REMEDIATION GRAPHS FOR SECURITY PATCH MANAGEMENT

Vipin Swarup
The MITRE Corporation, 7515 Colshire Drive, McLean, VA 22102, USA
swarup@mitre.org

Abstract Attackers are becoming proficient at reverse engineering security patches and then creating worms or viruses that rapidly attack systems that have not applied the patches. Automated patch management systems are being developed to address this threat. A central function of a patch management system is to decide which patches to install on a computer and to determine the sequence of actions that will cause them to be installed.

In this paper, we introduce the notion of a *patch remediation graph* that depicts ways in which a system can be patched so that the system eventually reaches a desired, secure state. We present a language for specifying patch dependencies, patch conflicts, and user preferences for patch configurations. We then present efficient algorithms that construct and analyze remediation graphs from current computer configurations and various patch constraints. Our analysis algorithms use the graphs to compute maximal preferred configurations and sequences of patch actions that, if executed, will transition computers safely to those configurations.

Keywords: Security patch management, software configuration management, risk management, model checking.

1. INTRODUCTION

Security patches have become the primary technique for closing software vulnerabilities that are discovered after software systems have been deployed. Unfortunately, security patches also reveal details of the underlying vulnerabilities. Attackers are becoming proficient at using this information to create worms or viruses that rapidly attack systems that have not applied the patches.

Several patch management systems (e.g., Microsoft/SMS [9], Radia Patch Manager [10], etc.) have been developed to combat this threat. These systems include features from sophisticated software version man-

Y. Deswarte et al. (eds.), *Security and Protection in Information Processing Systems*

agement, software configuration management, and (post-deployment) software maintenance management systems [2, 5]. Software version, configuration, and maintenance management systems are designed to assist software developers in developing, composing, and maintaining large, evolving software systems. They provide features such as storage of multiple versions of a software package, retrieval of any identified version, tracking changes to a version, and notifying users affected by a change. Patch management systems extend this functionality by including sophisticated system inventory modules that determine the current state of a computer system, configuration management modules that compute patches that may be installed on a system, and secure patch deployment modules that deploy and install the patches in a secure manner [3, 4].

However, while patch management systems such as Microsoft/SMS provide numerous sophisticated capabilities, they still expect human administrators to make critical decisions regarding which patches to install. In particular, they only support coarse-grained patch management policies, e.g., vendors may mark certain patches as being critical, and administrators may designate certain patch sets as being reference configurations. However, no support is provided to represent the relative desirability of patches, or to help an administrator decide which patch sets to designate as preferred configurations, or for end-users to express their policy requirements regarding security patches (especially regarding patches which are not "recommended" or "critical"). For instance, a user may wish to specify that a certain application is critical and takes precedence over non-critical security patches which conflict with the application.

A second problem arises when a computer has not been maintained in an up-to-date patched state. In this case, when an administrator seeks to update the configuration, he must install numerous patches in order to take the computer to a recommended configuration. For instance, Sun Solaris 8 currently has 113 security patches of which 81 are in the Recommended Patch List. An administrator must first select some subset of these patches as the target configuration and the patch management system will then install the patches sequentially (the system will manage all dependencies and conflicts). However, since the patches are installed sequentially, the computer may pass through states that are insecure. This temporary vulnerability is important since the patch management system might fail or be suspended while the system is in the vulnerable state. Thus, the sequence of patch installations should be selected to minimize such vulnerable states, and the administrator should be informed of any potential vulnerabilities.

In summary, existing patch management systems do not answer two questions that are important to the security administrator of an individual computer: (1) What is the most desirable set of patches that should be installed on a computer? (2) What sequence of patch installation and removal actions will provide a safe transition path to the desired state? In this paper, we present an abstract model and practical algorithms that answer these questions for heterogeneous, distributed computers. We use a preference relation to model the relative desirability of patches. We define and build a patch remediation graph which permits us to analyze properties of patch installation sequences, e.g., whether a sequence includes vulnerable states. Note that since most subsets of patches are valid configuration states, the number of states and the number of possible patch installation sequences is enormous (i.e., the size of the remediation graph is exponential in the number of patches). Hence, we rely on model checking to manage the state explosion problem. We emphasize here that we do not address the practical problems of system inventory analysis, patch deployment and installation, etc. in this paper as these functions are present in current patch management products. Rather, we focus on the novel problem of remediation analysis as expressed by the previous two questions.

