A Requires/Provides Model for Computer Attacks

Steven J. Templeton
Dept. of Computer Science
University of California, Davis
templets@cs.ucdavis.edu

Karl Levitt
Dept. of Computer Science
University of California, Davis
levitt@cs.ucdavis.edu

Abstract* – *Computer attacks are typically described in terms of a single exploited vulnerability or as a signature composed of a specific sequence of events. These approaches lack the ability to characterize complex scenarios or to generalize to unknown attacks. Rather than think of attacks as a series of events, we view attacks as a set of capabilities that provide support for abstract attack concepts that in turn provide new capabilities to support other concepts. This paper describes a flexible extensible model for computer attacks, a language for specifying the model, and how it can be used in security applications such as vulnerability analysis, intrusion detection and attack generation

Introduction

Traditionally computer attacks are described in terms of the single vulnerability exploited in the attack: a buffer overflow in sendmail, a race condition in rdist, a denial-of-service by sending pings to a broadcast IP address, etc. While such *single-point attacks* are a frequent occurrence, in isolation they are of little significance. Today's serious attacks are complex, multi-stage scenarios that coordinate the effects of various single-point attacks to reach goals not otherwise attainable. Such attacks can involve bypassing multiple security mechanisms and the use of numerous computer systems. The complexity and sophistication of these attacks indicate highly motivated adversaries and suggest the high value of the attackers' goals. Consequently, methods to understand, predict and identify these *scenario attacks* are important challenges for computer security research.

Typically, scenario attacks are described using the specific sequence of actions the attack uses to reach some specific goal. Such descriptions are useful for communicating the details of a specific attack or building specialized signatures for use in attack detection, but lack the ability to generalize beyond the stated scenario or to be utilized as a sub-goal in more complex attacks. As an alternative to describing attacks using explicit signatures we describe a model of attacks based on the requirements of the abstract components of the attack, the capabilities the components provide to satisfy the requirements of other components, and a method for composing the components into complete attacks. Because this model does not require a priori knowledge of a particular scenario, we can implicitly describe numerous unknown attacks.

Central to our model is our attack specification language, JIGSAW. JIGSAW provides a convenient tool for describing attack components in terms of *capabilities* and *concepts*. The language also allows detailed specification of the system being modeled and has extensions to support construction of security applications such as intrusion detection systems.

In this paper we will describe with examples how attacks can be modeled using our method. We include illustrative examples in JIGSAW and discuss how this model can be used to create intrusion detection, attack analysis, and attack generation systems. We also discuss how the model can be extended to include policy requirements, incorporate both signature and anomaly based intrusion detection systems, create retargetable sensors, and automated response.

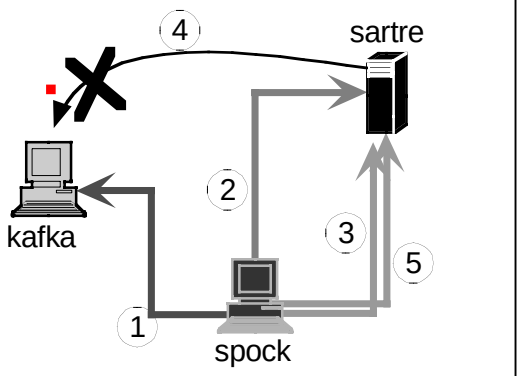
Scenario Attacks – an example

Two computers, *kafka* and *sartre*, have a trust relation in that *kafka* may execute commands on *sartre* using rsh[1]. To exploit a vulnerability in

[1] See [7] for a full description of the RSH protocol including analysis and exploits.

rsh, an attacker on host *spock* causes a *synflood* denial-of-service attack against *kafka*. [see figure 1].

Figure #1: RSH Connection Spoofing



1: synflood initiated on kafka
2: spock probes sartre for starting sequence number info
3: spock sends spoofed packet to sarte
4: sartre responds to kafka, but kafka is not responding
5: spock sends spoofed packet sartre expected to get from kafka. data includes body of attack.

