Automatic Configuration Vulnerability Analysis

Sudhakar Govindavajhala Andrew W. Appel
Princeton University
{*sudhakar,appel*}*@cs.princeton.edu*

February 5, 2007

Abstract

We have constructed a logical model of Windows XP access control, in a declarative but executable (Datalog) format. We have built a scanner that reads access-control configuration information from the Windows registry, file system, and service control manager database, and feeds raw configuration data to the model. We found a surprising result: commercial software from major vendors routinely has user-to-administrator privilege-escalation vulnerabilities that result not from buffer overruns (or other bugs inside the software) but just from misconfigurations of permissions and registry entries. Our scanner and analyzer run efficiently, and quickly detect these configuration bugs. Furthermore, our new Windows model can be combined with previous models of Unix, firewalls, and CERT advisories to give a more accurate global picture of the vulnerabilities in a heterogenous enterprise network. Our tool could be used by software vendors (and system integrators) to improve their installation configurations and by sysadmins for day-to-day vulnerability analysis.

1 Introduction

Unix has a simple access control model with three privileges: read, write and execute (plus the sometimes mysterious setuid [5]) given to users, groups, and others for operations on just a few kinds of objects (files, directories, etc.). In contrast, Windows access-control is rather more complex, with access-control lists (prioritized "allow" and "deny" by groups) comprising up to 30 different privileges for operations on about 15 different kinds of objects [7]. For example, on a "service" object one can have the privilege "change what program.exe is run to effectuate the service."

In this paper we will show that a formal model of Windows access control can be applied to analyze the configurations of real machines. What we have learned from running our model is that ordinary professional software developers at commercial software vendors (including Adobe, AOL, Macromedia, Microsoft, IBM, Trend Micro, Symantec, and Cisco) have difficulty in evaluating the consequences of the access-control configurations that they choose for their software and services. The consequence is that commercial software can and does have exploitable privilege-escalation vulnerabilities (user-to-administrator, guest-to-any-user) caused by access-control misconfiguration. In addition, individual application developers design configuration of their programs independent of each other. The security of a host is dependent on the configuration of each of these programs; ours is the first tool a system integrator can use to find configuration vulnerabilities in the totality of a software installation.

Our logical model of Windows access-control is expressed as inference rules in Datalog which are directly executable in a Prolog system. We have a scanner that reads relevant parts of the Windows registry, file system, and service control manager database on a given host to provide input to our logical model. The model runs, and prints out a list of privilege-escalation vulnerabilities, each one with a trace of how each vulnerability might be exploited.

2 Datalog overview

Our system adopts Datalog as the modeling language for elements in the analysis. Previous work has shown that Datalog's declarative semantics is sufficient for encoding access control policies [11] and for describing the interactions of operating systems, network firewalls, and file systems [21]. The declarative nature of a Datalog model allows us to cleanly specify the semantics of something even as complex as Windows XP security configurations. Datalog is efficient to evaluate: in theory, bounded by polynomal time; in practice in our application, less than a second to analyze real installations.

Syntactically, Datalog is a subset of Prolog with limited forms of clauses. The reasoning rules in our system are declared as Datalog clauses. A *literal*, $p(t_1,\ldots,t_k)$ is a predicate applied to its arguments, each of which is either a constant or a variable. In the formalism of Prolog, a variable is an identifier that starts with an upper-case letter. A constant is one that starts with a lower-case letter, or that is `'quoted'`. Let $L_0,\ldots,L_n$ be literals. A reasoning rule in Datalog is represented as a Horn clause: L_0 `:-` L_1, $\ldots$, L_n. Semantically, it means if $L_1,\ldots,L_n$ are true then L_0 is also true. The left-hand side is called the *head* and the right-hand side is called the *body*. A clause with an empty body is called a *fact*. A clause with a nonempty body is called a *rule*. A Datalog program is a set of facts and reasoning rules to infer from these facts. The execution of a Datalog program infers from these facts using the reasoning rules.

Prolog permits data-structure trees in places where Datalog allows only constants; and Prolog has control-flow features (such as *cut*). Prolog is Turing-complete (it is not polynomial-time like Datalog), and is less declarative in nature (order of rules matters). We actually use a slight superset of Datalog (we permit lists and other data structures, not just constants, in the input—though no new lists are constructed at run time); this increases its expressiveness without sacrificing its declarative or polynomial-time properties.

3 Objects in the Windows XP model

In Unix, one has to deal with various operating system objects like files, directories, threads and processes. One uses primitives like locks, message queues and semaphores for interprocess communication. One uses sockets for network programming. Standard textbooks provide excellent introduction to Unix and how to use these objects [2, 28]. In addition to these mechanisms, Windows provides other primitives, the most important of which are the *registry* and *services*. We will summarize these; see Russinovich *et al.* [25] for a detailed treatment.

Registry. The Windows registry is a centralized hierarchical database to store configuration information for the operating system and for applications and services running under Windows. The registry stores a wide range of configuration settings, from boot parameters to user desktop settings to program settings. If an adversary can overwrite the contents of a sensitive key with the path of his library or executable, he could cause his code to be executed [30, 17]. Because of the registry, each individual application does not have to maintain its configuration in application specific configuration files.

Each entry in the registry is indexed by a key. The keys are hierarchical in structure, just like files and directories. $A\backslash B$ refers to the key B that is a subkey of the key A. Programs often store application-specific information under the root key `HKEY_LOCAL_MACHINE` (`HKLM`). We do not find it necessary to model the hierarchical structure; for us $A\backslash B$ is simply an atomic literal.