The remainder of this paper is organized as follows. Section 2 presents an abstract model for patch configuration management while Section 3 presents algorithms for patch remediation. Section 4 compares our approach with related work. Section 5 concludes this paper.

2. SECURITY PATCH MANAGEMENT: MODEL

Several models of software configuration management exist. Version models [2] label each distinct version of a software product and relate the different versions using labeling conventions or graphs. Change models model a software product in terms of a single base version and a series of changes (e.g., software patches). Hybrid models include attributes of both version models and change models. However, these models do not typically guide system administrators in selecting appropriate versions and changing a system's state from one version to another. In this section, we present an abstract security patch management model that permits a security administrator to reason about the desirability of possible patch configurations and to determine how to transition a system securely from one configuration to a preferable configuration.

2.1 Security Patch Management

We model the patch configuration state of a system in terms of a baseline configuration and a set of changes (e.g., security patches). We represent changes by a set of patches rather than a sequence of patch installation and removal actions; however, our model and algorithms can be generalized to handle patch sequences.

If $\mathcal{B}$ is a finite set of *baseline system configurations*, and $\mathcal{P}$ is a finite set of *security patches*, then $S = \mathcal{B} \times 2^{\mathcal{P}}$ is the set of all possible *patch configuration states*. If p is a patch and $s = \langle b, P \rangle$ is a state, then we will write $p \in s$ to denote that $p \in P$.

Let $\tau \subseteq S \times S$ be a *state transition relation*. τ represents the state change caused by installing or uninstalling a single security patch. Patch installation is often state-dependent, e.g., a patch may only be installed in states in which a specific version of an application is installed while certain other patches are not installed.

A *patch action graph* $G = \langle S, \tau \rangle$ is a finite directed graph whose vertices are the patch configuration states in S and whose directed edges are given by τ. We say that a state s' is *reachable* from state s if there is a directed path in G from s to s'.

Security patches typically have consistency constraints, e.g., two patches conflict with each other and must never be installed together in the same state, or one patch must always be installed together with another. We represent these in terms of a consistency predicate $V \subseteq S$, i.e., V defines the set of *consistent* states.

Due to the consistency constraints, it is not possible to reach the most secure state simply by installing all available patches. Rather, an administrator must make a choice regarding which subset of patches to install. We represent this choice using a *preference relation* $<\subset S \times S$ which is a quasi order over S, i.e., $<$ is irreflexive and transitive.

Informally, *patch remediation* is the process of transitioning a system from an initial state to a preferred, consistent state by performing a sequence of patch actions. We will present an algorithm to compute consistent states that are reachable from a given initial state and are maximal under $<$. We elaborate on this in the remainder of this section.

2.2 Patch Configuration States

There are several well-described inventory tools that collect information regarding the baseline software products installed on a computer as well as the patches that have been installed thus far. These tools are specific to a computer's operating system. For instance, tools for Windows-based systems may determine configurations by examining the

Windows Registry of each system. Tools for Unix-based systems may determine which applications are currently installed by using one of several techniques. One technique is to compute hashes of all executable files in certain directories and to compare those hashes with the standard hashes published by software vendors. Another technique is to look for patterns in audit records and to compare those patterns with standard patterns caused by the presence of an application.

Popular operating systems do not maintain adequate information to precisely define a specific patch configuration state. In particular, operating systems do not typically maintain the entire sequence of patch installs and uninstalls from the base version of an application. Rather, they only maintain a set of all patches currently installed on the computer. Hence, in this paper we model a computer's state as a set of patches, together with a baseline configuration that includes all the software products installed on the computer.

2.3 Patch State Consistency

Consistency constraints for patch states are properties that states must satisfy in order to be compliant with operational security policy. They can be defined in terms of the following three predicates on patch states:

(`critical` p) **:** This specifies that the patch p must be installed in all consistent states. That is, (`critical` p)$(s) = (p \in s)$. For instance, organizations typically mandate that all computers connected to the organizations' networks must have certain critical security patches installed. This can be expressed using `critical` constraints.

(p `conflicts-with` q) **:** This specifies that patches p and q cannot both be installed in any consistent state. That is, (p `conflicts-with` $q)(s) = (p \notin s) \vee (q \notin s)$. For instance, this can express the adverse interactions that a security patch may have with another patch or application, and indicates that they must not be installed together.

(p `requires` q) **:** This specifies that patch p is installed in a consistent state only if patch q is also installed. That is, (p `requires` $q)(s) = (p \in s) \supset (q \in s)$. For instance, this can express the typical constraint that a set of patches must be installed together. This is so common that we introduce a derived construct to express it: (`coexists-with` $p_1 \ldots p_n$) is defined to mean that (p_i `requires` p_j) for all $1 \leq i, j \leq n$.