SynFlood is a single point flooding attack that exploits a vulnerability of the TCP protocol. This attack prevents a target host from accepting new connections to the particular port. By sending large numbers of TCP syn packets to a port on the target host, but never responding to the syn/ack the target returns, the attacker can prevent the target from opening new connections to that port. This works because the target host holds open a connection for each syn packet it receives while waiting for the initiator to respond. Because the queue that handles the connections is of limited size, by creating a large number of these half-open connections, the attacker exhausts the queue on the target host. Eventually, the waiting connections on the target host will time-out and they will be reset, which opens the queue for new connections. However, if the attacker is able to send packets to the target faster than the connections are reset, the target host will be prevented from responding to new connections.

Assuming the target port is TCP/11111 on *kafka*, while the synflood is active, packets for new connections to this port will be discarded. This effectively prevents *kafka* from responding to packets sent to this port. Meanwhile *spock* then sends syn packets to the rsh port on *sartre* and monitors the time and what starting sequence number *sartre* specifies *spock* should use to send the ack packet. By using this information, the attacker can now guess what sequence number a connection from *kafka* to *sartre* would use. Knowing this, *spock* sends a syn packet to the rsh port on *sartre*, with the source address and port in the packet forged to be *kafka*, TCP/11111. *Sartre* sends the syn/ack packet to *kafka*, but due to the denial-of-service, *kafka* is not responding (if it responded it would realize it did not start the connection and reset). *Spock* now sends an ack packet, again claiming to be from *kafka*, and using the sequence number he guessed *sartre* would have sent to *kafka*. Assuming the sequence number matches, the packet is accepted as if it had originated from *kafka*. The data portion of this packet can contain commands to be executed via rsh. Therefore, this attack provides the capability to execute arbitrary code on the target computer.

This example illustrated the key elements of a scenario attack. It uses a combination of single point attacks (plus other activity not necessarily by itself an attack), to exploit a vulnerability that could not be directly exploited. Both the synflood attack and the sequence number probing are required for the forged packets to be accepted. The single-point attack is the synflood; the higher-order vulnerability is in the rsh trust relation.

This specific scenario was easy to describe, however many variants can be created. The large number of variant scenarios is the primary challenge in describing these attacks. We identify five types of variants:

- Mutation
- Resequencing
- Substitution
- Distribution
- Looping

These are described below.

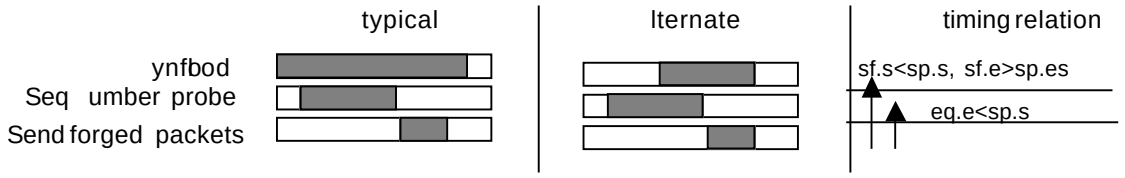
Mutation is the simplest method to create variants. This involves altering the attribute values of the attack events. Changing a host IP address, port number, or username are examples. These

are trivial variants, easily captured by using variables in the attack description.

Resequencing refers to modifications in the order of events. The synflood in our example needs to be in effect to prevent *kafka* from responding. As a variant, rather than start the synflood first, the synflood could occur after the sequence number probe. [see figure 2]. As long as each event is positioned so that the capabilities it provides are available when required, the attack can hold[2].

Substitution replaces one attack in the scenario with another equivalent attack. In our example, replacing the synflood with any attack that would prevent *kafka* from responding to packets sent by *sartre*, and not negate other requirements would work. These include single point attacks such as a *packet-storm*[3], *cable grounding*[4], or a *ping-of-death*[5]. Such *event equivalency* is a powerful method for creating new scenarios. Any event that provides the capabilities of another and remains non-interfering[6] can replace the original event in the scenario. This includes single-point and scenario attacks. For example, the buffer overflow vulnerability in Linux 5.1 imapd could be exploited to gain root access to *kafka* and reboot it. Alternately, by exploiting a buffer overflow in a cgi script on the web-server for the group, the attacker could execute code as httpd. Using this he could identify usernames and attempt to crack passwords. With a username/password pair the attacker could log on to *kafka* and as the user and cause a denial of service to the desired port. Both attacks are event equivalent.

