar, I.V. Ramakrishnan and R. Ramesh, Adaptive Pattern Matching, *SIAM Journal of Computing*, 1995.

[Sekar98] R. Sekar, Y. Cai and M. Segal, A Specification-Based approach for Building Survivable Systems, 21st National Information Systems Security Conference.

[Spafford91] E. H. Spafford. The Internet Worm Incident, *Technical Report CSD-TR-993*, Purdue University, West Lafayette, IN, September 19, 1991.

[Vankamamidi98] R. Vankamamidi. ASL: A specification language for intrusion detection and network monitoring. *M.S. Thesis, Department of Computer Science, Iowa State University*, Dec 1998.

[Yang98] G. Yang. A Real-time Packet Filtering Module for Network Intrusion Detection System, *M.S. Thesis, Department of Computer Science, Iowa State University*, Jul 1998.