A Taxonomy for Comparing Attack-Graph Approaches

Todd Heberlein[1], Matt Bishop[2], Ebrima Ceesay[2], Melissa Danforth[2], C.G.Senthilkumar[2], Tye Stallard[2]

Abstract

Automated attack-graph tools show how individual vulnerabilities in a system can be composed to create meta-vulnerabilities or how individual sensor alerts can be joined to describe a multi-stage scenario attacks. In recent years many organizations have developed attack-graph type systems, but because these organizations often do not clearly define their terms, consistently use terms such as "attack-graph", explain their assumptions, or identify their limitations, comparing the various efforts is difficult at best. This paper describes our efforts to establish a taxonomic foundation for comparing and contrasting attack-graph approaches, and we use the taxonomy to compare several important efforts in the field.

1 Introduction

Attack graph systems is a growing field that focuses on how individual attacks can be composed to create larger attack paths. Such systems can answer questions such as "If an attacker starts from a machine outside my network, is there a path that will allow him to eventually read secret file CorporateSecrets?" Or "If an attacker penetrates host X in my network, how deeply into my network can he penetrate?" The attack graph answers to such questions can help prioritize the changes a network administrator makes to his network or indicate where additional intrusion detection sensors should be installed. Derivatives of the basic attack graph work take intrusion detection sensor alerts and compose them into a single composite alert. This approach reduces the total number of alerts security analysts must process and provides a context in which to understand an individual alert.

Unfortunately, while the attack graph field is growing in popularity, the lack of definitions, inconsistent use of terms, and a lack of standard feature sets have made comparing and contrasting different efforts very difficult. This paper presents our early efforts to provide a framework with which we can analyze efforts in this field. Lincoln Labs and other efforts have provided such a capability for intrusion detection systems [Hain 99], and we are trying to provide a similar capability for this emerging field.

Section 2 develops the core security definitions upon which we will build our attack graph terms. In particular, Section 2.1 provides an overview of the core terms while the rest of the section tries to develop them more formally. Because of the wide variety of representations for a "graph" in the literature, Section 3 develops our description of an attack graph that we believe can adequately represent the other efforts. Section 3 also identifies the salient features of an "exploit" as used by most of these systems. Section 4 develops several key features that can

[1] Net Squared, Inc. 530-758-4338
[2] University of California, Davis. 530-752-2149

be used to compare and contrast attack graph efforts, and Section 5 briefly illustrates the features by applying them to several attack graph efforts. Finally, Section 6 summarizes the paper.

2 Definitions

While one of the primary focuses of the various attack graph efforts is to identify how an adversary can chain together vulnerability exploitations to increase his capability, the papers rarely explicitly define what they consider a vulnerability. Furthermore, the wider literature in computer security does not appear to provide a well-defined and widely accepted definition of a vulnerability.

Publications outside the area of attack graphs do provide some definitions for vulnerabilities, but they do not fit well with our needs. Matt Bishop defines a vulnerability as "Errors in computer systems and programs ... that enable users to violate the site security policy" [Bish 99]. The problem with such a definition is that it assumes a site has a reasonably precise security policy, and too often this is not the case. The Program Analysis (PA) [Neum 78] and Research into Secure Operating Systems (RISOS) primarily identify a list of flaws that can lead to vulnerabilities. In effect, they define a vulnerability through a serious of error exemplars.

Thus, although the word "vulnerability" is used regularly throughout the computer security literature, especially on attack graphs, we have not been able to identify a definition for it that is both concise and functional with respect to evaluating attack graph efforts. This section develops a series of definitions we will use throughout this paper.

Section 2.1 provides an overview of the key terms and their relationships to one another. As developing formal definitions is important but makes for very dry reading, we have provide this summary upfront to allow the reader to skip the more formal definitions, returning to them only as needed. Section 2.2 describes the terms in more detail.

2.1 Definition Overview

In a distributed environment we define a network reference monitor as the collection of hosts' reference monitors, the expected semantics defined by privileged programs and network services, and network mediation devices such as firewalls and routers. A vulnerability is a condition that gives an adversary greater capability than would otherwise be expected given the configuration of the network reference monitor. The next several sections provide details on these concepts.