Each registry key entry in the registry has a security descriptor that determines who can perform what access to the registry entry. The operations that can be controlled by access-control on registry keys are: reading a key, writing a key, deleting a key, enumerating subkeys, adding a subkey, requesting notification for changes to this key or its subkeys.

We consider a registry key to be *sensitive* in an installation if its contents is a name ending .exe, .dll, .bat, and so on. We ignore nonsensitive keys, and we model each sensitive key as a Datalog *fact* such as,

```
resource(registry,
  registryKey('hklm\SOFTWARE\Microsoft\Windows\CurrentVersion\Telephony'),
  acl( ... ) ).
```

Here we have elided the security descriptor (`acl`); we describe access-control-lists below.

Services. Every operating system has a mechanism to start processes at system startup, providing functionality not tied to any particular user. For example, when the operating system boots up, one would want the network programs and the web server program to to start automatically, even if no user has logged on. Windows services are similar to Unix daemon processes, but more general: services are first class objects, which allows general and flexible access control. The configuration on each service comprises two components: properties of the service and access control on the service. The properties of a service include the user account to instantiate the service, whether the service is started by the operating system during boot or manually by the user, the binary path to the executable, and the dependencies on other services. The operations that can be controlled by access-control on services are: starting a service, stopping a service, reading the configuration of a service, modifying the configuration of a service, querying the status of a service, enumerating other services that are dependent on this service, pausing or continuing a service.

```
resource(service, 'DHCP', acl( ... ) ).
```

It turns out that the *current* properties of a service are almost irrelevant to our analysis, so we omit them from our model; what matters are the *permissions* to modify the properties. If an untrusted user is granted “permission to configure the service” (which would be manifest in the contents of the `acl` for that service, elided above), he could take over the administrative account of the host. This is done by configuring the service to run an attack program under an administrative account and restarting the service. The following commands typed in a Windows Command Shell achieve this:

```
$ sc config weakService binPath=c:\attack.exe obj=".\LocalSystem" password=""
$ sc stop weakService
$ sc start weakService
```

Security Identifiers. Windows uses *security identifiers (SIDs)* to identify various entities such as users and groups that perform actions in a system. When a SID is displayed in clear text, each SID carries an *S* prefix, and its various components are separated with hyphens: *S-1-5-21-346327843-89743984-384343-1128*. Some standard SIDs are: `S-1-1-0` (the group *Everyone*), `S-1-5-11` (the group *Authenticated Users*), `S-1-5-18` (the account *Local System*), `S-1-5-19` (the *Local Service* account), `S-1-5-20` (the *Network Service* account), `S-1-5-32-544` (the *Administrators* group).

Accounts. Users belong to groups, and have privileges. The registry contains information about each user (i.e., account) which we model as Datalog facts such as,

```
user('Administrator',
  'S-1-5-21-3034706114-1836988657-2654252735-500').

userInfo('S-1-5-21-3034706114-1836988657-2654252735-500',
 groups([
  sidAndAttributes('S-1-1-0',
   ['SE_GROUP_ENABLED', 'SE_GROUP_ENABLED_BY_DEFAULT', 'SE_GROUP_MANDATORY']),
  ... ]),
 privileges(['SeAuditPrivilege', 'SeBackupPrivilege', ... ])
).
```

Where we use ellipses (...), the actual data has much longer lists of groups and privileges.

System Accounts. The user mode components of the kernel of the Windows operating system (e.g., *csrss.exe* and *lsass.exe*) run under the *Local System* account. This account has complete access to all the resources of the machine and most of the daemon programs run under this account. In Windows XP, Microsoft introduced the accounts *Local Service* and *Network Service* to run parts of the operating system that do not need complete access to the machine's resources. For example, Windows exports its registry over the network. The operating system service responsible for responding to remote registry requests runs as *Local Service*. (Even though *Local Service* does not have complete access to the registry, it can use Windows's delegation facility to access the registry on behalf of the client.) One cannot log in to the accounts *Local System*, *Local Service* and *Network Service*, but can control these accounts from any account in the *Administrators* group.

User Accounts. The *Administrator* account is the account one uses upon first setting up a workstation or a server, before creating any other account. It is a member of the *Administrators* group. Any member of the *Administrators* group has complete control of the machine. Windows has various other groups, such as *Authenticated Users*, *Everyone*, *Server Operators*, *Power Users* , *Network Configuration Operators*. Members of the *Power Users* group can create user accounts, can modify and delete accounts they create, stop and start system services which are not started by default. Any principal who supplied a credential to logon belongs to *Authenticated Users* group. Every machine comes with a built-in account *Guest* which is disabled by default. Once enabled, the user *Guest* does not have to supply a credential. Every principal including the *Guest* user is a part of the group *Everyone*. A principal in the *Network Configuration Operators* group can modify the network configuration. The Microsoft documentation for *Server Operators* group reads: *Server Operators can perform most common server administration tasks* [27]. There is also the notion of *Limited Users*, which determines the set of mandatory groups and privileges given to a new user account by the “User Accounts” dialog box when the “Limited” radio-button is selected. In this article we explore privilege escalations between the different user accounts, system accounts, and groups.