Figure 2: variations in connection spoofing timing.



The typical timing relation for connection spoofing is to start the synflood and continue it throughout the attack. An alternate starts the synflood late, but it is active when the forged packets are sent. The timing relation shows that the forged packets (sp) must be sent during the synflood (sf), the forged packets can not be sent before the sequence number probe (seq) has terminated.

Rather than vary the denial-of-service attack, anything that is event equivalent to the sequence number guessing, or forged packet send could replace these events in the example. If the overall goal of the attack was to allow execution of arbitrary code on *sartre*, then any combination of events that leads to that goal will be functionally equivalent. For example, rather than exploit the rsh trust relation, it may be possible to exploit a NIS password trust relation[6]. Clearly, even for this simple scenario, a multitude of permutations exist.

[2] Success of the attack is a different problem. This is discussed in section.

[3] packet storm: typically an extremely high volume of packets being sent between two computers resulting in complete saturation of the network. This is commonly caused by tricking two or more computers to go into a feedback like mode responding to the response of the other.

[4] cable grounding: physical damage to the network wire, e.g. sticking a straight pin through coax to short inner and outer conductors.

[5] ping-of-death: a generic term for a ping w/ packet data of size or content that causes the recipient to halt or go into a long reset cycle. Can also refer to packets which cause the recipients kernel to halt.

[6] This refers to the situation, where the effects of one concept (on certain capability instances) conflicts with those of another concept. E.g. one denial-of-service collaterally causes denial-of-service on a host required to be available for the attack.

Distribution creates variants by use multiple values for the same attribute throughout the attack. For example, rather than use *spock* to synflood *kafka* alone, the attacker could use *spock*, *kirk*, and *sulu*. Or, rather than use real IP addresses, since the synflood requires no response randomized, fake IP addresses could be used. Distribution can cause an explosion in the number of events that make up the scenario.

Looping variants repeat some portion of the scenario multiple times. This can be done intentionally to create variants to defeat detection, however looping is a natural occurrence in many attacks. For example, the number of packets a synflood attack requires to initially exhaust the targets queue, and the frequency of additional packets required to keep it full will depend on the systems involved. Similarly, the number attempts to guess the correct sequence number may vary. The number is to some extent statically a function of the system configuration, but also can vary due to dynamic issues such as network and CPU load.

Some complex attacks such as a worm are conceptually recursive attacks. Once the worm has successfully propagated to a new host, it copies in its body, and repeats the process on other hosts. These also generate looping-like variants; the actual path of the worm will vary from site to site.

In addition to the many ways to create variants just stated, the variety of single-point attacks is an additional issue. Single-point attacks are the building blocks of attack scenarios, and by substitution, can give rise to an uncountable number of variants. Because these provide the attacker with an arsenal of exploits to find some means to reach their goal, the sheer number of attacks increases the difficulty of identifying all possible attacks.

Clearly, numerous possible scenario attacks can be created. Specifying each individually would be impossible. For other than the most restricted examples, explicitly describing all variants would be combinatorially infeasible, and doing so requires that all possible variants are known. Because explicit descriptions frequently contain event sequences that are included in descriptions of others, using explicit signatures requires many event sequences.

In order to eliminate these problems we do not use explicit signatures. Rather, we define scenario attacks in terms of the abstract components (concepts) of the attack. Each concept is described locally without specifying with which other concepts it will link. We define only the capabilities required by a concept, and the capabilities it can provide to others. In this way, linkage occurs naturally using the requires/provides metaphor.

Now that we have some understanding of what we mean by a scenario attack, problems in specifying them explicitly, and the basis of our solution, we will describe how we model them.

Our Model

We describe attacks as the composition of abstract attack "concepts". Each concept requires certain capabilities that must occur for a particular instance of the concept to be entailed. Each concept may also provide specific capabilities to other concepts. Also defined for each concept are the required relations between the attribute values of the capabilities. These include details such as operating system, host/port numbers, and cover any required timing relations.