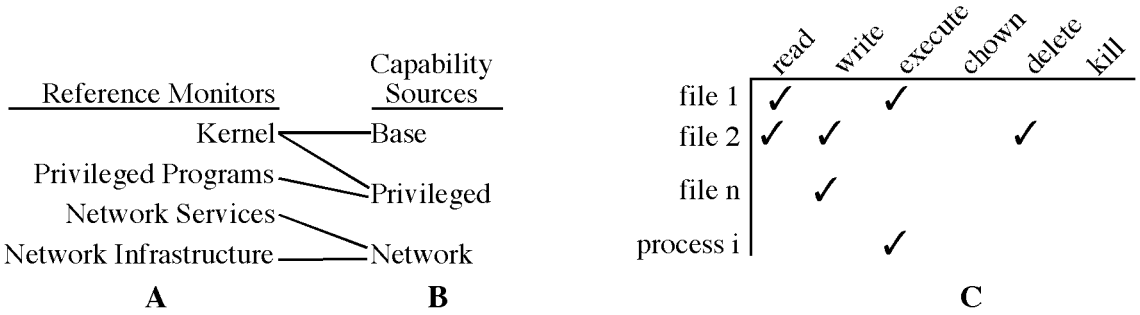

Figure 1: Reference Monitors and Capabilities

Figure 1 summarizes the core definitions of this section. Column A shows three sources of reference monitors – components that determine if what operations a process can perform on objects in the network. The first reference monitor element in the column is the tradition kernel reference, which uses a processes associated identities and objects' access control lists to allow or

block a process from performing certain operations on the objects. The second reference monitor element is the privileged programs. Privileged programs add identities to a process, thereby sidestepping the normal protections provided by the kernel's reference monitor. Usually these privileged programs do not allow arbitrary operations to be performed with their newly granted identities; instead, the privileged programs enforce their own security policies as defined by the code. The third reference monitor element is the network services. Like privileged programs, network services side step the operating system reference monitor by granting identities to subjects, in this case the subjects may be on another host. Also, like privileged programs, the network services generally do not provide arbitrary access with their granted identities. They grant limited access as defined by their code. The last element in the network reference monitor is the network infrastructure, including routers, firewalls, and software wrappers. These network elements control access across a network.

Column B in Figure 1 shows the sources of capabilities and the lines with column A indicate the reference monitor elements that control these capabilities. The first capability source, called Base, is what is provided a normal process on an operating system. The process has a specific set of identities, and the kernel's reference monitor uses the access control list for objects under its control to determine the processes set of capabilities (operations that it can perform on the objects). The second capability source, called Privilege, is the set of additional operations that can be performed because of additional identities granted by the privileged programs. These privileges are controlled by the kernel's reference monitor, which limits access to the privileged program, and the logic in the privileged program itself. The third capability source, called Network, is the set of capabilities provided to a subject by network services. The network infrastructure may limit access to the network service, and the network service provides its own limitations as defined by its code.

Column C in Figure 1 shows a capability matrix, a derivative of the access matrix, that shows (via a check mark) which operations an adversary can perform on which objects in the network. Not shown is the set of identities (e.g., users and group IDs on various hosts) controlled by the adversary. These identities, combined with the reference monitors, determine the capability matrix.

A vulnerability is then defined as a set of conditions that enable the adversary to add capabilities (e.g., add one or more checks to the capability matrix). An exploit is a set of instructions (in code, a script, or steps to be manually carried out) designed to take advantage of the vulnerability and provides specific capabilities to the adversary (checks to the matrix). An attack is the action of carrying out an exploit and adding to add a capability.

2.2 More Formal Definitions

2.2.1 Reference Monitors

In this section we discuss our model of a reference monitor. As we are looking at attack graphs across a network with attacks frequently taking advantage of privileged programs and network services, we look at a distributed reference monitor composed of traditional host reference monitors, the reference monitors enforced by logic in privileged programs and network services, and the reference monitor created by network mediating devices such as firewalls.

For the purpose of our discussions we assume nodes in the network support a reference monitor as largely defined by Karger and Schell [Karg 74]:

> Definition: **Reference Monitor** – a hardware/software combination that monitors and mediates all references by programs to all resources controlled by the operating system.

The definition presumes that processes must be associated with one or more identities (e.g., users and groups), and resources (files, devices, processes) must support a form of access control list (ACL) that define which identities can perform what operations on them (e.g., permission lists). In short, the ACLs define the normal capabilities (the ability to manipulate resources controlled by the operating system) available to a process running with a given set of identities, and the reference monitor enforces that definition.

Some kernel reference monitors support richer semantics; for example, access may also be controlled by the application that gave rise to the process. More concretely, a semantically richer reference monitor could limit processes generated by executing the *passwd* program, which runs with the root identity, to only reading and modifying the appropriate password files. However, our focus here is on basic components widely available in today's networks.

Modern operating systems support the concept of privileged programs. A privileged program grants a subject with an existing process on the system additional capabilities. The privileged program runs with (presumably) identities not currently available to the subject, so the subject can in theory perform more operations than the kernel reference monitor would otherwise allow. In general the privileged programs do not allow the adversary to perform arbitrary actions with their newly granted identities but instead enforce their own access controls via policies defined by the program's code. Unfortunately, few privileged programs and network services concisely and explicitly define their security policies.

> Definition: **Privileged Programs Reference Monitor** – the collection of privileged programs that provide additional identities to processes and the restrictions those programs provide by their own internal logic.