4 Windows Security Overview

We now show how to model the algorithm used by the Windows kernel to determine whether an access should be allowed. In making this decision, the kernel considers the following inputs: *Who is the principal requesting the access?*, *What are the intentions of the principal (specified in the request)?* and *What is the protection on the object to be accessed?*

4.1 User attributes

Account privileges Windows provides a flexible access-control model where an owner of a resource can specify the level of access each user has. This model is useful in protecting access to a single object. However, sometimes users perform operations that have a system-wide impact. For example, the ability to shut down a system or change the time of a system has system-wide impact. The capability to perform these actions must be controlled carefully. Windows uses the notion of *privileges* to achieve this purpose. When a user logs onto a host, after authentication, the system identifies the privileges associated with the user and stores this information in the kernel as a part of the the process control block for the user’s shell.

There are about thirty different privileges in Windows controlling various capabilities such as: replace a process-level token, generate security audits, load/unload drivers, skip access checks, lock pages in memory, force local/remote shutdown of a host, control and view audit messages, change the system time, create a new process token on behalf of any user, act as a part of operating system. Some of these privileges (such as load-driver privilege) allow a principal to directly or indirectly get complete control of the host. In our model, we refer to these privileges as super privileges.

Token A *token* is a per-process (per thread in some cases) data structure, maintained in the kernel as a part of the process control block, that contains the security information. It stores information regarding the user account, a list of account privileges for the user account, a list of SIDs representing the user, groups the user belongs to, the session identifier, and other security related information associated with the process or thread. A token is created when a user logs on to a system and is attached to the initial process (typically `userinit.exe`) that is started on behalf of the user. A child process inherits its parent’s token. When a process makes a certain request (like opening a file in write mode or opening a service to configure its properties), the kernel consults the token of the calling process to determine the privileges of the process.

The Datalog predicate `userToken(Principal, Token)` identifies that the principal `Principal` gets the token `Token` when he logs in. The configuration scanner supplies information (from the registry) regarding the list of users on a host and their credentials.

When a principal accesses an object, the kernel looks up the process token of the process making the request to determine its credentials: the user, the privileges, and the groups the user belongs to. The predicate `processToken(UserSid, Privileges, Groups)` encodes the credentials of the process requesting the access. `UserSid` is the user identity on behalf of whom the process runs, `Privileges` is the set of privileges (like `SeTakeOwnershipPrivilege`, `SeSystemtimePrivilege`) the process holds, and `Groups` is the set of groups the user belongs to. A sample process token looks like:

```
processToken( 'S-1-5-21-1214440339-507921405-1060284298-500',
              privileges(['SeBackupPrivilege',  'SeChangeNotifyPrivilege',
                          'SeSystemtimePrivilege']),
              groups(['S-1-1-0', 'S-1-2-0',
                      'S-1-5-11', 'S-1-5-32-544'])
            )
```

4.2 Modeling requested access

The Windows XP access control algorithm is modeled by the predicates `windowsAccessCheck`, `checkAccessList` and `checkACE`. These predicates take as an argument the access requested. In our model, the `RequestedAccess` argument is an atomic permission like `FILE_READ_DATA` and `FILE_READ_ATTRIBUTES`. In practice, one requests more than one permission, like `FILE_READ_DATA` and `FILE_WRITE_DATA`—to read and write a file. To check for such nonatomic permissions, we will have to call the corresponding access check function on each atomic permission requested. This is a deviation from the way the kernel implements the algorithm. However, this deviation is functionally equivalent when used with atomic predicates. The kernel implementation avoids calling the function more than once for nonatomic permission by doing clever bit manipulation. Since it is not straightforward to do bit manipulation in Datalog, in our model access check can only be called on an elementary permission like `FILE_READ_DATA`. This makes our reasoning rules easier to write and debug and thus increases the assurance of our system. However, Windows allows a principal to request more than one permission—like all of `FILE_READ_DATA, FILE_WRITE_DATA, DELETE, READ_CONTROL, WRITE_DAC`—simultaneously. To model this behavior of Windows, when a principal requests both `FILE_WRITE_DATA` and `WRITE_DAC`, we invoke the predicate `windowsAccessCheck` twice—the first time with `FILE_WRITE_DATA` and the second time with `WRITE_DAC`.

4.3 Security Descriptor

A *security descriptor* is a per-object data structure identifying who can perform what action of the object. A security descriptor can be set on objects like processes, threads, semaphores, sections, waitable timers, registry keys, files and services. A security descriptor consists of the following attributes: a revision number (version of the security model), flags (controlling inheritance characteristics), owner SID, group SID, a discretionary access-control list (specifying who has what access to the object) and a system access-control list (specifying which attempted operations by which users should be logged). For our discussion only the owner SID and discretionary access-control list (DACL) are relevant.

An *access-control list* (ACL) is a list of zero or more *access control entries* (ACEs) which say who is allowed what access to an object. Each positive (negative) ACE grants (denies) specified access to the principal or group referred by the SID listed in this ACE. Each Access Control Entry has a 32-bit mask specifying (bitwise) a set of “standard” and “object-specific” permissions. Standard permissions are common to all ojects: `DELETE`, `READ_CONROL`, `WRITE_DAC`, and `WRITE_OWNER` for deleting, reading the security descriptor, writing the security descriptor and writing the owner of an object respectively. Examples of object-specific permissions include `FILE_WRITE_DATA` (for file objects), `SERVICE_CHANGE_CONFIG` (service objects), and `KEY_WRITE` (registry objects).

