Model-Based Analysis of Configuration Vulnerabilities[*]

C.R. Ramakrishnan and R. Sekar

{cram,sekar}@cs.sunysb.edu

Department of Computer Science

State University of New York

Stony Brook, NY 11794.

Abstract

Vulnerability analysis is concerned with the problem of identifying weaknesses in computer systems that can be exploited to compromise their security. In this paper we describe a new approach to vulnerability analysis based on model checking. Our approach involves:

- Formal specification of desired security properties. An example of such a property is "no ordinary user can overwrite system log files."
- An abstract model of the system that captures its security-related behaviors. This model is obtained by composing models of system components such as the file system, privileged processes, etc.
- Verification techniques to check whether the abstract model satisfies the security properties.

A model-based approach can be used to detect known and as-yet-unknown vulnerabilities, in contrast with previous approaches (such as those used in COPS and SATAN) which mainly address known vulnerabilities.

This paper demonstrates our approach by modelling a simplified version of a UNIX-based system, and analyzing this system using model-checking techniques to identify nontrivial vulnerabilities. A key contribution of this paper is to show that such an automated analysis is feasible in spite of the fact that the system models are infinite-state systems. It shows that known techniques, such as model abstraction and contraint-based representations, can be employed to tackle the infinite-state model-checking problems that arise in vulnerability analysis.

Clearly, a realistic UNIX system is much more complex than the one that we have modelled in this paper. Nevertheless, we believe that our results show that automated and systematic vulnerability analysis of realistic systems to be feasible in the near future.

[*]This research is supported partially by DARPA-ITO under contract number F30602-97-C-0244 and by the NSF under grants CCR-9705998 and CCR-9711386.

1 Introduction

System configuration vulnerabilities can be traced back to classic problems in software engineering, such as unexpected interactions between different system modules and violation of hidden assumptions. For instance, consider a vulnerability that existed in early versions of the `fingerd` service. In servicing a query "`finger` *user*," this program needs to read a file named `.plan` in the home directory of *user*. The `fingerd` service runs with root privileges, and in the earlier versions of UNIX used to open the `.plan` as root. In the presence of symbolic links, this creates the following vulnerability. User u could symbolically link a file f as his/her `.plan` even if u has no read access to f. User u can then read f by simply running `finger` u! This vulnerability arises due to the interaction between the way the finger server operates and the way symbolic links are implemented.

As a second example, consider the vulnerability involving the mail notification program `comsat`, which waits for reports of incoming mail for any user and prints the first few lines of the message on the terminal in which the user is logged on. This terminal is determined from the file `/etc/utmp`, which used to be configured as world-writable (so that it can be written to from user level at the time of login). A malicious user can modify `/etc/utmp`, substituting `/etc/passwd` in the place of the terminal that he/she is logged on. The user can then send mail to self containing a line that starts with `root::0:0:` (which means that `root` has empty password). Upon receiving this mail, `comsat` will overwrite the password file with the message. The user can now login as `root` without providing a password.

1.1 Our Approach

Formal methods are a good choice to address software engineering problems that arise due to unexpected interactions among system components [9]. In this paper, we describe an approach for analyzing system configuration vulnerabilities using techniques drawn from formal methods research. Our approach involves:

- *Construction of high-level models of system components.* In order to detect the kind of vulnerabilities described above, we would start with abstract models that capture the behavior of UNIX file system, `comsat` and mailer programs, and a model of user behavior. Currently, we are developing these abstract models manually. In future, we expect that the model extraction process would be machine-assisted, employing program analysis techniques.

- *Formal statement of desired security-relevant properties of the composite system.* Vulnerabilities can be viewed as "flaws" in system configuration that can be exploited to violate certain security objectives of the overall system. To detect vulnerabilities, we need a formal statement of such security properties. One example of a property is that no ordinary user can overwrite system log files. Another example is that the password file cannot be modified except by a superuser, or by using a password changing program.

- *Automated analysis of system model to check deviation from desired security properties.* Configuration vulnerabilities can be identified by analyzing the overall system behavior (obtained by composing the models developed above), and determining if it violates the desired security properties. We use model-checking [6, 16, 7] for this analysis. An important benefit of model checking is that when a property is violated, a model checker provides a counterexample that shows how the property is violated.

This paper demonstrates our approach by modelling a simplified version of a UNIX-based system, and discovering nontrivial configuration vulnerabilities in this system through automated analysis. Clearly, a realistic UNIX system is much more complex than the one that we have modelled in this paper. Nevertheless, we believe that our results show that automated analysis of realistic systems to be feasible in the near future, especially since faster techniques for model-checking are being developed.

Although the use of model-checking has been examined in related contexts such as verification of cryptographic protocols and network vulnerabilities [20], ours is the first attempt at using model-checking for system configuration vulnerability analysis. One of the principal difficulties in this context is that system models tend to possess an infinite number of states. This is because we need to model such aspects of the system as file names and (nested) directory structures, with the ability to create or destroy an infinite number of files. Traditional model-checking techniques (such as those employed in [20]) are limited to finite-state models only.

A key contribution of this paper is to show that nontrivial vulnerabilities can be discovered using automated model-based analysis, in spite of the fact that the system models are infinite-state. It shows that known techniques, such as model abstraction and contraint-based representations, can be employed to tackle the infinite-state model-checking problems that arise in vulnerability analysis.