Similarly, most modern operating systems support network services – programs that interact with entities across a network. These network services run with their own identities on a host, so they can potentially provide any remote entity with all privileges on the host available to the program's identities; however, as with privileged programs, the network services typically restrict what a remote entity can do through its own security policy defined by the program. And as with privileged programs, networks services rarely concisely and explicitly define their security policy

> Definition: **Network Services Reference Monitor** – the collection of network services that provide additional identities to processes and the restrictions those programs provide by their own internal logic.

The security policies for the privileged programs network services are defined by their code, but since the code for these programs are often not widely available and rarely reviewed and understood by personnel at most sites, we claim the security policy for there programs are defined by their "expected behaviors."

Finally, as we are operating in a networked environment, we define a third component of the distributed reference monitor, the Network Reference Monitor. While network services provide identities to remote entities, components such as firewalls may prevent some remote entities from taking advantage of the network services.

> Definition: **Network Reference Monitor** – the collection of hardware (i.e., routers, switches, and firewalls) and software that monitor and mediate communications between hosts. The security policy for the network reference monitor is distributed across ACLs defined for each mediating component.

To conclude this section, in a networked environment the reference monitor responsibility is distributed between (1) the hosts' operating system reference monitors, (2) the

privileged programs on the hosts, (3) network services on the hosts, and (4) the network devices and software that mediate communication between hosts.

2.2.2 Adversary Capabilities

This section provides our basic view of capabilities, and it is very closely tied to the various reference monitor components listed in the previous section. We begin with the two simple definitions of an adversary and capability, and then we refine the capability concept in terms of the reference monitor pieces.

Definition: **Adversary** – a subject that may consist of any number of humans and programs working together to perform actions in the environment against the wishes of the owner(s) of the environment.

For our purposes we want to measure a set of systems and resources with respect to some adversary's ability to interact with those resources. The adversary may be a single interactive human intruder. The adversary may be a group of humans working in a coordinated fashion. The adversary may be an automated and independent program that can divide and propagate throughout the organization. The adversary may begin "inside" the organization, "outside" the organization, or both. We may want to measure how many resources an adversary can acquire, or we may want to determine if an adversary can access a specific resource.

Definition: **Capability** – the ability of a process with a given set of identities to perform a given operation on an object.

A more formal definition of "capability" can be found in [Bish 03], but our simple triple (identities, operation, object) captures what we need. For example, "a process (e.g., an editor) owned by Bob (the identity) can read (the operation) the file foobar (the object)" defines a capability.

On each system in the network the adversary has set of capabilities defined by the set of identities controlled by the adversary and the various reference monitors and their security policies (e.g., ACLs) that they enforce. Following the three elements of the reference monitor described in the previous section we refine the set of adversary capabilities into three groups: the Base Adversary Capability, the Privileged Adversary Capability, and the Network Adversary Capability.

> Definition: **Base Adversary Capability(*I*)** – Let *I* be the set of identities controlled by the adversary. Then the Base Adversary Capability(*I*) is the set of all operation-object pairs allowed by the Reference Monitor for the identities in *I*.

Essentially, the Base Adversary Capability identifies what the adversary can normally do on a system without relying on any special programs or services.

Most modern operating systems support the concept of a privileged program that grants the adversary an additional set of identities. For example, UNIX-class operating systems support the Set User ID (SUID) and Set Group ID (SGID) programs that can add additional user and group identities to a process. My Mac OS 10.3.3 system currently has 81 such programs installed. As mentioned in the previous section, these programs, in effect, bypass the security provided by the Reference Monitor and provide the adversary with the theoretical ability to perform any actions allowed by granted privileges. The privileges and restrictions on these privileges (as encoded by the program logic) are controlled by what we described as the Privileged Program Reference Monitor. The capabilities provided to the adversary on a host are defined as follows:

Definition: **Privileged Adversary Capability(*I*)** – Let *I* be the set of identities controlled by the adversary. Then the set Privileged Adversary Capability(*I*) is the set of all operation-object pairs allowed by the combination of the Reference Monitor (which controls access to the privileged programs) and the Privileged Program Reference Monitor for the set of identities *I*.

In addition to privileged programs, most modern operating systems provide the capability to run a network service – a program that interacts with entities across a network (e.g., a web server). The network service runs with a given set of identities on its host system. Whereas the operating system Reference Monitor controls access to privileged programs, the Network Security Monitor controls access to the network services, so they help define the capabilities provided to the adversary.

Definition: **Network Adversary Capability(*IH*)** – Let *IH* be the set of identity-host combinations controlled by the adversary. Then the Network Adversary Capability(*IH*) on a single host is the set of all operations-object pairs allowed on the host by the combination of Network Reference Monitor and the Network Service Reference Monitor of the set of identities *IH*.