Capabilities

Capabilities are the information required or the situation that must exist for a particular aspect of an attack to occur. For example, a successful telnet login requires a valid username/password and the telnet service to be available on a particular port. It also requires network access to the telnet host, that the host is running, and that the telnet host can authenticate the user. More formally, capabilities are a semantic object that encapsulates a number of semantically typed attributes. These allow descriptions of particular capability instances and a means of relating one capability to another. Capabilities are the atomic elements in our model. The generalized capability is a template for instances of the capability.

Some capabilities form the links between concepts in the model. Others represent the lowest level objects in the model, that is, they define the limit of what is modeled and what is assumed to exist outside the model. For example, rather than model a SynFlood, we assume they exist and consider their effects only.

Related to external capabilities is system configuration information. Because vulnerabilities are tied to specific software versions, and the particular hardware and system configuration determine the scenario attacks that can be successful, detailed information about the computer system must be included in the model. Including configuration moves the model from abstract attacks to those feasible in a specific system. This is particularly important when considering the effects of known vulnerabilities in a particular scenario.

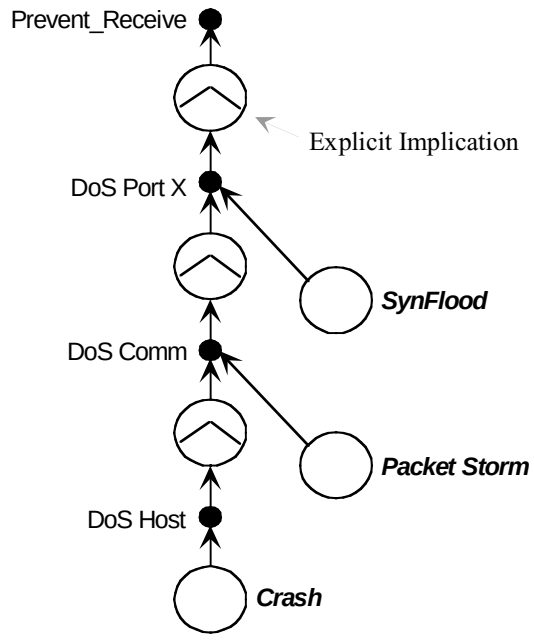
Concepts

Concepts define abstract situations that form subtasks in a scenario attack. Defined for each concept are requirements that must be satisfied if an instance of the concept is to hold. These are Boolean relations on capabilities and configurations. If the requirements are met, then the concept specifies a mapping from the requirements to new capabilities. These capabilities are now available to support other concepts.

Typically concepts refer to subtasks of the scenario, but can also be proscribed policy statements. In this way the model includes compositions that would violate some desired policy. These specify goals of concern. When incorporated into an intrusion detection system, these would be included in the reportable concepts, i.e. those concepts for which we should generate some alert.

Inherent implication. In many cases, the existence of a capability necessarily implies the presence of another. For example, if host A is prevented from sending a packet X to host B, then host B is prevented from receiving packet X. If host B is prevented from receiving packet X, then host B is prevented from sending reply packet Y back to host A. Here the capability of preventing transmission of one packet implies prevention of receipt of the packet by another host and similarly prevents the host from responding to the packet. Such implication chains are common in our attack model. Concepts defining each local implication must be explicitly stated. [see figure 3]

Figure #3: Model Fragment Illustrating hierarchical relation of capabilities and implication chains.



Central to our model are the following features. These are common to all scenario attacks.

Multiple events can provide equivalent capabilities

This is the key concept of our model. An attacker can use different actions to support the same goal. Many different probes will return equivalent information; many different DoS attacks will have the same effect on the target system. By specifying the required capabilities of an attack we capture all possible means of supporting it without specifying explicitly what will provide it. By specifying the provided capabilities of an attack we capture how the attack could be used without specifying explicitly what will use it.

Attack scenarios may have many variants

Because attacks are composed of subtasks, replacing a subtask with any other that provides equivalent capabilities without removing other required capabilities, will result in a valid variant. The more complex the scenario is, the more variants that are possible.

Exploits can be combined in unknown ways to create sophisticated attacks.