An **access control entry** is encoded as the primitive predicate `ace( aceType(Type), aceRights(RightsList), Sid)` that specifies that an access control entry of type `Type` (one of `'ACCESS_ALLOWED_ACE_TYPE'` or `'ACCESS_DENIED_ACE_TYPE'`) grants or denies to the entities represented by the identifier `Sid` the rights specified in the list `RightsList`. The predicate `dacl(AclList)` encodes a **discretionary access control list (DACL)**, where `AclList` is a list of access control entry predicates, storing the entries in the same order as they appear in the security descriptor. We show an example ACE predicate and a DACL with a single ACE:

```
  ace(aceType('ACCESS_ALLOWED_ACE_TYPE'),
       aceRights(['FILE_WRITE_DATA']),
       sid('S-1-5-21-854245398-1637723038-725345543-1003')).

 dacl([ ace(aceType('ACCESS_ALLOWED_ACE_TYPE'),
           aceRights(['FILE_WRITE_DATA']),
           sid('S-1-5-21-854245398-1637723038-725345543-1003'))  ])
```

A **security descriptor** is encoded as the predicate `securityDescriptor(Owner, Dacl)` where `Owner` represents the security identifier of the owner and `Dacl` is a predicate that encodes the discretionary access control list.

4.4 Determining access

The predicate `windowsAccessCheck(Result, ObjectProtection, RequestedAccess, RequestingToken)` models the algorithm the kernel uses in determining whether to permit an `RequestedAccess` access to an object with protection `ObjectProtection` by a process with token `RequestingToken`. The variable `Result` is instantiated to `allowed` or `denied` accordingly. In section 4.4, we described the algorithm the kernel uses to make an access control decision. We now formally describe the algorithm.

Rule 1: No ACL implies no protection. If a file does not have an Access Control List ("Null DACL"), then any access is permitted on the file. The formal rule is:

```
windowsAccessCheck(allowed,
        securityDescriptor( Owner, dacl(null)),
        RequestedAccess,
        RequestingProcessToken).
```

SeTakeOwnershipPrivilege privilege gives write-owner access The privilege SeTakeOwnershipPrivilege in the caller's token gives WRITE_OWNER access to any resource. With WRITE_OWNER permission, one can change the owner SID of a resource to one of the SIDs in the caller's process token. (Technically, the SID in the process token will have to be marked as having the potential for being an owner.) After obtaining the ownership of a resource, the adversary will be able to get full control of the resource by launching further attacks, as described below. Similarly, as we discussed in section 4.1, any super privilege will give the adversary complete control over all the resources of the host. To summarize, the SeTakeOwnershipPrivilege and other super privileges will give the adversary complete control over all resources on a host, thus resulting in system-wide compromise. In our model, we encode these multi-step attacks as a single step:

```
windowsAccessCheck(allowed, SecDescriptor, RequestedAccess,
        processToken(Owner, PrivList, GroupSids,
                     TokenRestrictedSids) ) :-
hasSuperPrivilege(true, PrivList).
        %Check if token has a ‘‘ super’’ privileges
```

Rule 3: Owner always gets access. The owner of a resource always gets "change the ACL" permission and hence can give arbitrary entity arbitrary access. Thus the owner of a resource always get full access to the resource. This is expressed as:

```
windowsAccessCheck(allowed,
        securityDescriptor(Owner, Dacl),
        RequestedAccess,
        processToken(Owner, PrivList, GroupSids,
                     TokenRestrictedSids).
```

Rule 4: Consult the Access Control List If none of the previous rules apply, then the kernel consults the access control list. Each Access Control Entry (ACE) in the access-control list is examined from first to last looking for an entry that denies or allows the action. An ACE is processed if the ACE is an access-deny or access-allowed ACE and the SID in the ACE matches a SID in the caller's access token. If it is an access-denied ACE, then the access is denied. If the ACE is an access-allowed ACE, then the access is allowed. If the end of the list is reached without a matching ACE, the request is denied. Formally, we write this as:

```
windowsAccessCheck(Result,
        securityDescriptor(ObjectOwner, dacl(Acl)),
        RequestedAccess,
        processToken(ProcessOwner, PrivList, Groups,
                     TokenRestrictedSids)   )  :-
       checkAccessList(Result, RequestedAccess, Acl, Groups).
```

The predicate `checkAccessList(Result, RequestedAccess, Acl, SidsList)` models the algorithm the kernel uses to decide whether an access control list `Acl` allows or denies `RequestedAccess` to a principal with `SidsList` as the list of security identifiers of the groups in the process token. This predicate examines the `Acl` from first to last and unifies the `Result` variable to `allowed` or `denied` accordingly. If there is no access control on the object, then the request is granted. If the end of the list is reached, then access is denied.

The predicate `checkACE(Result, AceEntry, Access, SidsList)` means that an elementary access control decision, using the access control entry `AceEntry`, for a request `Access` by a principal whose list of security identifiers of the groups in the process token is `SidsList`, is `Result` (one of `allowed`, `denied`). Formally, we write this as:

```
checkAccessList(allowed, Access, dacl(null), SidsList).
checkAccessList(Result, Access,
  dacl(acl([AclHeadEntry| AclTail])), SidsList) :-
      (
        checkACE(Result, AclHeadEntry, Access, SidsList);
        % ; is Prolog OR operator
        checkAccessList(Result, Access,
                        dacl(acl(AclTail)), SidsList)
      ).
% An empty access control list denies access
checkAccessList(denied, Access,
  dacl(acl([])), SidsList)

checkACE(allowed,
         ace( aceType('ACCESS_ALLOWED_ACE'), AceRights, Sid),
         Access, SidsList) :-
         accessInAceMask(Access, AceRights),
         sidInGroup(Sid, SidsList).

checkACE(denied,
        ace( aceType('ACCESS_DENIED_ACE), AceRights, Sid),
        Access, SidsList) :-
        accessInAceMask(Access, AceRights),
        sidInGroup(Sid, SidsList).
```