The Network Adversary Capability does not include identity parameter, ***I***. This is because the operating system's reference monitor does not, in general, mediate access to the network services. The reference monitor capability is essentially distributed to the network infrastructure (e.g., firewalls, routers, and other network wrappers) that determines if an adversary can reach the network service and by the network service itself, which is responsible for providing any identification and authentication capability and imposing any restrictions based on the provided identification.

The adversary's total capability is then based on the union of each of these capabilities across all hosts. On many hosts, the initial identity set, ***I***, will be empty, but depending on the availability of a network services and the Network Reference Monitor, the Network Adversary Capability may be non-empty. Thus we add the following definition:

Definition: **Total Adversary Capability(*IH*)** – Let *IH* be the set of identity-host combinations controlled by the adversary. Then the Total Adversary Capability(*IH*) is the union across all nodes in the network of the Base Adversary Capability(*I*), the Privileged Adversary Capability(*I*), and Network Adversary Capability(*IH*), where *I* is identities available to the adversary on each host.

To summarize this section, an adversary is the actor, which can be any combination of users and programs, attempting perform actions against at odds to the wishes of the owner of the environment. The Total Adversary Capability defines all the operations on all the objects in the network that the adversary can perform as defined by the distributed reference monitor.

2.2.3 Expected Semantics

In the previous section we used the phrase "expected semantics" for privileged programs and network services, and this phrase is one source of difficulty when trying to reach a consensus on what a vulnerability is. For example, suppose a privileged or network program has a little used and little known feature that provides additional capabilities. For example, the "debug" command in many Sendmail servers circa 1988 was added to help debug the server's behavior and was used by the Morris Worm. More recently, applications on Windows systems can read a users address book and send email, a feature widely exploited in email-based worms. Are these examples of vulnerabilities or features?

Unfortunately, few privileged programs and network services provide explicit, complete, and concise descriptions of what they should do, and this complicates the job of securing a site. For example, in the mid 1990s UC Davis developed specification-based detection, a technique that detected attacks by detecting unspecified behavior of privileged programs and network services [Ko 97], and later Network Associates and other organizations applied similar techniques to operating system wrappers, essentially extensions to the reference monitor [Ko 01], but in each case researchers needed to develop their own specifications for the relevant programs. No existing database of specifications exists for these programs. Furthermore many individuals and sites develop their own privileged programs for various reasons. Also, new programs seem to appear daily that include a network server component, and often these services are not adequately described in the documentation.

Thus, without formal and widely agreed upon specifications for the behavior of privileged programs and network services we must settle on a much less formal term: "expected semantics".

2.2.4 Vulnerability

Previously we defined the TotalAdversaryCapability as the set of all activity on all objects an adversary could perform as defined by the reference monitors, network reference monitor, and the expected behaviors of the privileged programs and network services. Let us define a partial ordering for this set such that TotalAdversaryCapability1 <= TotalAdversaryCapability2 if and only if an adversary with TotalAdversaryCapability2 can perform all actions on all resources that an adversary with TotalAdversaryCapability1 can.

> Definition 1: **Vulnerability** – a set of conditions that allow an adversary with the capability set $TotalAdversaryCapability_i$ to achieve a capability set $TotalAdversaryCapability_{i+1}$, such that $TotalAdversaryCapability_i < TotalAdversaryCapability_{i+1}$.

In other words, a vulnerability is a set of conditions (i.e., programming flaw (e.g., buffer overflow in a program) or configuration flaw (e.g., permissions that allow the password file to be replaced)) that allows the adversary to acquire additional capabilities.

3 Normalizing Attack Graph Efforts

With core definitions security definition in place, this section looks at common definitions specific to the attack graph field. In particular, in Section 3.1 we briefly look at the different ways "graphs" are used in attack graph efforts, and in Section 3.2 we identify how an exploit is described in the various efforts.

3.1 Attack Graphs

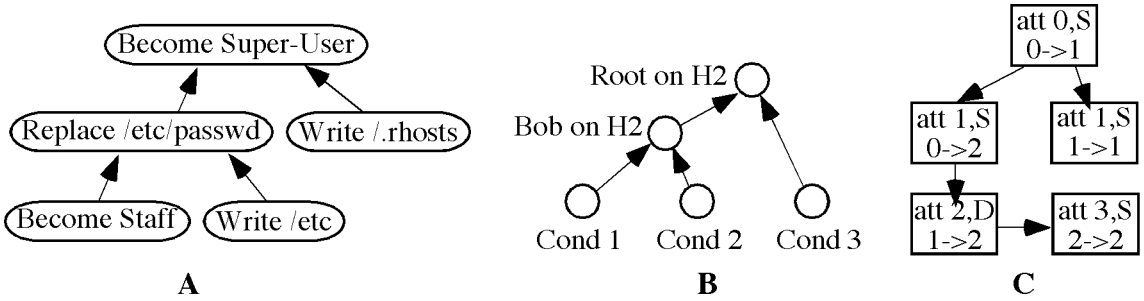

Figure 2: Various Attack Graphs

One particular problem when comparing various attack graph efforts is that the different publications present their "attack graphs" differently. Figure 2 shows three different representations used in the literature. Graph A is from the Kuang work as shown in [Bald 90], and each node represents a goal with arrows shows the goal/subgoal relationships. Graph B is based on the GMU work described in [Amma 02], and each node represents a state in the network with arrows showing how an exploit takes advantage of some conditions to enable subsequent conditions. Graph C is based on the CMU work [Shey 02], and each node represents a specific attack with the arrows showing the order of attacks.