Because of the large number of variants, we will not know all possible scenarios explicitly. As new exploits are discovered and added to the model, they can combine in ways not originally realized. A clever series of actions could cause a DoS attack not previously considered.

The strength of this method is that concepts are defined locally without a priori knowledge of the specific attacks that could incorporate the concept. While typically we use the study and analysis of known (or theorized) attacks and vulnerabilities to identify concepts, this method allows us to focus on the abstract relation between events and information rather than the cataloging of specific scenarios.

By defining concepts locally we facilitate the collaboration of security researchers. Anybody can create new concepts and add them to the model. They do not require knowledge of the entire model, only how their concepts will be used locally.

Figure 4(a): An Example Capability Specification – Prevent Packet Send

```
capability capability_x is
      ip_addr:      ip_addr_type;
      port_set:     set of port;
      start_time:   time_type;
      end_time:     time_type;
end.
```

Figure 4(b): An Example Concept Specification – RSH Connection Spoofing

```
concept RSH_Connection_Spoofing is

      requires
          Trusted_Partner:    TP;
          Service_Active:     SA;
          PreventPacketSend:  PPS;
          extern SeqNumProbe: SNP;
          ForgedPacketSend:   FPS;
      with
          TP.service is RSH,             #- The service in the trust relation is RSH
          PPS.host is TP.trusted,        #- The blocked host is the trusted partner
          FPS.dst.host is TP.trustor,    #- The spoofed packets are sent to the trustor
          SNP.dst.host is TP.trustor,    #- The probed host is the trustor
          FPS.src is [ND.host,PPS.port] #- claimed source of forged packets is blocked

          SNP.dst is [SA.host,SA.port]  #- The probed host must be running RSH on the
          SA.port is TCP\RSH,            #-   normal port
          SA.service is RSH,             #-

          SNP.dst is FPS.dest            #- probed host must be where forged packets are sent

          active(FPS) during active(PPS)#- forged packets must be sent while DOS is active
      end;

      provides
          push_channel:      PSC;
          remote_execution:  REX;
      with
          PSC.from    <- FPS.true_src;   #- Capability to move code from attacker to RSH server
          PSC.to      <- FPS.dst;        #-
          PSC.using   <- RSH;            #-

          REX.from    <- FPS.true_src;   #- Capability to execute code on remote host
          REX.to      <- FPS.dst;        #-
          REX.using   <- RSH;            #-
      end;

      action
          true -> report ("RSH Connection Spoofing: TP.hostname")
      end;
end.
```

Concepts

Concepts specify a set of required capabilities, mapping requirements, the provided capabilities and the value assignments for them. Each concept statement begins with the keyword **concept** and an identifying name. Each concept statement will have a **requires** block and a **provides** block, and optionally an **action** block. [See figure 4(b)] The requires section lists the types of capabilities and configuration information the concept uses, along with a label for each instance used in the concept. Capabilities defined external to the model are prefixed with the keyword **extern**. Typically these refer to vulnerabilities originating outside of the model. Next in the requires' **with** section, any relations that must hold between the required capabilities are stated. These form a conjunctive list of Boolean expressions. The language includes a number of operators defined for the data types common to the model. Examples are text comparison operators, set and temporal relations. A number of system functions, such as the `Hostname()` are also included. Additional functions can be specified as required to extend the ontology.

If all the statements in the requires' **with** section are true for some instance of the capabilities, the concept holds and the statements in the **provides** section come into play. Similar to the **requires** section, the **provides** section begins by listing the capabilities needed. In this case however, these are the new capabilities that are provided by the concept. The provides' **with** section specifies value bindings from the required capabilities to the provided capabilities. Conceptually, when capabilities satisfying the requires are present, new capabilities with the bound values are asserted.

Including the keyword **extends** and the name of a defined concept after the concept name allows us to specify simple inheritance. Child concepts inherit all characteristics of the parent, but must satisfy all additional requirements of the child and will provide all additional capabilities of the child. Concepts defined using the prefix **abstract** refer to concepts that are only in the model when extended. For example, the abstract concept "install_tool" is used as a template for concepts involving specific tools.