5 Modeling privilege escalation

We use the predicate `resource(Type, Name, Dacl)` to identify various resources on a host. `Type` indicates the type of the resource—it is one of `service`, `registry` and `file`. `Name` identifies the resource and `Dacl` is the protection on the resource. By scanning the host, one could generate the list of all the resources on a machine. The predicate `userToken(Principal, Token)` identifies that the principal `Principal` gets the token `Token` when he logs in. We generate this predicate for each user on the machine. `canWrite(Principal, resource(Type, Name, Dacl))` is a derived predicate that specifies that principal `Principal` can write the resource of type `Type` (`service`, `registry`, `file`) identified by `Name` and with a security descriptor `Dacl`.

It is well known that the ability to overwrite a file will allow an adversary to launch Trojan-horse attacks. Similarily, if the adversary can overwrite the registry keys that store program paths to be executed during log on process and boot process, the adversary can launch Trojan-horse attacks. If the adversary is allowed the permission to configure a service, then he can configure a malicious program to run under an administrative account and get complete control of the host. If the adversary has `FILE_WRITE_DATA` permission on a file, he could overwrite the file. If the adversary has `KEY_SET_VALUE` permission on a registry key, he could overwrite the contents of the key. If the adversary has `SERVICE_CHANGE_CONFIG` permission on a service, he could reconfigure the service. We encode these as:

```
canWrite(Principal, resource(file, Name, Dacl)) :-
        userToken(Principal, ProcessToken),
        windowsAccessCheck(allowed, Dacl,
                           'FILE_WRITE_DATA', ProcessToken).

canWrite(Principal, resource(registry, Name, Dacl)) :-
        userToken(Principal, ProcessToken),
        windowsAccessCheck(allowed, Dacl,
                           'KEY_SET_VALUE', ProcessToken).

canWrite(Principal, resource(service, Name, Dacl)) :-
        userToken(Principal, Token),
        windowsAccessCheck(allowed, Dacl,
                           'SERVICE_CHANGE_CONFIG', Token);

canWrite(Principal, resource(Type, Name, Dacl)) :-
        userToken(Principal, Token),
        windowsAccessCheck(allowed, Dacl,
                           'WRITE_DAC', Token);
```

The `WRITE_DAC` rule above is interesting. If an adversary is exploiting a `WRITE_DAC` misconfiguration, then the adversary will have to first modify the access control on the object to give him access and then overwrite the resource. For the sake of simplicity, we model this multi-step attack as a single step.

`trusts(Principal, Resource)` is a predicate that specifies that `Principal` trusts `Resource`.

If a principal executes code in a file, he trusts the file. It is impossible to statically identify which files a user trusts. We make the conservative assumption that any file in system directories such as `C:\Program Files` and `C:\Windows` is executed by all users of the host and in particular by the administrator. For all other executable files, we assume that these files are at least executed by the owner of file. Similarly, we assume that sensitive registry keys are trusted by at least the owner of the registry key. If a principal is granted the permission to configure a service, the principal can choose the program to be instantiated when the service is run and the account the service under. In particular, principal can instantiate a malicious program to run under an administrative account. Thus, any principal who can configure a service can get administrative access to a machine; an administrator should trust any service resource. If a principal `Target` trusts a `resource(Type, Name, Dacl)` and if a principal `Attacker` can write to this resource, then the adversary `Attacker` can launch a privilege escalation to `Target`. This is formally encoded as:

```
trusts(Administrator, resource(service, Name, Dacl)).

execCode(Attacker, Target) :-
        canWrite(Attacker, Resource),
        trusts(Target, Resource).
```

6 Security bugs found

We used our tool to see how software from various vendors was configured in the default installation. In all the cases we describe, we discovered the error and reported it to the vendor, who had not previously received reports of it; except that one of Macromedia's and all of Microsoft's configuration errors were corrected by their vendors in security patches before we reported them.

Figure 2 shows how unprivileged users on actual Windows XP hosts could obtain administrator privileges through several paths. We found three classes of bugs: file system misconfigurations, registry misconfigurations and service misconfigurations. In general, these misconfigurations are traceable directly to standard installations of commercial software.

Service misconfigurations. Several vendors poorly applied the Windows access control model to their services; a common mistake is to assign the SERVICE_CHANGE_CONFIG permission indiscriminately to services. The Windows XP documentation states, "…because this grants the caller the right to change the executable file that the system runs, it should be granted only to administrators" [9]. But that warning fails to explain clearly that permission to configure a service allows both setting the executable *and* selecting the account under which the service runs, e.g., change the "run-as" account to *Local System* [8, 17]. From *Local System*, all things are possible (including installing password sniffers to launch further attacks in the guise of any ordinary user).