The rest of this paper is structured as follows. In Section 1.2, we describe previous approaches for vulnerability analysis and summarize the main benefits of a model-based approach over previous techniques. In Section 2, we describe our approach for modelling a simplified subset of UNIX. Our analysis approach is described in Section 3. The results of this analysis are presented in Section 4. Finally, concluding remarks appear in Section 5.

1.2 Related Work

Research in vulnerability analysis has focussed primarily on identification of *configuration errors* such as improper file permission settings. Existing works [11, 2, 23] employ a set of rules that enumerate *known* causes for vulnerabilities. We call these works collectively as *rule-based.* Widely used tools, such as COPS and SATAN search for occurrences of such known vulnerabilities [11]. However, the generation of the rules relies on expert knowledge about interactions among many components of the system. Few experts have a complete understanding of the interactions among all components of modern computer system. Issues such as race conditions, many possible interleavings, hidden assumptions etc. [3] make it very hard for humans to come up with all such rules.

A model-based approach does not suffer from these disadvantages. Human involvement is needed primarily to develop models of individual system components. The problem that is hard for human reasoning, namely, that of reasoning about interactions among system components, is relegated to a mechanical procedure. The advantages of a model-based approach are:

- *Identification of known and unknown vulnerabilities.* Analysis of formal models can identify known and *as-yet-unknown* vulnerabilities. In contrast, the rule-based approaches have been limited to examining the system for known vulnerabilities.

- *Modularity.* The effort required to add new system components (e.g. new privileged programs or significant software upgrades) is determined only by the new components to be added. Models of existing components need not be changed. This contrasts with rule-based approaches, where new rules need to be added that capture not only the interactions among new components, but also interactions between new and old components.

- *Generating patterns for misuse intrusion detection.* In the case of vulnerability analysis, counter-examples generated by a model checker correspond to exploits. These exploits can be mechanically translated into patterns for a misuse intrusion detection system.

This paper builds on some preliminary results on model-based vulnerability analysis we had reported earlier in [17]. Since then, Ritchey and Ammann [20] have suggested a promising approach for automating network vulnerability analysis. Their approach starts with higher level models than ours. Their models capture *known exploits* on individual systems, e.g., that a given version of a web server contains a vulnerability that allows a remote user to gain access as a local user, and that a certain host is running this version of the server. Model checking is then used to check if these exploits can be "strung together" to achieve a greater degree of access than what can be obtained by individual exploits. In contrast, our approach is aimed at discovering the individual exploits from models of (legitimate) behaviors of systems. Another important difference is that their models are finite, which enables them to use widely available model checking tools such as SMV [8] and SPIN [13] to perform vulnerability analysis. In contrast, we need to deal with infinite state systems. This is because finite models cannot capture components such as file systems where files can be added, renamed or removed, and there is no bound on how many times these operations may be repeated.

2 Modeling Security-related Behaviors of Systems

In this section, we first describe our model for a small subset of a UNIX-based system. This subset captures a simplified view of the file system and other operating system facilities, and is sufficient to uncover nontrivial vulnerabilities during the analysis process. We begin with a short description of the language we use for modelling, described in enough detail to understand the models. (A complete description of the language is not relevant for the purposes of this paper, and hence not include here.)

2.1 Modeling Language

We describe our model using a modelling language similar to CSP [12], but extended with object-oriented capabilities. Since our underlying model checker is based on the Prolog language, many features of the modelling language are similar to those in Prolog. In addition, some of the syntactic features are similar to those used in Promela, the modelling language used in the SPIN [13] verification system.

In this language, a system is modelled as a collection of concurrently executing processes that communicate with each other. Each process is viewed as an object. Its internal state is encapsulated, and cannot be accessed by other processes. Communication among processes takes place via method invocations. Method invocation is synchronous: it causes the invoking process to block until the method invocation is completed, and return values sent back.

Objects are instances of classes. A class definition consists of the definition of the encapsulated state, definitions of externally visible methods and definitions of local methods (otherwise known as private methods or helper functions). Single inheritance is supported in the language. Thus, a class definition is of the form:

class ClassName(DataMemberNames) [:BaseClassName(DataMemberNames1)] {MethodDefn}+

An object is created by invoking the class name with parameters corresponding to its data members, using the syntax:

ClassName(DataMembers)

Methods are defined using the following syntax.

[**private**] MethodName(Parameters) ::= MethodBody.

Like predicates in the Prolog language, method invocations in this language always return a boolean value. In addition, some of the arguments to a method may get instantiated as a result of invocation. This feature is used to communicate return values. We use the convention that all of the return parameters appear after the input parameters in method invocations.

The language supports basic data types such as booleans, integers, floats and strings. Usual operations on these types are also supported, and can be used to construct complex expresions from values (or variables) of basic types. The language also supports compound types based on an algebraic type system, similar to that provided by Prolog. These types are used to represent structured data such as file names, file contents, and the contents of the entire file system. Two of the most common compound types are tuples and lists.

We assume that different processes execute concurrently. The language uses an interleaving semantics to determine the result of concurrent executions. At the lowest level, operations such as assignment are performed atomically.