In order to provide support for using a JIGSAW model for intrusion detection systems, concepts have an optional **action** section. This consists of a list of guarded statements that are evaluated when the concept is active. Actions are used to report information to security personnel, other CIDF compliant IDSs, retarget sensors, or take some automated response. As this is outside the intended focus of this paper, readers interested in this and other details of the JIGSAW language are referred to the forthcoming technical report on the U. C. Davis Global Guard project.

Applications of the Model

Attack Discovery

Our model can be used to discover new attacks. We have found that from a pedagogical standpoint, jus can lead to interesting conclusions. For example, by analyzing the effects of Ethernet MAC control PAUSE packets and the ARP protocol we discovered an exploit to allow unauthorized root logons. Sending a flood of PAUSE packets will cause some switches to block all traffic to a network segment. Such a flood can be induced on certain hosts by crashing the kernel while flooding the computer with packets. While the network is segmented, we can transmit counterfeit ARP reply packets, which would cause packets destined to a NIS server on a different segment to be redirect to a compromised local host running a fake authentication server, allowing us to fraudulently validate the logon. [See figure 5]

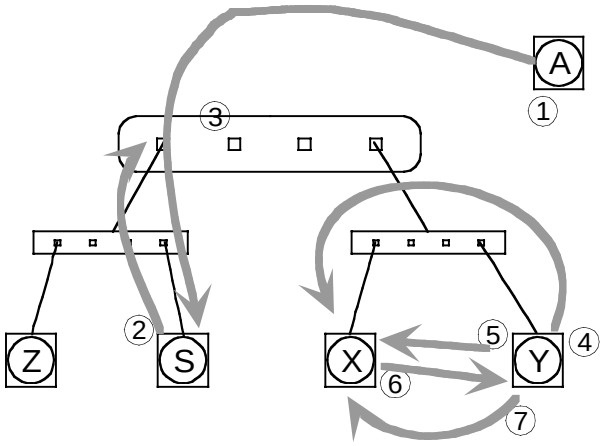
When implemented in software and given an attack concept as a goal, we can automatically discover how attacks could compromise a target. This was readily seen when a proof-of-concept program was tried on a subset of the model.

Mid-level Intrusion Detection

This model provides a natural mechanism for correlating observed activity into scenario attacks. Because the strength of our model is in correlating events, and low-level exploit detection is best done using specialized detectors, we

define our IDS effort as a mid-level system and rely on the output of other systems. Concepts specified as external indicate what sensors will be required by the IDS, and what details the sensor must provide. For efficiency, implementation will require consideration of at what level sensors should be utilized. In practice this will be determined by the available information provided by other IDSs. Those systems that provide signatures of scenarios can be readily added into the model by adding a complementary external concept.

Figure #5: Ethernet PAUSE / ARP reply Scenario.



1) A floods S with packets, then crashes OS.
2) Vulnerability in S causes flood of Ethernet MAC control PAUSE packets to be sent to switch.
3) Vulnerability in switch causes network segments to be isolated from each other.
4) Y sends ARP reply packets that claim MAC of Y is MAC for Z, the NIS server.
5) Y telnets to X.
6) X authorizes telnet via Z, but ARP spoofing redirects packets to Y.
7) Fake authentication process on Y fraudulaently validates telnet for Y.
** SUCCESS! **

Because many low level attacks are unrelated or do not compromise the system, by sending alerts only when events correlate, we can significantly reduce the number of false positives.

As part of the UC Davis Global Guard project we are designing a correlating IDS that incorporates our model and the output of CIDF [8] compliant intrusion detection systems as inputs to our system. CISL[4] reports are wrapped with JIGSAW concepts that describe the capabilities the detected "intrusion" provides. In this way we can interface to any compliant IDS. For example, port scans provide capabilities about what an attacker knows about your system. A buffer overflow provides capabilities including local_execution_as_user and DoS_process.

Extensibility. Because concepts are defined locally, new attacks can be incorporated with minimal effort. Attacks that provide only previously defined capabilities will need no work other than to be wrapped to express those capabilities. If a new capability is also defined, the capability will must also be specified and some concept must be specified to use it. If this is a variant of another capability, inference chaining concepts are all that is needed.