We found that in the default configurations of Windows XP (until patched in August 2004), the *SSDP Discovery Service* (marked as **SSDP** in Figure 2) and the *Universal Plug and Play Device Host service* (**UPNP** in Figure 2) granted "permission to configure the service" to the *Authenticated Users* group. A normal unprivileged user is a part of the *Authenticated Users* group and hence a normal user can configure the executable and the account under which these services run. Then, the adversary needs to make the service reload the new configuration. He needs to wait for the service to be restarted (he could, for example, force the system administrator to reboot the machine by consuming too many resources so that the system is too slow to respond). We also noticed that usually when a principal is granted the SERVICE_CHANGE_CONFIG permission, he is also granted SERVICE_STOP and SERVICE_START permissions. One could use the Service Controller (`sc.exe`) to trivially reload the service in the new configuration as follows:

```
$sc config weakService binPath=c:\attack.exe obj=".\LocalSystem" password=""
$sc stop weakService
$sc start weakService
```

Via the same SERVICE_CHANGE_CONFIG mechanism, the following (Windows XP Professional) access-control decisions gave paths from *Local Service*, *Network Service*, *Network Configuration Operators*, and *Server Operators* to *Local System*: The *Local Service* account had permission to configure the *Universal Plug and Play* service (labeled **UPNP** in Figure 2), *Smart Card Services* (**SCardSvr**) and the *Smart Card Helper Service* (**SCardDrv**). The *Network Service* account had permission to configure the *Microsoft Distributed Transaction Coordinator* service (**MSDTC**). The *Server Operators* group had permission to configure **uPnP**, *Simple Service Discovery Protocol* (**SSDP**), *NetBios over TCP/IP* (**NetBT**), and *Smart Card Services* (**SCardSvr**). The *Network Configuration Operators* group had permission to configure the *Dynamic Host Configuration Protocol* (**DHCP**), **NetBT**, and **Dnscache** services. This defeats the principle of least privilege that was the motivation for creating *Local Service* and *Server Operators*. If the adversary

Program Name	Vendor	Mechanism	#instances
Lotus Notes	IBM	File	354
VPN Client	Cisco	File	18
PC-cillin Anti-virus2006	Trend Micro	File	103
PC-cillin Anti-virus2006	Trend Micro	Service	6
Illustrator	Adobe	File	170
Anti-virus	Symantec	File	6
AOL Messenger	AOL	Registry	2
Dreamweaver	Macromedia	Service	1
Flash	Macromedia	File	1
Windows XP	Microsoft	Service	7
Windows XP	Microsoft	Registry	40

Figure 1: A summary of vulnerabilities discovered by our tool. The column “mechanism” reflects which operating system object the software misconfigured. The column “instances” reflects the number of objects that lead to a privilege-escalation vulnerability. For example, Lotus Notes had 354 executable files that are writable by any untrusted user on the host. Further details are available in Govindavajhala’s PhD thesis [14].

were to find a buffer overflow bug in a program running as *Local Service*, this escalation path enabled the adversary to take complete control of the host.

Finally, although Microsoft describes *Power Users* as “includes many, but not all, privileges of the Administrators group,” [4, page 31] it is well known that there are many privilege-escalation paths from *Power Users* to *Local System*; we have found more than 20 with our tool.

Other vendors’ software also had access-control configuration bugs in their services: The *Everyone* group was granted the permission to configure the *Macromedia Licensing Service*, installed by **Macromedia’s** Dreamweaver program.

Registry misconfigurations. The standard configuration of **AOL** included a registry entry binding the name of a DLL file to be loaded and executed (in some circumstances) by the AOL software. The access permissions permitted any user to write this entry; the attacker could substitute the name of his own DLL and wait for some other AOL user to execute it, thereby accomplishing a user-to-user privilege escalation. We found similar weaknesses in the registry configurations of several other vendors.

File misconfigurations. In addition to Trojan horses via service configuration, some vendors’ software is vulnerable to a more traditional kind of file-system-based Trojan horse: The *Everyone* group was granted the permission to write to 170 executable (.EXE and .DLL) files from **Adobe**. The adversary can write to these files and wait for a system administrator or other user to execute the files. *Everyone* was granted permission to write to 103 files of the Internet Security 2006 virus scanner from **Trend Micro**. *Everyone* was granted permission to write 354 files of Lotus Notes from **IBM**.