For instance, the following is an excerpt of the patch consistency constraint specification for Sun Solaris OS 2.3 (these patches have been superceded and are now obsolete):

(critical 101318)
(critical 101782)
(critical 102167)
(critical 103705)
(101318-32 conflicts-with 101524-01)
(101362-51 conflicts-with 101262-21)
(101782 requires 101318)
(102903 requires 101318)
(102932 requires 101318)
(101782 requires 101359)
(102167 requires 101359)
(103705 requires 101359)

A patch state s is consistent if it satisfies all the patch consistency constraints in the specification.

2.4 Patch State Transitions

A state transition represents the state change caused by installing or uninstalling a single security patch. Patches are specified as being applicable only to systems with a specific baseline configuration installed and a set of prerequisite patches already installed. Several patches also require the absence of conflicting patches. A state transition relation is specified as a set of rules. Each rule is a ground instantiation of one of the following rule templates with b^* being a sequence of baseline configurations, p^* and q^* being sequences of patches, and r being a patch. We write $r.p^*$ to denote the sequence whose first element is r and whose remaining elements are those in p^*.

```
(if (and ( baseline-in b*)
         ( patches-present p*)
         ( patches-absent q*))
    ( install-patch r))

(if (and ( baseline-in b*)
         ( patches-present r . p*)
         ( patches-absent q*))
    ( remove-patch r))
```

Each rule has a condition and a body. If the state satisfies the condition, then the rule is applicable and has the effect of either adding or removing the patch r from the state's patch set. The condition has three clauses. For a state $s = \langle b, P \rangle$ to satisfy the condition, its baseline configuration b must be in b^*, all patches in p^* must be in P, and the patches in q^* must not be in P. Note that the `remove-patch` rule also requires the patch r to be installed in the state. The state transition relation τ is given by the union of all specified rules.

2.5 Patch Preference Relation

A preference relation $<\subseteq S \times S$ is a quasi order (i.e., an irreflexive and transitive) relation between configuration states that represents the "desirability" of configurations. Preferences may be provided by software vendors (e.g., when a vendor recommends that certain patches be installed), by security administrators (e.g., when an administrator recommends that certain patches not be installed due to site-specific conflicts), or by end-users (e.g., when a user specifies that it is not desirable to install patches that conflict with mission-critical applications).

(**preferable** p^*)**:** This returns a quasi order $<$ in which states with a larger set of patches in p^* are preferable to states with a smaller set. That is, if $s = \langle b, P \rangle$ and $s' = \langle b', P' \rangle$, then $s < s'$ if $(p^* \cap P) \subset (p^* \cap P')$. For instance, this can be used to specify recommended sets of patches as typically published by vendors. Note that if p^* consists of a single patch r, this returns a quasi order $<$ in which states with patch r are preferable to states without patch r.

(**seq** $<_1$ $<_2$)**:** This returns a quasi order $<$ that first compares two states using $<_1$, and if they are incomparable then compares them using $<_2$. That is, $s < s'$ if either ($s <_1 s'$) or if (s and s' are incomparable under $<_1$ and ($s <_2 s'$)). This construct is used to compose multiple quasi orders in order to construct a single preference relation.

2.6 Remediation

We can now define what we mean by patch remediation.

DEFINITION 1 *Let S be a set of patch configuration states, and let $\tau \subseteq S \times S$ be a patch transition relation. A* patch action graph $G = \langle S, \tau \rangle$ *is a finite directed graph whose vertices are the states in S and whose directed edges are given by τ.*

DEFINITION 2 *A* patch remediation posture *is a triple* $\langle S, V, \tau, < \rangle$ *where* $\langle S, \tau \rangle$ *is a patch action graph,* $V \subseteq S$ *is a set of consistent states, and* $<$ *is a preference relation over* S.

DEFINITION 3 *Let* $\langle S, V, \tau, < \rangle$ *be a posture. Then, the* patch remediation relation *for the posture is a relation* $R \subseteq S \times V$ *such that* $R(s, s')$ *holds if and only if* $s' \in V$ *(i.e., s' is a consistent state),* s' *is reachable from* s *in* $G = \langle S, \tau \rangle$, *and* $s < s'$.