Another advantage of this method is its ability to utilize anomaly and specification based IDS as sensors. By discovering what an anomaly can provide for an attack we can wrap it for use as a sensor. While such meta-analysis of these detection systems is mostly unknown, what is known is promising. For example, the self/nonself work of Forrest[5] primarily detects buffer overflows. Depending on the specific process involved, these can predict remote-to-user, local-to-user or local-to-root attacks. Similarly Tripwire[9] can be used to identify capabilities linked to changes in specific files (for example .rhosts). Similarly the specification based intrusion detection work of Ko[10,11] can be wrapped to state the capabilities a particular violation of the specification would provide.

Prediction. An interesting result of our model is that when a concept holds, we can state that the capabilities provided exist, not that they have been acted upon. For example, if a concept in-

dicting that a particular sensitive file could be read, we do not yet know that it has been, or even if the attacker knows they have this ability. This gives our IDS the ability to predict what the attacker could do, and to provide some level of warning to prevent the event from occurring.

Automated Response. In an IDS context, our model extends readily to support limited automated response. By removing a capability that supports a concept we can remove the capabilities that the concept provides. Because the model is defined in terms of capabilities, we have already defined possible actions. In the previous example, the read could be prevented by automatically encrypting the file, changing permission bits, terminating the attackers connection, etc. We must stress that such responses are "authorized" attacks and will have ramifications perhaps not unlike true attacks. Implementers must be careful to accurately evaluate the cost of any action, and be aware that the automated response may be the true goal of the attacker.

Related work

Other work in describing scenario attacks has been limited to descriptions of explicit signatures of scenarios without consideration of how they might compose into more elaborate scenarios. [13] use descriptions of limited scenarios specified in a C-like language, STATL, in their intrusion detection system, NetStat. Attacks are defined in terms of specific events indicating state transitions. The terminal state in a STATL description, when reached implies the attack has occurred. Currently, over 50 attack signatures are defined for NetStat. Each of these are distinct specific signatures. Any correlation of events is explicitly defined and limited to the individual signatures included. While some variants are captured by the descriptions, any new scenario will require a complete new signature.

The IDIOT project[12] uses colored petri nets[32] to represent signatures of attacks. The petri nets specify a partial ordering of the UNIX commands that are executed in an attack. This is different from our work in that signatures are in terms of specific commands; there is no linkage of attacks; no local specification of attack components, and no notion of the requires/provides ontology. The significance of IDIOT to this project is that there is a similarity between our linkages of concepts and colored petri nets. Petri nets are a powerful way of modeling systems and have been extended in a multitude of ways to handle a variety of systems.

Configuration checkers such as COPS[3] and SATAN[2] use a number of tools to test for system vulnerabilities. These are interesting because they do not just look at single host vulnerabilities, but vulnerabilities of a networked system. Besides looking for specific listed vulnerabilities these incorporate search based vulnerability analysis from SU-Kuang[1] and Net-Kuang[14]. These search for configuration settings that cause an exploitable vulnerability and are not limited to simple vulnerabilities in isolation, but look for sequences of events that could be exploited. Unfortunately, they have only focused on a few high-level vulnerabilities. Also, their model of UNIX security is intrinsic to the tools' code, and therefore not available for other security uses.

Future Work

This model is incredibly rich and suggests many directions for work. A few are described here.

Attack Generation. A related and somewhat shadowy application of our model is in the automated generation of attacks. It differs from attempting to identify vulnerabilities in that it is not given complete knowledge of the system, and from an IDS in that rather than sensors, it must use attack scripts to interact with its environment. Such a system would probe and attack repeatedly until it found a scenario that reached its goal. While this tests the limits of present day planning and expert systems, limited simulations of this have been promising, and we have proposed using it as a tool in evaluating IDSs.

Formal Theory of Attacks. By modeling attacks formally we can prove statements about the attacks such as the absence of a scenario that can attain some goal, or that removal of a vulnerability will (or will not) eliminate the potential

for any attack to reach a particular goal. We have begun looking at notation and axioms toward this end in the near future.