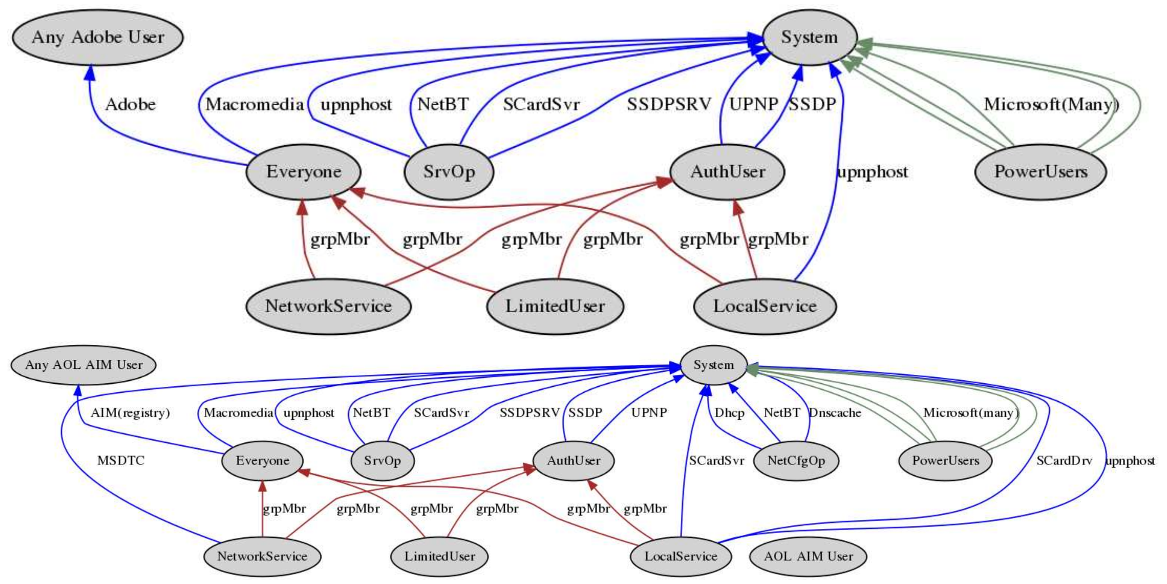

Figure 2: Privilege escalations in a single host of a large, professionally managed network (above) and in a default configuration of Windows XP prior to Service Pack 2 with AOL installed (below). These vulnerabilities originated from the installation of commercial software. LimitedUser is an unprivileged user, NetworkService and LocalService are low-privileged accounts used to run some operating system programs. Everyone, SrvOp (Server Operator), NetCfgOp (Network Configuration Operator), AuthUser (Authorized User) and PowerUsers are groups. The arcs labeled grpMbr show that the user is a member of the group. All other arcs show privilege escalations. There are about thirty escalation paths from PowerUsers to System.

7 Implementation and performance measurements

Our tool to analyze a Windows host comprises two phases: configuration collection and analysis. Our configuration collector ("scanner") is a C++ program that uses several Windows APIs to read the file system (directories and their access control lists) and registry (services, users, and sensitive keys). The scanner produces Datalog facts such as the `resource(...)` facts shown in the previous section.

Our scanner takes about 5 minutes to run on a typical host (2.2GHz Pentium IV with 512MB RAM), of which 6 seconds is the scan of the Service Control Manager, about 2 minutes is the registry scan, and about 2 minutes is the file-system scan. These scans are I/O-bound and it is unlikely that they can be made substantially faster.

The scanner produces output in the form of an ASCII file of Datalog facts. To keep this file small, there are many facts about the registry and file system that our scanner need not report. For example, many objects owned by the Administrator's account have security descriptor configurations that prevents anyone other than an Administrator from modifying it. Such objects would not aid the adversary in a privilege escalation attack and can be safely ignored so that our analysis phase is efficient. In addition, as we have explained, the scanner reports only "sensitive" registry keys (those whose contents appear to be the names of executable files). With these optimizations, the typical scanner output is about 92 kilobytes (registry) + 10 kilobytes (services) + 6 megabytes (file system).

Our scanner can run with almost no privileges (i.e., as a Limited user) and still gather sufficient data to