A method body consists of a sequence of operations. All operations return a boolean value, with the value `true` denoting successful completion of the operation and `false` denoting failure. Method invocation begins with the execution of the first operation in the sequence. If this operation succeeds, then the next operation is executed and so on. If all of the operations in a sequence succeed, then we say that the entire sequence succeeds. Otherwise, the sequence fails, returning the value `false`. In such a case, the method invocation itself returns `false`, indicating failure.

Operations can either be primitive or compound. A primitive operation is either a method invocation, or an application of a predefined predicate, such as equality or other relational operations. Note that the equality predicate has the same semantics as in Prolog — in particular, it can bind values to variables. A compound operation is constructed from primitive operations using the following constructs:

- *Atomic execution:* atomic OpSeq has the same semantics as OpSeq, except that the operations within OpSeq will be executed atomically, i.e., their execution will not be interleaved with the execution of other processes.

- *Alternation:* OpSeq1 || OpSeq2 will succeed if either OpSeq1 or OpSeq2 succeeds. Otherwise, it will fail. Note that OpSeq2 will be executed only if OpSeq1 fails.

- *Parallel composition:* OpSeq1 | OpSeq2 will result in concurrent execution of operations in OpSeq1 and OpSeq2. If either of these sequences succeed, then the entire construct succeeds.

- *if-then-else:* if OpSeq1 then OpSeq2 else OpSeq3 will result in execution og OpSeq1. If it succeeds, then OpSeq2 will be executed, and its success or failure will determine the success or failure of the if-construct. If OpSeq1 fails then OpSeq3 will be executed, and its success or failure will determine the success or failure of the if-construct.

- *Guarded command:* G1 − > OpSeq1 || G2 − > OpSeq2 || · · · || Gn − > OpSeqn has the following semantics. One of the guards that evaluate to `true`, say Gk, will be chosen arbitrarily, and the correponding operation sequence OpSeqk will be executed. If this sequence fails, an alternative guard that evaluates to true will be chosen, and the corresponding to that guard will be executed. If none of the operation sequences (corresponding to guards that evaluate to true) succeed, then the guarded construct fails. Otherwise, it succeeds. For simplicity, we restrict the guards to contain only predefined operations, such as equality checking. They cannot contain method invocations.

- *Loop construct:* loop OperationSequence construct has the meaning that OperationSequence will be executed repeatedly. More precisely, its behavior can be specified using the recursive definition

  loop OpSeq = OpSeq, loop OpSeq

Semantics of operation failure is similar to that of Prolog. When a failure occurs, execution backtracks to the point where an alternative execution path could be taken. All variable bindings made between this point and the point of failure are undone, and execution now proceeds down the alternate path. Note that this implies that (a) the same operation may succeed many times, possibly with different variable bindings, and (b) in an operation such as OpSeq1 || OpSeq2, it is possible for OpSeq1 to succeed, but a subsequent operation that follows alternation may fail, in which case, execution may backtrack to the alternation construct, at which point OpSeq2 may be tried.

Additional details about the language are provided together with the examples below.

2.2 Model of File System

The state of a file system is modelled as a set of tuples of the form `(FileName, Owner, Group, Permissions, Content)`. The file name is represented as a list: a name such as "/a/b/c" would be represented as `[a,b,c]`. The owner and group are represented as integers. The permission field captures the usual UNIX permission information on files. The file content is represented as `normal(C)` for normal files whose content is given by `C`, and `link(F)` for (symbolic) links to another file `F`.

To simplify the presentation, we are not representing directories as files. However, the directory structure is captured implicitly in the way files are named. In effect, this means that information such as directory level permission cannot be represented directly; they must be propagated and represented as permission on files contained in the directory.

The file system behavior is captured by the `fileSystem` class shown below. Note the use of Prolog-style convention: variable names start with a capital letter, while constant, class and function names start with a lower case letter.

The file system supports operations to read and write files. The `write` operation can also be used for file creation, while a `remove` operation is provided for file deletion. File attributes such as ownership and permission can be changed using the operations `chmod`, `chgrp` and and `chmod`. Our file system model does not capture hard links, but symbolic links can be created using the `symlink` operation. All of the file operations make use of a helper function called `resolve` that first resolves symbolic links into real file names, and then performs permission checking.

```
class fileSystem(S) {
   // public methods
   read(File,U,G,C) ::= resolve(File,U,G,read,F1), getContent(F1,C)
   write(File,U,G,C) ::= resolve(File,U,G,write,F1), updateFile(F1,C)
   remove(F,U,G) ::= resolve(File,U,G,write,F1), delete(F1)
   chown(F,U,G,O) ::= resolve(F,U,G,root,F1), chngOwner(F1,O)
   chgrp(F,U,G,O) ::= resolve(F,U,G,owner,F1), chngGroup(F1,G)
   chmod(F, U, G, M) ::= resolve(F,U,G,owner,F1), chngMod(F1,M)
   symlink(L,F,U,C) ::= write(L, U, G, link(F))
```

A number of other helper functions such as getContent and updateFile are used in the above methods. The definition of these helper functions is straightfforward. Of these, we provide the definition of getContent below, while omitting the others in the interest of conserving space.

```
getContent(F, C) ::= member((F,U,G,P,normal(C)), S)
```