Nondeterminism. Attacks do not succeed every time. In order to function with real world attacks, an attacker must contend with many unpredictable situations. For example, exploiting race conditions typically require multiple attempts before succeess. DoS attacks that use packet floods to consume resources contend for these resources with other packets on the network. These attacks only lower the probability of a non-DoS packet getting through. Also, for some attacks service cannot be more than degraded – the attack may need to get packets through itself.

In order to construct a successful attack, we need to properly evaluate the likelihood of its success. This can determine which resources and how many are used. Consider a DoS attack on a large e-commerce site. If the attack cannot consume sufficient bandwidth the attack will be a failure. Similarly, if the attack uses to many resources or uses them improperly the likelihood the attack will be detected early will increase. A sophisticated attacker would consider the likelihood of success vs. the likelihood that the attack would be detected.

Conclusions

Traditional methods of modeling attacks fail to adequately capture the complexity of scenario attacks. Methods limited to describing simplistic signatures or single-point exploits are not capable of generalizing to unknown attacks. The require/provides model presented here eliminates these problems and can utilize the strengths of earlier methods. Because attack concepts are defined locally the model can be constructed without prior knowledge of the attacks that will be covered by the model. The JIGSAW language provides a convenient method for constructing the model and can be used to construct mid-level intrusion detection systems

References

[1] Baldwin, R. W. (1991) Kuang: Rule-based security checking. In Kolstad, Rob. Daemons and dragons: the COPS security auditor. (tutorial) UNIX Review v9.n3 (March, 1991)

[2] Farmer, Dan, and Venema, Wietse (1995) SATAN: Security Administrator's Tool for Analyzing Networks, http://www.fish.com/~zen/satan/satan.html

[3] Farmer, Dan, and Spafford, Eugene. (1990) The COPS security checker system. In Proceedings of the Summer 1990 USENIX Conference, June 1990.

[4] Feiertag, R., Kahn, C., Porras, P., Schnackenberg, D., Staniford-Chen, S., Tung, B. (1999) A Common Intrusion Specification Language (CISL). At http://gost.isi.edu/projects/crisis/cidf/cisl_current.txt

[5] Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T. A Sense of Self for UNIX processes. Proceedings of 1996 IEEE Symposium on Computer Security and Privacy, 1996

[6] Garfinkel S. and Spafford, E. (1997) Practical Unix and Internet Security, 2nd ed., O'Reilly and Associates, pp. 584-87

[7] Guha, B.; Mukherjee, B. (1997) Network security via reverse engineering of TCP code: vulnerability analysis and proposed solutions. IEEE Network, vol.11, (no.4), IEEE, July-Aug. 1997. p.40-8.

[8] Kahn, C., Porras, P., Staniford-Chen, S. and Tung, B.(2000) A Common Intrusion Detection Framework. Submitted to the Journal of Computer Security.

[9] Kim, G. and Spafford, E. H. (1997) Tripwire: A Case Study in Integrity Monitoring. In D. Denning and P. Denning, editors, Internet Beseiged: Countering Cyberspace Scofflaws, Addison-Wesley, 1997.

[10] Ko C.C.W. (1996) Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach, Ph.D. Thesis, University of California at Davis, August 1996

[11] Ko, Calvin, Fink, George, and Levitt, Karl (1997) Execution Monitoring of Security-critical Programs in Distributed Systems: A Specification-based Approach, In Proceedings of the 1997 IEEE Symposium on Security and Privacy, pp. 134-144

[12] Kumar, S. and Spafford, E. H (1994) A Pattern-Matching Model for Intrusion Detection PROCEEDINGS OF THE NATIONAL COMPUTER SECURITY CONFERENCE; pp. 11-21; Baltimore, MD; Coast TR 95-06

[13] Vigna G. and Kemmerer, R., (1998) NetSTAT: A network-based intrusion detection approach, Proceedings of the 14th Annual Computer Security Applications Conference, Scottsdale, Arizona, December 1998

[14] Zerkle, Dan and Levitt, Karl (1996) NetKuang – A multi-host configuration vulnerability checker. In Proceedings of the 6th USENIX UNIX Security Symposium. San Jose, CA. July 1996, p.195-2