Fortunately, each of these graphs can be mapped into what we call a canonical graph representation (see Figure 3). In this model, each node represents the state of the entire system at a given point in time. Furthermore, we divide the state into two subgroups. The first subgroup, visually depicted in the upper half of the node, represents the state of the network itself, including the presence of programming flaws in programs or whether a particular user has write access to a file. The second subgroup, in the lower half of the node, represents the identities (e.g., users and groups) across the network acquired by the adversary. As mentioned previously, these two components of the system state (network state and adversary identities) can be combined to create the capability matrix shown in Figure 1, so we could also create a graph in which the nodes represent the adversary capability matrix at any moment in time. However, we believe the network state and adversary identities nodes are more intuitive.

The arcs in the graph represent an attack that changes the state of the network, and we label the attack arc with the exploit used. An attack changes the state by adding an identity to the adversary's set (exploit 1 to node in middle), changing the network state (exploit 2 to node on right), or a combination.

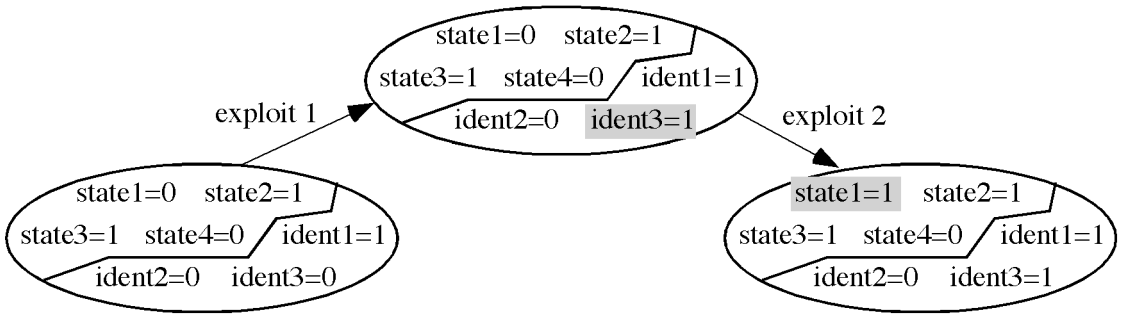

Figure 3: Canonical Attack Graph

3.2 Exploit Specification

While the various attack graph efforts we reviewed varied widely regarding what the nodes and arcs were in the graphs they displayed, the various efforts were much closer in their representations of exploits. In or canonical graph model in Figure 3, the exploits are the means by which the system transitions from one state to another. Figure 4 shows examples for specifying the same exploit in three different systems – an attack against a buffer overflow vulnerability in an ssh daemon. The specification in column A is from the CMU effort [Shey 02]; column B shows a JESS expert system rule from a class project at UC Davis; and column C shows the exploit in the Requires/Provides system developed at UC Davis [Temp 00].

Every attack graph effort we reviewed supported the same basic structure: a set of preconditions that needed to be satisfied before the exploit could be carried out and a set of post-conditions that described the change of state in the system. The authors of the various projects typically used different names (e.g., preconditions and affects (CMU effort), predicates and

actions (JESS version), requires and provides (UCD [Temp 00] effort), and goals and subgoals (Kuang [Bald 90])), but all the efforts used the same basic approach. In the examples in Figure 4 the precondition part of the exploit rule is highlighted in gray.

A

```
attack sshd-buffer-overflow is
    intruder preconditions
        plvl_A(S) ≥ user
        plvl_A(T) < root
    network preconditions
        ssh_T
        R(S,T,sp)
    intruder effects
        plvl_A(T) := root
    network effects
        ¬ssh_T
end
```

B

```
(defrule sshd-buffer-overflow
    (priv ?s ?v)
    (test (>= ?v 1))
    ?plvl <- (priv ?t ?p)
    (test (< ?p 2))
    ?sshd <- (var ?t ssh)
    (connect ?s ?t ssh)
    (not (ids ?s ?t ssh))
  =>
    (retract ?plvl)
    (retract ?sshd)
    (assert (priv ?t 2))
)
```

C

```
concept sshd_buffer_overflow is
requires
    Privilege_Level : PLS;
    Privilege_Level : PLT;
    Service_Active  : SA;
    Reachable       : R;
with
    PLS.level >= USER;
    PLT.level < ROOT;
    SA.host is PLT.ip_addr;
    SA.service is SSH;
    R.src is PLS.ip_addr;
    R.dst is PLT.ip_addr;
    R.port is SA.port;
end;
provides
    remote_execution : REX;
with
    REX.from  <- PLS.ip_addr;
    REX.to    <- PLT.ip_addr;
    REX.using <- SSH;
end;
```

Figure 4: Examples of Specifying an Exploit

4 Attack Graph Features

In this section we examine several key features that can be used to compare and contrast attack graph systems.