The helper function resolve is nontrivial, and we provide its definition below. Its parameters are: the file name F to be resolved, the user and group identifiers U and G with respect to which permission checking needs to be done, and an option argument Opt. It returns the reolved name F1 in the fifth parameter. The resolve method will fail either if the file name does not exist, or if the permission check fails.

```
resolve(F,U,G,Opt,F1) ::=
  resolveLink(F,F2),
  member((F2,O,G1,P,normal(C)), S),// check if tuple is present in set S
    ((U = root) ||                   // no permission checks for root
     ((Opt = owner), (U = O)) ||    // Opt=owner means to check if user U is file owner
     ((Opt = write),
        (((U = O), (P = (S, (R, 1, X), G2, U2))) |     // owner perm
          member(U,G1), (P = (S, O1, (R, 1, X), U1)) | // grp perm
          (P = (S, O1, G2, (R, 1, X))))                // user perm
           then resolveLink(F, F1)) ||
     ((Opt = read), .... ) ||
       .... // other options omitted
resolveLink(F,F1) ::=  if (member((F,U,G,P,link(F2)),S) then resolveLink(F2,F1)
                      else F1 = F
```

The resolve method uses a helper function resolveLink. The latter is a recursive function that keeps following symbolic links until a normal file name is identified. It does this by checking if a file named F is present in the file system. (Note that the state of the file system itself is being captured as a list, and the member is a predicate that is used to search for a specified element in the list.) If F is not present, then resolveLink fails. Otherwise, if the content of F is of the form link(F2), then resolveLink is invoked recursively to follow this link. Otherwise, it must be a normal file and hence F is returned as the resolved name. Note that the definition of resolveLink correctly captures the fact that no permission checking is done for symbolic links.

Once symbolic links are resolved, resolve proceeds to check file permissions. Note that no checking is done if the userid U corresponds to that of root. Otherwise, permission check proceeds, based on the value of option Opt. If the option is read, write or execute, the corresponding permissions are checked. If the option is owner, then the permission check operation needed is one of checking if the userid provided as a parameter is the same as the file owner.

2.3 Model of UNIX Processes

UNIX processes are modelled using a base class called `unixProc` that captures behaviors common to all processes, plus a derived class per program that we wish to model. The state of a `unixProc` object is characterized by its real and effective user/group identifiers, plus information about groups known to the system. In addition, it contains a reference to the filesystem object, and the values of command line arguments. It provides helper functions that correspond roughly to system calls.

```
class unixProc(UID, EUID, GID, EGID, ArgList, FS, UserGroups) {
    read(F,C) ::= FS.read(F,EUID,EGID,C)
    write(F,C) ::= FS.write(F,EUID,EGID,C)
    .
    .                   // definition of several other similar methods is omitted
    .
    run(F,ArgList1) ::= // corresponds to fork+exec(F) in UNIX
      FS.resolve(F,EUID,EGID,exec,F1),
      if FS.resolve(F1,EUID,EGID,setuid,F2)
         then FS.getOwner(F1, EUID1)
      else EUID1 = EUID,
      if FS.resolve(F1, EUID, EGID, setgid,F3)
         then FS.getGroup(F1,EGID1)
      else EGID1 = EGID,
      FS.getContent(F1, program(C)), // F1 must contain a program
      create(C, UID, EUID1, GID, EGID1, ArgList1, FS, UserGroups)
          // create is a language construct that results in creation of
          // a new object belonging to the class of its first argument.
          // The state of the object should correspond  exactly to the
          // parameters supplied to create
    .... // Other methods omitted
}
```

Subclasses of `unixProc` define externally accessible methods, and make use of the methods provided by `unixProc` class. They also need to provide a main function that gets executed as soon as a process is created. The process terminates (and the object destroyed) when the main function terminates.

Based on `unixProc`, we can define an lpr class as follows. At the level of the filesystem, lpr either copies the file to be printed into a spool directory or links it there symbolically, depending upon a command line option.

```
// In addition to usual process parameters, lpr takes 2 arguments:
// the name of the file to be printed, and an option that indicates if this
// file is to be copied to the spool directory before printing, or just
// symbolic-linked from the spool directory.
```

```
class lpr(U, EU, G, EG, [File, Opt], FS, UG): unixProc(U, EU, G, EG, [File,Opt], FS, UG) {
  main() =
    atomic { // N is used to create a temporary name for the spool file
        read([var,spool,lp,count], N),
        write([var,spool,lp,count],(N+1)%1000)
    },
    FS.resolve(File,U,G,read,F1), // accessiblity of File checked for U,
    if (Opt = s)                          // but subsequent operations are
        then symlink([var,spool,lp,N], File) // performed with root privilege
     else read(File,C), write([var,spool,lp,N],C)
    }
}
```

In a similar manner, we can define the behavior of a highly simplified mail receiver/sender as follows. This mail server operates by storing every incoming email message in a spool directory corresponding to the recipient. For simplicity, we model the act of storing in a way that loses previous contents of the spool file.

```
class mailer(FS, UG): unixProc(root, root, sys, sys, [], FS, UG) {
    send(U, M) ::= write([var,spool,mail,U], M)
```

Finally, we model the action of the `comsat` mail notifier program. It looks up the file `/etc/utmp` to identify the terminal where each user is logged in. Whenever a new message is received for a user, `comsat` prints the message on the user's terminal.

