Model-Based Analysis of Configuration Vulnerabilities

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.

This approach can be used to automatically detect known and as-yet-unknown vulnerabilities. This is in contrast with approaches such as those used in COPS and SATAN, which mainly address previously exploited vulnerabilities. Another advantage of our approach is that it is modular. For instance, to identify system vulnerabilities after addition of a server program, we only need to develop a model for the new server; the reanalysis of the system is done automatically.

Traditional model checkers can analyze only finite-state systems. Finite-state models cannot capture components such as file systems where files can be added, renamed or removed. Hence finite-state techniques cannot be used to detect vulnerabilities that depend on file names, contents, or directory structures. In our approach, we permit infinite-state models by developing an alternative model checking technique that exploits the nature of vulnerabilities. We demonstrate the usefulness of this technique by showing how it detects nontrivial vulnerabilities in a simplified model of a Unix system.

Contents

1 Introduction **1**
1.1 Vulnerability Analysis: State of Art vs. New Approach 1

2 Modeling Security-related Behaviors of Systems **3**
2.1 Modeling Language 3
2.2 Model of File System 3
2.3 Model of UNIX Processes 4

3 Detecting System Vulnerabilities **6**
3.1 Model Checking Infinite-State Systems 7
3.2 Generating Counter-Examples 8
3.3 Beyond Invariant Properties 8
3.4 Vulnerabilities in Distributed Systems 9

4 Analysis Results **9**
4.1 Vulnerabilities due to `comsat` 9
4.2 Vulnerabilities due to `lpr` 10

5 Summary and Future Work **10**

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 the mail notification program `comsat`. This program waits for reports of incoming mail for any user and prints the first few lines of the message on his/her terminal. This terminal is determined from the file `/etc/utmp`, which was configured to be world-writable. A malicious user could modify this file by substituting the `/etc/passwd` in the place of the terminal that he/she is logged on. The user then sends mail to self containing a line that starts with `root::0:0:`. Of particular significance here is that the second field in this line, which corresponds to the password field, is empty, which implies that the superuser has no password. The `comsat` program promptly overwrites the password file with the message. The user can now login as `root` without providing a password. This vulnerability is not the result of an error in any one component, but arose from interactions among several components. The `comsat` program assumed that the contents of `/etc/utmp` were trustworthy, but this implicit assumption was violated by the other components.

Formal methods are a good choice to address software engineering problems of the kind described above. (See [7] for a survey of recent successes in using formal methods for building safety-critical systems.) In this paper, we propose a formal methods based approach for analyzing system configuration vulnerabilities. Our approach is aimed at detecting interactions among multiple system components that lead to vulnerabilities, rather than errors in individual programs. It 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 the model extraction process be automated, at least partially, using existing 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.* Given a formal model of system components, we can derive their composite behavior. This behavior can be analyzed to check if it violates the desired security objectives. Model checking [4, 13, 5] is a popular verification technique that can be used to for this purpose. 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.

The key issues in developing such an approach are: (a) development of languages that simplify the development of abstract and accurate models of component behaviors, (b) development of appropriate model-checking techniques that are efficient enough to handle realistic systems. The second issue is particularly challenging, since the models are in general infinite, and hence well-known model-checking techniques are not applicable. We propose a solution to these problems in this paper. We demonstrate, using a simple model of UNIX system, that nontrivial vulnerabilities can be detected automatically using our approach.

1.1 Vulnerability Analysis: State of Art vs. New Approach

Existing approaches [8, 3, 16] for analyzing configuration vulnerabilities can be broadly characterized as *rule-based*, i.e., they employ a set of rules that enumerate known causes for vulnerabilities. The tools then systematically check the system configuration to identify if these causes are present in the system. For instance, a world- or group-writable `.login` file is a well known vulnerability that enables one user to gain all access privileges of another user. Widely used tools, such as COPS and SATAN search for occurrences of such known vulnerabilities [8]. Generation of the rules relies on expert knowledge about interactions among many components of the

system. Unfortunately, few experts have a complete understanding of the interactions among all components of modern computer system. Issues such as race conditions, many possible interleavings and hidden assumptions 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 our approach are:

- *Identification of known* **and** *unknown vulnerabilities.* Our model-based approach has the ability to identify known and *as-yet-unknown* vulnerabilities. In contrast, the rule-based approaches are limited to examining the system for known vulnerabilities.
- *Modularity of approach.* In our approach, 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.* When vulnerabilities are identified by our analysis, we may sometimes be able to rectify them. Other times, there may be no immediate fix, as it may require changes to vendor-provided software, or since the changes may interfere with legitimate functionality of the system. A second line of defense for vulnerable systems is *misuse intrusion detection*, where system use is monitored in order to detect known exploitations. The exploit scenarios are usually specified by an expert, while the process of detection of exploits is automated. The approach outlined in this paper enables automation of the first task as well, since the counter examples generated by our model-checking technique correspond to exploits. We can mechanically translate these exploits into patterns for a misuse intrusion detection system. An important benefit of this approach is that the intrusion detection system can recognize any action that threatens to exploit a system vulnerability, and not just those that are *known* (by past experience or expert knowledge) to be potentially dangerous.
- *Automatic generation of rules for checking vulnerabilities of specific configurations.* To accomplish this, we perform perform model checking with incomplete information about initial system configuration. For instance, we may leave out information about initial contents of most parts of the filesystem. Our model-checking technique will then produce vulnerability scenarios that are conditional upon the (unspecified) initial configuration. Each of these conditions then capture a potential vulnerability that can be checked using a rule. The main benefit of using these rules is that they can be checked much more economically (in terms of time and memory usage), than running the model checker.

This paper builds on some preliminary results on model-based vulnerability analysis we had reported earlier in [2]. Since then, Ritchey and Ammann [14] 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 [6] and SPIN [10] to perform vulnerability analysis. In contrast, we need to develop model checking techniques that can be used for vulnerability analysis of 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.

2.1 Modeling Language

The choice of modeling language is important: a language with constructs appropriate for a domain will promote clear, concise and accurate models, while the use of an inappropriate language can lead to introduction of errors or artifacts into the model.

Our modeling language is based on concepts from CSP [9], CCS [12], and object-oriented languages, and is designed to support a distributed object paradigm. In this language, a system is modelled as a collection of communicating processes. Each process is viewed as an object (in the sense of OO-languages). Its internal state is encapsulated, and cannot be accessed by other processes. Operations can be invoked on an object by sending messages to it on its input channel. Operation invocation is synchronous: it causes the invoking process to block until a return value is sent back at the conclusion of the operation. The communication channels are not explicitly mentioned in the language, thus the syntax of operation invocation looks exactly like method invocation syntax in OO-languages.

Basic data types supported in the language include integers, floats and strings. Compound types (such as those needed for representing file systems), as well as several other features of the language, are similar to Prolog. The language supports an algebraic datatype system, similar to Prolog. Two of the most common compound types are tuples and lists.

Like Prolog predicates, 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 function invocations.

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. Multiple statements that must be executed atomically are specified by enclosing the statements with the `atomic` construct.

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 `normal(C)` for normal files whose content is given by `C`, and `link(F)` for links to another file `F`.

To keep the model small, 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 directly as permission on files contained in the directory.

The file system behavior is captured by the following class. 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. Note also that the comma operator is used to denote sequencing of operations, with the semantics that subsequent operations in a sequence are executed only if all of the preceding operations return *true*.

```
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))
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;
```

Most of the public methods have their obvious meanings. The `resolve` method resolves (symbolic) link names into names of ordinary files, and performs permission checking.

All of the public methods above have been defined in terms of helper functions such as `getContent`, `updateFile`, etc. The definition of `getContent` is provided here, while the others are omitted due to space limitations:

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

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 base class provides helper functions that correspond roughly to system calls. The advantage of using this approach is that common errors in specification, such as the use of incorrect process parameters (e.g., userid or effective userid) is significantly reduced. 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. In the class unixProc below, note the use of the `let` construct to introduce new variables.

```
class unixProc(UID, EUID, GID, EGID, ArgList, FS, UserGroups) {
    // This class provides only private methods (accessible to subclasses)
    read(F,C) ::= FS.read(F,EUID,EGID,C)
    write(F,C) ::= FS.write(F,EUID,EGID,C)
    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 an 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, G,FS,[File, Opt],UG): unixProc(U,root,G,sys, [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 extend the `filesystem` and `unixProc` classes presented above to support such structured files. Of particular interest is an operation 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, which uses the syntax `Guard1 -> Body1 | ··· | GuardN->BodyN`.

The guarded command construct is within a loop, which indicates that the reader 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, U, U, G, G, Args, FS, UG) |
                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, certain aspects of vulnerability analysis make current model checking techniques unusable, as explained below.

- *Infinite-state models:* Many components of the system model described in Section 2 are have infinitely many reachable states (e.g., the states of the file system). Current model checking techniques work mainly with finite-state systems.
- *Counter-examples:* Not all vulnerabilities may be fixable (e.g., bug-fixes to vendor software). In such cases, we use intrusion detection techniques for thwarting any attempt to exploit the vulnerabilities. Hence, we need to identify *all* potential vulnerabilities. This is in contrast to traditional design-time applications of model checking where it suffices to find any one error.

We develop a customized model checker that exploits the characteristics of vulnerability analysis to addresses the above problems. We use the XSB tabled logic programming system [15] for rapidly prototyping our model checker. The following features of the XSB system makes it an attractive vehicle to prototype our model checker:

- *Tabling:* The XSB system maintains memo tables to remember previously invoked queries (calls) and their answers. Tabling ensures that the fixed point computations needed in a model checker terminate for finite models.
- *Term Constraints:* The XSB system provides logical variables that conform to the semantics of the unbound variables used to model data nondeterminism in our modeling language (see Section 2). Furthermore, the unification mechanism in XSB permits us to represent and manipulate equality constraints over terms; such constraints can be naturally used to represent infinite sets of states.
- *Programmability:* Being a logic programming system, XSB permits one to easily specify interpreters and metaprograms: a feature that is exploited for constructing abstract system models for systems where infiniteness stems from unbounded execution.

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.

Although an algorithm for translating our models into Prolog has been developed, we have not implemented this algorithm at the time of this writing; instead, we hand-translated the model in Section 2. In the following, we first assume that the property to be verified is specified as a formula in temporal logic [11]. 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 an interpreter, 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.

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 [11]. 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.

3.4 Vulnerabilities in Distributed Systems

Since we model the system as a composition of system of concurrent processes, our analysis can be readily used, *without modification*, to detect vulnerabilities in distributed systems. Consider the following example of vulnerability arising from misconfiguring a distributed system. A user on host `A`, `bob`, who was helping with installation of beta software was included in `sys` group, but then the group association was not removed after the installation tasks were done. The configuration scripts for `A` are such that `/etc/rc.d/rc` executes the commands in `/etc/sysconfig`. The permissions on `/etc/sysconfig` are such that it is writable by members of `sys` group. Using a model that captures the execution behavior of boot-time processes and scripts, our analyzer will conclude that members of `sys` can usurp root privileges on host `A`. Assume that the home directories of system staff are exported to `A` from NFS server `B`. By `su`-ing to some system staff's account, say `sally`, `bob` can modify the `sally`'s `.login` file, and hence gain access to any machine to which `sally`'s home directory is exported by `B`. Again, the our analyzer can trace through such an scenario based on the execution models for login processes and the rules governing NFS exports.

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. As mentioned earlier, the configuration parameters can be used to develop rules (or code) that can be fed into configuration checking tools such as COPS and SATAN.

Our prototype system identifies several vulnerabilities in the system, three 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 treatment using term-constraints, enables us to derive this scenario, even when the `/etc/passwd` file is not mentioned in the model.

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

The second 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 also identified, even when the model checker was provided no information about the original state of the system.

```
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)]
```

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 contains 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 Summary and Future Work

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.

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 key technical contributions of this paper are (a) a modelling language that is suitable for modelling behaviors of system components such as the operating system, privileged programs, etc, (b) development of specification and verification techniques that can analyze models developed in this language. We developed a highly simplified model of a UNIX system and a few select programs in our modelling language, and showed that our verification technique can identify nontrivial vulnerabilities in this system. Our model checker was able to identify these vulnerabilities in spite of the fact that our models represent infinite-state systems.

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. So far, we have been able to extract reasonable models from shell scripts, such as boot scripts and login scripts, and are now investigating how these techniques can be extended to more complex programming languages such as Java or C++.

References

[1] 8lgm Security Advisories, `http://www.8lgm.org/advisories-f.html`.

[2] Names left out for anonymous refereeing, Model-Based Vulnerability Analysis of Computer Systems, 2nd Int'l Workshop on Verification, Model Checking and Abstract Interpretation, 1998.

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

[4] 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.

[5] 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.

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

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

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

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

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

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

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

[13] 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.

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

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

[16] 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.