4.1 Monotonic vs. Non-monotonic Exploits

A monotonic function is a function f() for which f(x) >= f(x′) for x >= x′. A strictly monotonic function is one for which f(x) > f(x′) when x ≠ x′. We can define exploits similarly. We define the capability matrix after stage s as C(s), and the capability matrix at stage s+1 after an exploit as C(s+1). If after an exploit is launched at stage s and C(s+1) ⊃ C(s), then the exploit is strictly monotonic. Essentially a monotonic exploit is one that adds capabilities and for which no capabilities are lost.

From a practical point of view, what does this mean? Imagine an exploit E0 that has preconditions P1 and P2 will allow an adversary to reach his goal. Furthermore, suppose exploit E1 provides the precondition P1, and exploit E2 provides the precondition P2. If E1 and E2 are both monotonic attacks, the adversary can launch E1 followed by E2 or launch E2 followed by E1. The order does not matter. However, suppose exploit E1 is non-monotonic (e.g., it takes away some capability) such that it invalidates the preconditions needed for E2. Then if the adversary launches exploit E1 first, then he cannot launch exploit E2 because the preconditions for it no longer hold. Thus, the adversary cannot not satisfy precondition P2, so he cannot carry out exploit E0 to achieve his goal. On the other hand, if the adversary launches exploit E2, he can then launch E1 followed by E0 and achieve his goal. Thus, from the adversary point of view, for an approach that supports non-monotonic exploits, the order in which attacks take place matters.

From our modeling point of view, a system that supports non-monotonic exploits requires many more possible states to be explored.

Most attack graph efforts only work with monotonic exploits; although, most efforts do not state this or may even be unaware of it. For example, the exploit rule in column B of Figure 4 shows what appears to be a non-monotonic rule: the line "(retract ?ssh)" removes a potential capability. However, such a rule could potentially result in a false negative (the system reporting that there is no attack path when in fact there is) depending on the order of rule execution. Of the systems we reviewed, only the ones based on symbolic model checkers could effectively handle non-monotonic exploits.

While monotonic vs. non-monotonic exploits pose interesting challenges, Paul Ammann, who has developed systems that support both approaches ([Ritc 00] for non-monotonic exploits and [Amma 02] for monotonic exploits), claims in [Amma 02] that it is only an academic interest since in their experience with building a database of exploits, all exploits can be reasonably modeled as monotonic attacks.

4.2 Single Path vs. All Path

An attack graph system can produce a single attack path showing one example of how an adversary can achieve his goal, or the system can produce all possible paths. If a system generates all attack paths, additional graph-based analysis can be performed such as identifying which vulnerabilities contribute to most attack paths or identifying the minimal set of vulnerabilities that need to be removed to prevent the adversary from reaching a particular goal.

Most systems we reviewed generated a single attack path. In many cases this was a side effect of the system they were using. For example, many symbolic model checkers only generate a single counter example (the attack path). Likewise, expert systems typically stop once a goal has been achieved.

4.3 Forward vs. Backward Chaining

Some attach graph system use forward chaining techniques, while other exploit backward chaining techniques. Backward chaining systems such as the original Kuang [Bald 90] and many since then start with a goal (e.g, acquire the root identity on a particular system). Then the system finds exploits for which the post-condition portion matches the goal. If the preconditions of an exploit are already satisfied, then an attack path is found from the initial condition to the goal; otherwise, the preconditions of the exploit are added to a set of goals to be satisfied.

Forward chaining systems such as the GMU work described in [Amma 02] start with the existing conditions and finds exploits for which the precondition portions match the existing conditions. Exploits found to match these conditions are executed, the new conditions (i.e., capabilities) are added to the system, and the process continues until no more exploit preconditions can be matched to the conditions or some goal is achieved.

One advantage forward chaining has over backward chaining is that the user does not need to identify a specific goal for the adversary to achieve. The user could simply ask, "given the following network conditions, an adversary controlling the following initial identities, and the adversary knowing the following exploits, how far into my network can he penetrate?"

While not entirely accurate, backward chaining systems can be viewed as a depth-first search, while forward chaining systems can be viewed as a breadth-first search.

We have read some claims that backward chaining is more efficient because states that are not necessary to achieving the goal are never explored. However, from our experience with

backward chaining systems, a tremendous amount of backtracking often occurs when trying to find a path. We have not read any detailed arguments or seen any experimental evidence indicating under which conditions one approach is more efficient than the other.