We represent the content of the `/etc/utmp` as a list of records. We add additional helper functions to the file system model to support writing and reading from such structured files. Of particular interest is a helper function called `readRec` that allows access to a specific record whose first component is specified as an argument to `readRec`.

```
class comsat(FS, UG): unixProc(root, root, sys, sys, [], fs, ug) {
    main() ::= loop {
                  read([var,spool,mail,Rcvr], Msg)
                  readRec([etc,utmp], Rcvr, Tty),
                  write(Tty, Msg)
               }
}
```

The `loop` construct indicates that the operations inside the loop are executed forever, until the process is killed. These operations make use of an unbound variable `Rcvr.` Such variables are treated as existentially quantified. Operationally, this amounts to binding the variable to an arbitrary value in its domain. Thus, `comsat` nondeterministically chooses some file in the mail directory such that the corresponding user is logged in, and printing the message on the user's terminal. Data-nondeterminism, as captured by the use of such unbound variables, is a key mechanism that simplifies our models.

We now develop a model of a user. The user's behavior is also highly nondeterministic in nature: he/she selects an arbitrary file in the system, and may read this file or overwrite it with arbitrary content. The user may also run arbitrary commands, or send an arbitrary message to an arbitrary user. Arbitrary choice in data values is captured by using unbound variables. The arbitrary choice among the commands is captured by the guarded command construct within the loop. The guarded command construct is within a loop, which indicates that the user will keep performing these actions indefinitely.

```
class user(U,G,FS,UG): unixProc(U,U,G,G,[],FS,UG) {
    main() ::= loop {
                true -> read(F1, C) ||
                true -> write(F1, C) ||
                true -> run(lpr, Args) ||
                true -> mailer.send(U1, M1)
               }
```

Finally, we put all of the classes defined so far into a single system model using a class called init. Note the use of | operator, which denotes parallel composition of multiple processes.

```
class init(FS, UG) {
    main() ::= mailer(FS, UG) | comsat(FS, UG) | user(U, G, FS, UG)
}
```

3 Detecting System Vulnerabilities

In our approach, we use model checking techniques to analyze the behaviors of the system model. In the simplest case, security properties are invariants: properties that must hold at every state of the system. For instance, the simple model described in the previous section does not model legitimate ways to modify the password file (e.g., `passwd`); hence, the constancy of the password file is a desired system invariant. In Section 3.3 we describe how more complex (*temporal*) properties that depend on order of events can be specified.

One of the important features of model checking techniques is their ability to generate counter-examples, which are sequences of states that lead to violation of the given property. In our application, the counter-examples correspond to the steps that an attacker can use to exploit system vulnerabilities. However current model checking tools cannot be used as is, since many components of the system model described in Section 2 have infinitely many reachable states (e.g., the states of the file system). Current model checking tools work mainly with finite-state systems.

Hence we developed a prototype model checker based closely on the XMC system [18], a system that has been developed by one of the authors of this paper. XMC can handle certain classes of infinite-state systems by using implicit representation of state space using constraints.

The XMC system is implemented using the XSB tabled logic programming system [22] by casting the model checking problem as a query evaluation problem. Tabling provides stronger termination properties for XSB in comparison with (untabled) logic programming systems. In particular, computations for solving equations using iterative procedures (e.g., fixpoint iteration) can be programmed very easily, thus making XSB an ideal platform for rapid implementation of program analyzers and model-checkers.

IN XMC, the transition relation of the system model is treated as an external database; the verification problem is solved by running reachability queries over this database. Our prototype follows the same approach. The significant difference is that the reachability queries are made over an infinite transition system in the case of vulnerability analysis. This infiniteness is handled by exploiting the following features of the XSB system.

- Following XMC, we represent states of the system model using terms. The difference in our prototype is that these terms may contain logical (unbound) variables that represent unknown data in the system models.
- We use constraints between terms to relate states in the system model. In the finite-state case handled by XMC, the states are represented by ground (i.e. variable-free) terms, and relation between states is a simple table. In our prototype we relate states, now represented by possibly non-ground terms, using equality constraints among the variables.
- We use the power of tabling in XSB to identify "similar" states (in our case, states that differ only in the names of bound variables) and to reuse computations when a state similar to one encountered earlier is seen again.

In our implementation, we translate the high-level model of the system into a Prolog database (a set of facts) that represents the system's transition relation. As noted in Section 2, our modelling language resembles Prolog in many ways. This factor considerably simplifies the translation algorithm. (In fact, the translation is very similar to that described in [10] for translating processes described in a value-passing process algebra into logic program rules representing the transition relation.) Although the translation algorithm is conceptually simple, its implmentation still requires significant effort. Given the small scale of our models, we have found it easier to perform this translation manually.

In the following, we first assume that the property to be verified is specified as a formula in temporal logic [14]. We then describe the notion of intentions model (see Section 3.3) which alleviates the need to encode complex security properties in temporal logic. The model checking procedure is implemented as reachability query, and is evaluated using the XSB system.

3.1 Model Checking Infinite-State Systems

The infiniteness in the state space of a system arises from two factors— data nondeterminism (infinite branching factor), and execution histories (infinitely long paths)— each of which is handled using a different feature of the model checker.