Given an initial state s_0, a remediation related state s_f must be consistent, reachable from the initial state, and preferable to the initial state. However, a path from s_0 to s_f may take the system through intermediate states that are not consistent or that are not preferable to s_0. If an attack or fault causes patch remediation to abort in this intermediate state, then the system could be left in a final state that is less desirable (and possibly less secure) than the initial state. Thus, we define a special case of remediation where all intermediate states are preferable to the initial state, and where the system never moves from a consistent state to an inconsistent state.

DEFINITION 4 *Let* $\langle S, V, \tau, < \rangle$ *be a posture. Then, a* safe patch remediation relation *for the posture is a relation* $R \subseteq S \times V$ *such that* $R(s_0, s_n)$ *holds if and only if there exists a sequence of states* $s_0, \ldots, s_n$ *such that for all* $0 < i \leq n$, $\tau(s_{i-1}, s_i)$, $s_0 < s_i$, *and* $(s_{i-1} \in V) \supset (s_i \in V)$.

3. SECURITY PATCH MANAGEMENT: ALGORITHMS

We now present an algorithm for the computation of the remediation relation of a posture, for a given initial state. Let $\langle S, V, \tau, < \rangle$ be a posture, R be the induced remediation relation, and s_0 be an initial, possibly inconsistent, state. We want to compute all states s_n such that $R(s_0, s_n)$; we also want to compute the transition paths from s_0 to s_n. The states s_n are our goal states; they are consistent states that are reachable from s_0 and are preferable to s_0.

Since the transition relation can both add and delete patches and since it is sensitive to the absence and presence of patches, a simple state-space search for goal states will run into the state explosion problem in real-world systems. Hence, we consider an algorithm based on model checking. Model checking [1] is a technique for checking whether a formal model of a system satisfies a given property. A model, in our case, is a patch action graph. If a model does satisfy a specified property, then a model checker will return true; otherwise it will return a counterexample, i.e., a transition path that violates the property. Hence, we use a safety

GenerateRemediationGraph $(S, V, \tau, <, s_0)$
$p = \mathbf{AG}(\neg(s \in V \wedge s_0 < s))$
$S_{rg} = modelCheck(S, \tau, s_0, p)$
$\tau_{rg} = \tau \cap (S_{rg} \times S_{rg})$
$V_G = \{s \mid s \in S_{rg} \wedge s \in V \wedge s_0 < s\}$
return $(S_{rg}, \tau_{rg}, s_0, V_G)$.

Figure 1. Remediation Graph Generation Algorithm.

property of the form $\mathbf{AG}(\neg(s \in V \wedge s_0 < s))$. This expresses the property that it is not possible to reach a consistent state that is preferable to s_0. Model checking will return all states reachable from s_0 for which this property is false. That is, it will return all states s such that s is reachable from s_0 and a goal state is reachable from s; it will also return a transition path from s_0 to each s.

Sheyner et al [14] have described the use of model checking to construct attack graphs that represent all possible attacks on a networked system. We can adapt their algorithm to construct a remediation graph, i.e., a graph that depicts ways in which a system can be patched so that the system eventually reaches consistent states that are preferable to the initial state.

DEFINITION 5 *Let $RP = \langle S, V, \tau, < \rangle$ be a remediation posture and let $s_0 \in S$ be a state. Then, a* patch remediation graph *for the posture RP and initial state s_0 is a tuple $RG = \langle S_{rg}, \tau_{rg}, s_0, V_G \rangle$ where $S_{rg} \subseteq S$ is a set of states, $\tau_{rg} \subseteq S_{rg} \times S_{rg}$ is a state transition relation, s_0 is the initial state, and $V_G \subseteq V$ is a set of goal states.*

Goal states are defined to be all consistent states that are reachable from s_0 and are preferable to s_0. The algorithm (see Figure 1) first uses a model checker to compute the set S_{rg} of all states that are reachable from the initial state and that have a path to a goal state. It then restricts the patch action graph to the states in S_{rg}; the resulting graph is the remediation graph. As in [14], there are efficient BDD algorithms for all the operations used in this algorithm.

We can show that the patch remediation graph is exhaustive and succinct [14]:

LEMMA 6 *1 A sequence of patch actions takes a system from an initial state to a preferable, consistent state if and only if the sequence is a path in the system's remediation graph from the initial state to a goal state.*

```
ComputeMaximalGoalStates (V_G, <).
    for all s, s' ∈ V_G do
        if (s < s') then
            W = W ∪ {s}
    end
    V_MG = V_G − W
    return V_MG
```

Figure 2. Maximal Goal States Algorithm.