4.4 Building on Existing Framework or Hand-rolled Code

Many attack graph systems build on existing frameworks, while other efforts are coded from scratch. For example, the CMU work [Shey 02] used the NuSMV symbolic model checker, the initial Booz Allen & Hamilton and GMU work [Ritch 00] used the SMV symbolic model checker, and the student-based effort referred to earlier (Section 3.2) used the JESS expert system. Other efforts such as GMU's later work [Amma 02], the Sandia National Laboratories' work [Swil 01], and the original Kuang effort [Bald 90] essentially rolled their own code.

One side effect of building on existing frameworks is that the authors can often borrow claims from the framework they are using. Furthermore, by building on existing frameworks, the authors can typically point readers to additional literature that describes the underlying execution engines.

4.5 Formal Claims

Some papers simply describe their system's ability to generate graphs while others make explicit formal claims about their efforts. For example, [Shey 02] state that their approach is exhaustive (all possible attack paths are identified), and the number of states and transitions identified are minimal (e.g., all states and transitions are part of at least one attack path). Likewise, [Amma 02] explicitly describes their algorithm and identifies upper bounds on performance based on the number states in the network and exploits known to the system.

4.6 Performance

Few papers made any explicit performance claims or backed them up with actual results, and most efforts only illustrated their system with toy examples (e.g, three hosts and handful of exploits). The Kuang documentation [Bald 90] states that they ran their tool on a host with 300 users and 25 groups (in the early 90s the authors ran the tool on similarly large systems), but no specific times are reported. NetKuang [Zerk 96] was run on a real network of ten workstations with an average performance of finding the first attack path of 6.2 seconds.

The CMU work [Shey 02] described the performance of a simple model (3 hosts, 4 attacks) that took 5 seconds. When the model was expanded to 5 hosts and 8 attacks, the system took 2 hours. While these numbers certainly raise concern, this system produces *all* attack paths (not just the first one it finds) and it supports non-monotonic exploits. Furthermore, in personal communications Sheyner has stated that there have been a number of performance enhancements since this publication.

While our discussions have primarily focused on performance with respect to time, memory requirements can also be a limiting factor. On this issue, most publications are silent; however, as the efforts scale to production networks with large amounts of state that must be supported, this will certainly become an issue.

Feature Summary

In this section we identified a number of important features that can be used when comparing and contrasting attack graph efforts. These include:

- Supporting monotonic vs. non-monotonic exploits. This attribute describes the range of exploits the system can support.

- Single path vs. all paths. This attribute indicates whether the system generates a single attack path or all possible attack paths.
- Forward vs. Backward chaining. This attribute describes how the basic execution engine works (by matching post-conditions and trying to satisfy preconditions (backward chaining), or by matching preconditions and modifying the state with the post-conditions (forward chaining).
- Building on existing frameworks or using original code. This attribute tells whether the system makes use of existing execution engines.
- Formal claims. This attributes describes the actual claims made by the authors about the system.
- Performance. This attribute describes the upper and expected time and space performance of the approach. Typical factors affecting the results include the amount of state in the network, the number of exploits known, and the number of nodes and arcs in the final attack graph.

5 Example Analysis

With several important features used for comparing and contrasting attack graph efforts, we now briefly map various attack graph efforts onto this feature set.

5.1 Kuang System

The SU-Kuang system (Simple UNIX Kuang), developed in the late 1980s and early 1990s by Robert Baldwin [Bald 90] is, to our knowledge, the first attack graph system. Baldwin considered all attack graph type systems to be Kuang systems, and his implementation was simply one instantiation.

Attributes	Value
Monotonic vs. non-monotonic exploits:	Monotonic
Single path vs. all paths	Single path
Forward vs. backward chaining	Backward chaining
Building on existing framework	No
Formal claims:	None
Performance:	No numbers provided, but system has been run on relatively large production systems.

5.2 Carnegie Mellon University Work

The Carnegie Mellon University group originally teamed with Lincoln Labs to develop an attack graph system; although, both groups have gone their separate ways since. Below is the summarization of their system as described in [Shey 02].

Attributes	Value
Monotonic vs. non-monotonic exploits:	Non-monotonic
Single path vs. all paths	All paths
Forward vs. Backward chaining	Backward chaining
Building on existing framework	Modified NuSMV
Formal claims:	Several
Performance:	Potentially exponential, actual numbers provided.

5.3 George Mason Work

George Mason University originally teamed with Booz Allen & Hamilton to develop a system based on the SMV symbolic model checker [Ritch 00], but later GMU, concerned about the scalability issues abandoned the model checker in favor of a hand-coded implementation [Amma 02].

Attributes	Value
Monotonic vs. non-monotonic exploits:	Monotonic
Single path vs. all paths	All paths
Forward vs. Backward chaining	Forward chaining
Building on existing framework	No
Formal claims:	Several
Performance:	Bounded by the size of the resulting attack graph, not the exponential number of potential states.