Infiniteness due to data nondeterminism is handled by term constraints. Recall that data nondeterminism arises from unbound variables in the system model. Term constraints capture the possible values of such variables succinctly. The constraints are represented and manipulated by the XSB system itself, and need no further programming. For instance, consider the problem of verifying whether `/etc/passwd` can be overwritten in the system model in Section 2. Observe from the example that the system can evolve when an arbitrary user chooses to perform a write action of some file, or when a user sends mail. With the logic-programming-based model checker, neither the user nor the message needs to be *bound* to any particular value: we represent these as logical variables. Unification and backtracking automatically generate the cases of interest, by binding the variables only to values that lead to vulnerabilities. For instance, when a user sends mail, the process `comsat` is enabled, which sends a notification (using `write`) to the destination specified in `/etc/utmp`. Note that, at this point, neither the contents nor the permissions on `/etc/utmp` are known. The model checker tries each case in turn, by binding the variables to the needed set of values. If `/etc/utmp` is unreadable or if the required entry (the destination for notifying incoming mail) is not found, no notification is sent and the system reverts back to its original state. On the other hand, if the destination D for notification is present in `/etc/utmp`, then a write to D is issued. Since the contents of `/etc/utmp` are unknown, note that D will be left as a variable. If the destination D can be `/etc/passwd`, then it is indeed possible to change the password file in our model. Thus, the model checking algorithm concludes that if `/etc/utmp` specifies `/etc/passwd` as one of the notification destinations, then it is possible to violate system security.

Infinite execution sequences are handled by *abstracting* the sequences to finite (possibly repeating) segments of a certain kind. Of particular importance is the abstraction that bounds the lengths of sequences. Capturing unknown (or don't-care) values by variables can automatically abstract infinite execution sequences. For instance, consider a user write action to an arbitrary file in the system model in Section 2. This does not constitute "progress" since it does not enable any state change that was impossible before. The lack of progress is easily captured by term constraints. In the state before a write operation to an arbitrary file F, the file's content is represented by a variable, say C_F. In state after the write operation, the file's content is changed to C'_F, which is simply a variant (i.e., identical modulo variable renaming) of the original content. If the effect of write operation is known (say, the new content is α), then the new state is an instance of (i.e., is subsumed by) the old state: hence, no new transitions are possible. Thus we see that progress can be seen as change modulo term subsumption.

The above scenario assumed that nothing is known about the initial state of the system: the files, their contents, the relevant permissions, etc. When the system's initial state is (at least partially) known, a user's `write` action changes the system state; for instance, the constraint that `/etc/utmp` has no reference to `/etc/passwd` may no longer true after an arbitrary `write` action is done, if the access permissions of `/etc/utmp` allow the `write` action to succeed. Thus, the state of the system after an arbitrary `write` action is different from the initial state of the system. The model checker will explore the system evaluation from this state, and can again conclude that there is a potential vulnerability as long as `/etc/utmp` can be modified by an arbitrary user.

Variable abstraction alone is insufficient in general, and we employ approximations that lose information by either ignoring state changes (thus pruning execution sequences), or ignoring conditions on state changes (thus repeating execution sequences). Note that such an abstraction may be "incomplete" in the sense that vulnerabilities in the original model may not be present in the abstract model. However, this limitation is reasonable in our case if we assume that the system vulnerabilities will be exploited by human attackers using their intuition and expertise to come up with attack scenarios. This implies that the sequence of actions that they would perform to achieve intrusion will typically be a short sequence, and thus it may be acceptable to miss out vulnerabilities that require long sequences of actions. Based on this assumption, our method uses a search procedure with iterative deepening, stopping the search after a predetermined depth. The search procedure uses the programming and tabling capabilities of XSB.

3.2 Generating Counter-Examples

The counter-example traces produced by a model checker correspond directly to attack scenarios. Hence the set of all counter-examples can be used to drive intrusion detection. Note that, even in the finite-state case, it is infeasible to enumerate all possible counter-examples. To overcome this problem, we avoid an explicit enumeration of all counter examples, instead choosing to represent them using a finite-state automaton. The automaton represents the set C of counter-examples, such that each example $c \in C$ corresponds to a path in the automaton. The automata representation is succinct and can be used directly for intrusion detection. Moreover, such an automaton can be constructed by inspecting the memo tables built during model checking: the table entries form the states of the automaton and the dependencies between the entries form the transitions.

The automata-based representation of counter-examples extends naturally to the case of infinite-state systems as well. In this case, each state in the automaton is associated with a set of variables, while the transitions specify conditions on their values. Such automata have the ability represent *generic* counter-examples: those that are parameterized with respect to specific system configurations. We can generate such counter-examples by leaving the initial state of the system unbound and using data nondeterminism to lazily binding the state variables, as explained earlier with the `comsat` example. The automata representing these generic examples can then be instantiated for particular system configuration parameters to check for vulnerabilities. Thus the automata themselves are generic with respect to configurations. However, the automata must be regenerated when system's capabilities change, e.g., when new services are added.

Note that in general, for infinite-state systems, there may be no finite representation for the set of all counter-examples. However, note that the abstractions we use bound the length of counter example sequences, thereby making it possible to find a finite representation for the set of all counter examples.

3.3 Beyond Invariant Properties

In the comsat example explained earlier, the property of interest was an invariant. In general, however, one would be interested in path properties. For instance, there may be a password changing program passwd on a system that allows a user to modify his/her password, and thus change the contents of the password file. Clearly, execution paths where the password file is changed by the passwd program, or by a system administrator, do not correspond to any vulnerabilities.