2 A state s (transition t) is in the remediation graph if and only if s (t) is in some path in the graph from s_0 to some goal state.

Once we have computed a remediation graph, we can use it to select a goal state and a transition path to that state. Note that a remediation graph may contain nodes that are not consistent states or that are not preferable to s_0. However, every node in a remediation graph has a directed path (possibly empty) to a goal state that is consistent, reachable from s_0, and preferable to s_0. Remediation graphs permit the efficient computation of *maximal* goal states s_n such that there are no other goal states that are preferable to the s_n. Figure 2 shows how to compute the set V_{MG} of goal states that are maximal under $<$.

The transition relation of the remediation graph provides the transition paths from the initial state to the maximal goal states. These can be used by any patch management system to deploy and install the selected patches. For instance, we have implemented similar algorithms within the Outpost System [8] that was developed at MITRE.

Note that a remediation graph can be analyzed further, e.g., to select safe transition paths that never traverse states that are less preferable than the initial state, or to determine the periods during remediation that a system may become vulnerable to attacks. Such analysis can help a security administrator reason about the implications of various defensive actions available to the administrator.

4. RELATED WORK

Several researchers have used model checkers to perform vulnerability analysis of computer networks. Ritchey and Ammann [13], Jha *et al.* [7, 6], and Sheyner *et al.* [14] use model checkers to generate attack paths (or attack graphs) by which an attacker can use known attacks to compromise a networked computer system. Our approach is based on this body of work; however, we apply those techniques to a completely different problem, namely the problem of security patch reme-

diation. Further, we propose the new, novel concept of a remediation graph which represents all possible paths by which a defender can close vulnerabilities in a system.

Ramakrishnan and Sekar [11, 12] use a model checker to discover individual vulnerabilities on single hosts. While our approach is also host-based, we do not discover individual patches but rather use a model checker to discover patch installation sequences to goal states.

Patch management systems such as Microsoft/SMS [9] and Radia Patch Manager [10] include modules that compute patch installation sequences that satisfy patch dependencies and constraints. We extend this capability with a patch preference relation that models the relative desirability of patches and applications. Further, we also build patch remediation graphs that enable us to compute maximal goal configuration states and perform analyses such as finding safe patch installation sequences.

5. CONCLUSION

The main contribution of this paper is the automatic generation of patch remediation graphs. We presented language constructs for specifying patch consistency constraints, patch action rules, and user preferences for patch configurations. We showed how a model checker can be used to construct patch remediation graphs by generating counterexamples to a safety property. The graphs permit us to compute maximal goal states that represent the preferred final states for patch remediation. The graphs also give us the sequence of patch actions (patch installs and uninstalls) that would transition the systems to the desired final states. Finally, the graphs permit us to reason about properties of the transition paths, e.g., whether the system enters some vulnerable states during the remediation process.

Future work includes combining the model with a trust management model. With this, patch policy assertions (patch dependencies, constraints, preferences, etc.) would be represented as credentials. A delegation model would provide a flexible and powerful framework for handling patch management in a distributed and heterogeneous environment. We are also examining whether remediation graphs may be used to handle other security configuration management tasks.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[2] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2), June 1998.

[3] P. Devanbu, M. Gertz, and S. Stubblebine. Security for automated, distributed configuration management. In *In Proceedings, ICSE 99 Workshop on Software Engineering over the Internet*, 1999.

[4] P. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *The Future of Software Engineering*, volume Special Volume (ICSE 2000), 2000.

[5] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Proc. of the 1997 International Conference on Distributed Configurable Systems*, pages 269–278, May 1997.

[6] Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Minimization and reliability analyses of attack graphs. Technical Report CMU-CS-02-109, School of Computer Science, Carnegie Mellon University, February 2002.

[7] Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Two formal analyses of attack graphs. In *Proceedings of the 2002 Computer Security Foundations Workshop*, pages 49–63, June 2002.

[8] Key, Proulx, Michnikov, Rubinovitz, Wittbold, and Wojcik. The Outpost system. In *MILCOM 2000 presentation*, 2000.

[9] Microsoft Corporation. *Patch Management Using Microsoft Systems Management Server 2003*, 2003.

[10] Novadigm. *Radia Patch Manager*, 2003. http://www.novadigm.com/ products/patchmanager.asp.

[11] C. R. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, September 1998.

[12] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1-2):189–209, 2002.

[13] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–165, 2000.

[14] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 254–265. IEEE Computer Society, 2002.