6 Conclusions

Attack graphs, a field in which individual attacks are composed to create larger attack paths, have grown in popularity in recent years, but the lack of definitions, inconsistent use of terms, and a lack of standard feature sets have made comparing and contrasting different efforts very difficult. In this paper presents our current work to develop a standard framework for comparing and contrasting attack graph.

Our first efforts were to define the term "vulnerability". The term is used freely in all the publications and the definition of which would appear obvious, but as we discovered the term is difficult to define. Ultimately we defined a number of terms to support our definition of vulnerability, and finally had to fall back on "reasonable expectation" with regards to how we think the network should behave.

Next, we normalize the efforts by coming up with a standard representation of an attack graph. We settled on a node in the graph representing the entire system state, including variables in the network such as whether a particular programming flaw exists in a program and the set of identities acquired by the adversary, and arcs in the graph representing the adversary using an exploit to transform the system state. We claimed, but did not provide proof, that the other forms of attack graphs could all be mapped to our form.

We also looked at the representation of exploits, but unlike the attack graphs themselves, the various efforts proved to be relatively consistent in their design if not necessarily in their terminology. Virtually all efforts defined a set of preconditions that needed to be met in order to carry out an exploit and a set of post-conditions that defined the changes in the system following the exploit.

With definitions and normalization efforts in place, we then developed an initial set of features by which we could compare attack graph efforts. These features are (1) monotonic vs. non-monotonic exploits, (2) generating single attack path vs. all attack paths, (3) forward vs. backward chaining execution engines, (4) building on existing frameworks vs. original code, (5) formal claims that can be made about the system and results, and (6) the performance of the system in terms of performance and memory. To illustrate the use of these features, we briefly applied them to three attack graph systems.

7 References

[Amma 02] Paul Ammann, Duminda Wijesekara, and Saket Kaushik. “Scalable, graph-based network vulnerability analysis”, *Proceedings CCS02: 9th ACM Conference on Computer and Communication Security*, pages 217 – 224, Washington, DC, November 2002. ACM.

[Bald 90] Robert W. Baldwin. “Kuang: Rule based security checking.”. Documentation in ftp://ftp.cert.org/pub/tools/cops/1.04/cops.104.tar. MIT, Lab for Computer Science Programming Systems Research Group.

[Bish 99] M. Bishop, "Vulnerabilities Analysis," *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection*, pp. 125-136 (Sep 1999).

[Bish 03] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, Boston, MA, 2003.

[Hain 99] Joshua Haines, Lee Rossey, Rich Lippmann and Robert Cunnigham, "Extending the 1999 Evaluation", *Proceedings of DISCEX 2001*, June 11-12, Anaheim, CA.

[Karg 74] Karger, P. A., and Schell, R. R., *Multics Security Evaluation: Vulnerability Analysis*, ESD-TR-74-193 Vol. II, ESD/AFSC, Hanscom AFB, Bedford, MA (June 1974).

[Ko 97] C. Ko, M. Ruschitzka, and K. Levitt, “Execution Monitoring of Security-critical Programs in Distributed Systems: A Specification-based Approach”, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 134-1444, (1997).

[Ko 01] C. Ko, P. Brutch, J. Rowe, G. Tsafnat, and K. N. Levitt. "System Health and Intrusion Monitoring Using a Hierarchy of Constraints," *Recent Advances in Intrusion Detection (RAID) 2001, Lecture Notes in Computer Science*, W. Lee, L. Me, and A. Wespi, eds., Vol. 2212, pp. 190-203, (2001).

[Neum 78] P.G. Neumann, “Computer System Security Evaluation,” *1978 National Computer Security Conference Proceedings* (AFIPS Conference Proceedings 47), pp. 1087-1095, (June 1978).

[Ritc 00] Ronald W. Ritchey and Paul Ammann. “Using model checking to analyze network vulnerabilities”, *Proceedings, 2000 IEEE Symposium on Security and Privacy*, pages 156 – 165, Oakland, CA, May 2000.

[Shey 02] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeanette Wing. “Automated generation and analysis of attack graphs”, *Proceedings IEEE Symposium on Security and Privacy*, pages 254 – 265, May 2002.

[Stew 64] Justice Potter Stewart, Supreme court opinion, Jacobellis v. Ohio, 378 U.S. 184, 197 (1964).

[Swil 01] L. Swiler, C. Phillips, D. Ellis, and S. Chakerian. “Computer-attack graph generation tool”, *Proceedings DISCEX '01: DARPA Information Survivability Conference & Exposition II*, pp 307-321, June 2001

[Temp 00] Steven J.Templeton and Karl Levitt. “A require/provides model for computer attacks”, *Proceedings New Security Paradigms Workshop*, Cork Island, September 2000.

[Zerk 96] 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.