Path properties can be encoded in temporal logic [14]. They can eliminate "degenerate paths" such as those where the superuser changes the password file or it is changed by the passwd program. This is done by adding antecedents to the original safety property that are violated by such degenerate paths.

A problem that arises in the context of vulnerability analysis is that the description of degenerate paths tends to become very large, since there are many degenerate cases. For instance, there may be many different ways in which a superuser can change the password file: by overwriting it, by using an editor, by using the passwd command, etc. Enumerating all such degenerate paths is impractical since the temporal logic formula becomes very large and difficult to understand, and hence is likely to contain errors.

To address this problem, we propose the following approach where the original safety property is left unchanged. In order to eliminate degenerate paths, we develop a second model called the *intentions model.* An intentions model captures the intended outcome of executing every program. These intentions are stated in terms of the files that may be written or executed in the course of executing the program. The system model has vulnerabilities if it contains paths to unsafe states for which there exists no corresponding path in the intentions model.

For example, an intention model of mail daemon would be that it writes files in the directory /usr/spool/mail. The intention model of lpr would be that it writes files in the directory /usr/spool/lp. The intention model of passwd program would be that it writes /etc/passwd file. The intention model, by default, will refer to normalized file names, which correspond to an absolute file names that are not a symbolic links. This would be appropriate in the case of mail daemon and lpr. Situations where symbolic links are permitted, will be made explicit in the intentions model. For instance, an intention model of cp program will state that it will overwrite a file provided as an argument, regardless of whether it is a symbolic link or not.

When an intentions model is used, the model checker must disregard the "intended paths," i.e., paths where every action is intended is also in the intentions model. A simple way to do this is to leave the model checker unchanged, but prune away paths from the counter-example automaton. Clearly, more efficient techniques to eliminate intended paths can be developed, and is a topic of current research.

4 Analysis Results

4.1 Vulnerabilities due to `comsat`

Given the simple model of a UNIX system described in Section 2, our current implementation identifies the following vulnerabilities that would ultimately enable the password file to be overwritten. The vulnerabilities are presented in the format

```
when <condition>
scenario <exploit>
```

where `condition` specifies the configuration parameters under which the `exploit` is possible.

Our prototype implementation identifies several vulnerabilities in the system, some of which are shown below. The first one is trivial, and it corresponds to the case when the password file is world-writable. Nevertheless, it is interesting to note that the use of data-nondeterminism, and its implementation using term-constraints, enables us to derive this scenario, even when the `/etc/passwd` file is not mentioned in the model or in the initial state of the file system.

```
when   FS.resolve([etc, passwd],U1,G1,write,F1)
scenario
       [user(U1,G1,FS,UG).write([etc, passwd], M)]
```

The way to read this is as follows: if there exists a file `/etc/passwd` that is writable by some user `U1` belonging to a group `G1`, then this user can write the password file with arbitrary data `M`.

The second vulnerability identified is the `comsat` vulnerability described in the introduction. It happens when a user `U1` has permission to write the`/etc/utmp` file. Note again that this vulnerability was identified, even when the model checker was provided no information about the original state of the system. By virtue of the way term constraints are handled in our model-checker, it is able to infer the appropriate relationships that must hold between the contents of the `/etc/utmp` file and the password file, as well as the file permissions that must hold for the vulnerability to be exploited.

```
when
   FS.resolve([etc, utmp],U1,G1,write,F1)
scenario
   [user(U1,G1,FS,UG).write([etc, utmp], (U2, [etc, passwd])),
    user(U3,G3,FS,UG) invokes mailer.send(U2, M),
    comsat.read([var, spool, mail, U2], M),
    comsat.readRec([etc, utmp], U2, [etc, passwd]),
    comsat.write([etc, passwd], M)]
```

Note that although the scenario mentions three users `U1`, `U2` and `U3`, they are all existentially quantified — which means that they can all be the same user.

The third vulnerability is similar to the first one, but the user directly overwrites the spool file instead of using the mailer to update the spool file:

```
when
   resolve([etc, utmp],U1,G1,write), resolve([var, spool, mail, U2],U3,G3,write)
scenario
   [user(U1,G1,ug).write([etc, utmp], (U2, [etc, passwd])),
    user(U3,G3,ug).write([var, spool, mail, U2], M),
    comsat.read([var, spool, mail, U2], M),
    [etc, passwd] = comsat.readrec([etc, utmp], U2),
    comsat.write([etc, passwd], M)]
```

Another attack scenario is an interesting variation on the previous attack, and does not require write permission to `/etc/utmp`. It brings together two known exploits, one involving the use of symbolic links and the other being the `comsat` vulnerability mentioned above. Although we had developed the models ourselves, we had not realized that our model contained this vulnerability. It is noteworthy that in spite of the simplicity of the models used, our model checking procedure identified vulnerabilities that were unknown to us.

```
when
   FS.resolve([var, spool, mail, U2],U1,G1,write,F1)
scenario
   [user(U1,G1,FS,UG).symlink([var, spool, mail, U2], [etc, utmp]),
    user(U3,G3,FS,UG) invokes mailer.send(U2, (U4, [etc, passwd])),
    user(U5,G5,FS,UG) invokes mailer.send(U4, M),
    comsat.read([var,spool,mail,U4], M),
    comsat.readrec([etc, utmp], U4, [etc, passwd]),
    comsat.write([etc, passwd], M)]
```

4.2 Vulnerabilities due to `lpr`

Before analysis, we abstracted the system model for `lpr` by making the temporary spool file name to be a constant (i.e., making the counting modulo 1 instead of 1000). The combination of symbolic links and the standard spool file naming convention introduces the following vulnerability:

```
when
   FS.resolve(F1,U1,G1,write,F2),
   FS.resolve([etc, passwd],U2,G2,read,F3),
   FS.resolve(F1,U3,G3,read,F4),
   FS.resolve(F5,U3,G3,exec,F6),
   FS.getContent(F6,program(lpr))
scenario
   [user(U1,G1,FS,UG).write(F1,C1),
    user(U2,G2,FS,UG).run(lpr, [[etc, passwd],s]),
    user(U3,G3,UG).run(lpr, [F1])]
```

Since we start with an initial state that corresponds to an unbound variable, there are no files that can be printed in the initial state. The scenario shows that such a file can be created, and later read. It also requires read permission on the password file.

5 Concluding Remarks

In this paper, we presented a new model-based approach for analyzing configuration vulnerabilities. Whereas previous approaches relied on expert knowledge to codify causes of configuration vulnerabilities, this step is not necessary in our approach. Consequently, our approach can not only identify previously exploited vulnerabilities, but also discover new ones that have never been exploited. Our examples demonstrate this capability to discover such vulnerabilities: knowledge about vulnerabilities were never encoded into the system model, but our analysis was able to detect the vulnerabilities. It even identified vulnerabilities whose presence in the system model was unknown to us.

The results of our analysis can be used in many ways. The first and obvious use is in reconfiguring the system to eliminate the vulnerabilities identified by model-based analysis. The reconfigured system can be reanalyzed to ensure that (most) vulnerabilities have been eliminated. A second use is to feed the counter-examples generated by our analysis into an intrusion detection system. The intrusion detection system can now identify all attempts to exploit the vulnerabilities identified by our analysis, and may be able to prevent them from succeeding. A third way to use our analysis is to begin with minimal information about the initial state of the system, in which case our analysis generates assumptions about the initial system that lead to vulnerabilities. These assumptions correspond to the "vulnerability causes" that can be encoded into configuration checkers such as COPS and SATAN.

The main challenge in using the approach presented in this paper is one of scale. Although our model checker can easily handle the models described in this paper, more realistic system models will be much larger, making it significantly harder to perform an accurate analysis. However, we believe this is a temporary difficulty: some of the authors of this paper, as well as a number of other researchers, are developing better and better model checkers that are able to handle larger and larger systems. A second challenge is the effort required for developing models. We are investigating source code analysis techniques that can help automate the model generation process.

References

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

[2] R. Baldwin, Rule based analysis of security checking. MIT LCS Technical Report No. 401, 1988.

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

[4] M. Bishop and B. Bailey, A critical analysis of vulnerability taxonomies, TR CSE-96-11, Dept. of Comp. Sci., University of California at Davis, 1996.

[5] W. Chen and D.S. Warren, Tabled evaluation with delaying for general logic programs, Journal of the ACM, 43(1):20–74, January 1996.

[6] E. M. Clarke and E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, Proceedings of the Workshop on Logic of Programs, LNCS 131, 1981.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM TOPLAS*, 8(2), 1986.

[8] E. M. Clarke, K. McMillan, S. Campos, and V. Hartonas-GarmHausen, Symbolic model checking, Computer Aided Verification'96, pages 419–422.

[9] E. M. Clarke and J. M. Wing, Formal methods: State of the art and future directions, ACM Computing Surveys, 28(4), December 1996.

[10] Y. Dong and C. R. Ramakrishnan, An Optimizing Compiler for Efficient Model Checking, Proceedings of FORTE/PSTV, 1999.

[11] D. Farmer and E. Spafford, The COPS Security Checker System, CSD-TR- 993, Department of Computer Science, Purdue University, 1991.

[12] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[13] G. J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering, 23(5):279–295, May 1997.

[14] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1991.

[15] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[16] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In Proceedings of the International Symposium in Programming, volume 137 of LNCS, 1982. Springer-Verlag.

[17] C.R. Ramakrishnan and R. Sekar, Model-Based Vulnerability Analysis of Computer Systems, 2nd Int'l Workshop on Verification, Model Checking and Abstract Interpretation, 1998.

[18] C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury and V.N. Venkatakrishnan, XMC: A Logic-Programming-Based Verification Toolset, Computer Aided Verification (CAV), 2000.

[19] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Terrance Swift, S.A. Smolka, and D.S. Warren, Efficient model-checking using tabled resolution, Proceedings of CAV'97, June 1997.

[20] R. Ritchey and P. Ammann, Using Model Checking to Analyze Network Vulnerabilities, IEEE Oakland Symposium on Security and Privacy, 2000.

[21] H. Tamaki and T. Sato, OLDT resolution with tabulation, International Conference on Logic Programming, pages 84–98. MIT Press, 1986.

[22] The XSB logic programming system v2.1, 2000. Available from `http://www.cs.sunysb.edu/~sbprolog`.

[23] D. Zerkle, K. Levitt, NetKuang–A Multi-Host Configuration Vulnerability Checker, Proc. of the 6th USENIX Security Symposium. San Jose, California, July 22-25, 1996, pp. 195-204.