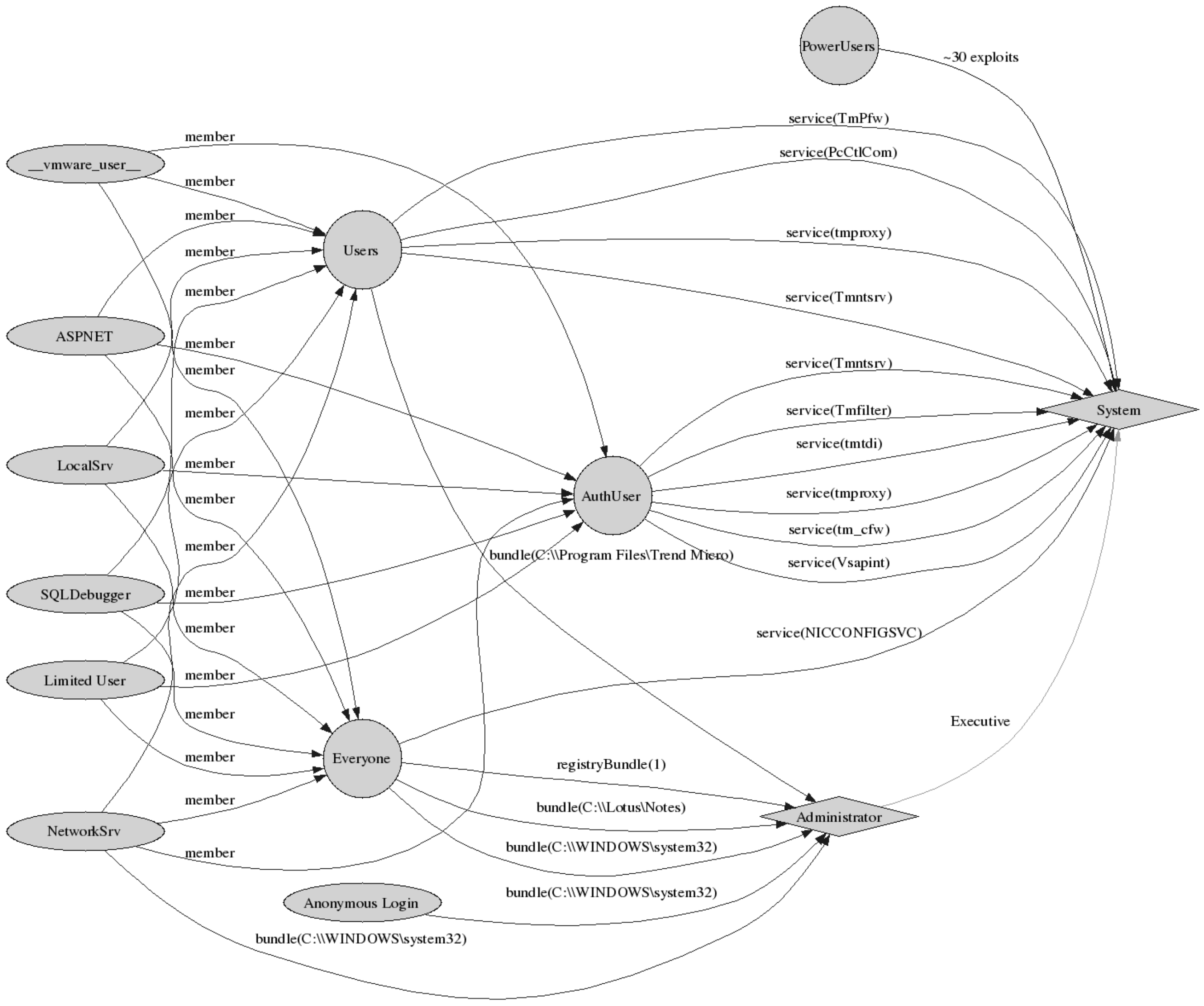

Figure 3: Privilege escalations between various users and groups in a single Windows host running just Lotus Notes, PC-Cillin antivirus, and VMware. The vulnerable software is Lotus Notes and PC-Cillin antivirus. Installing a trusted anti-virus program actually made the machine more insecure! *This graph was automatically generated by our tool.* The ovals in the extreme left column represent various users installed by the operating system or other software. The circular nodes represent the various groups on the host. The diamond nodes represent the administrative accounts on the machine. Each arc represents a privilege escalation path. As one can see the privilege-escalation graph is dense. In the limited space available on this page, it is a significant challenge to make the figure show complete details and yet be readable. This host is a machine managed by professional system administrators as a part of a large enterprise network.

perform a useful analysis. This is desirable, because a system administrator might legitimately worry about running yet another high-privilege software tool that might harm his system. It also means, of course, that all that raw information was already available to sophisticated attackers. We discovered only one weakness—in the System Restore program—that requires administrative access to be discovered.

One might think that a Limited user would be unable to scan certain (protected) subdirectories looking for writable executable files that could be used as Trojan horses. But these protected subdirectories are also difficult for the hypothetical attacker to exploit, for the very same reason that Limited users cannot traverse them. This does not cover *all* cases—a more accurate scan can always be made by running the scanner with Administrator privileges—but it illustrates why a Limited-user scan is still useful.

The data from the configuration phase is fed to the analysis engine, which could potentially run on a different host. We use the XSB Prolog engine, in part because it has efficient tabling. That is, it memoizes all computed facts so that they need not be recomputed, and this give the polynomial-time bound for Datalog. We compute all possible instances of `execCode(Attacker,Target)` for all possible Attackers and Targets. Each (Attacker,Target) pair is a privilege-escalation path, shown as an edge in the graphs of Figures 2 and results:windows-summary. Running the Datalog model in XSB to compute all privilege escalations takes less than one second.

8 Limitations of the model

Our model is, necessarily, an abstraction of what really happens in a Windows XP system. It may over- or underestimate the possible privilege escalations for several reasons.

Less-fruitful object classes. After some preliminary study, we concluded that the majority of configuration vulnerabilities arise from just three classes of objects: files, registry keys, and services. In principle, we could extend our model to cover objects such as processes, semaphores, mail slots, sockets, and so on, and this might uncover more privilege escalations.

Undocumented and ill-documented features. We built our model based on documentation, discussion with Windows experts, and behavioral analysis of the operating system. Almost assuredly Windows can exhibit obscure behavior that we have not understood and modeled.

Obscure service-specific permissions. Our system correctly identifies as security bugs misconfigurations in `FILE_WRITE_DATA`, `SERVICE_CHANGE_CONFIG`, `DELETE`, `WRITE_DAC`, `WRITE_OWNER`, and `KEY_WRITE`; the consequences of misconfigurations in these permissions are uniform across all entities of the specific object type and are very well understood. However, in some cases, the consequences of some permissions are not very well defined. For example, the meaning of `SERVICE_USER_DEFINED_CONTROL` is service-specific and is usually not well documented. Our system does not model them.

Static estimation of user capabilities. Similarly, because of the complex access control semantics of Windows, it is very hard to correctly determine all the capabilities of a user statically. We could not find documentation that describes how the kernel computes the dynamic capabilities when a user logs on to a host. Our static tool makes a best effort to guess these dynamic properties. Thus it is possible that our tool has missed some security attacks either because the dynamic process token is not correctly modelled in our system or because the meaning of a particular access permission is not well documented. With the lack of documentation, we find it very hard to perform a rigorous evaluation of these false negatives. In this tool, we have covered all the mechanisms that are discussed in vulnerability disclosure mailing lists.

Unrealizable attacks. In some cases our model will report privilege escalations that cannot be effectively realized. For example, a permission weakness in System Restore can be attacked if the adversary can guess a file name *and* the system then needs to be restored from backup; but perhaps the file name is too difficult for the adversary to guess, or the system is never restored. Our model will report the attack path all the same.

Bugs within executable software. Applications and the operating system itself can contain bugs such as buffer overruns. This is not a limitation of our model; in fact, in previous work [21] we have shown how to integrate CERT advisories (in OVAL format) into a Datalog model of an operating system (in that case a much simpler model, for Unix).

9 Related work

Datalog has been used as a security language for expressing access control policies [11]. The efficiency of Datalog and existing off-the-shelf Datalog evaluation engines [23] makes it readily usable in practice.

In this paper we have analyzed applications that (inadvertantly) grant too many privileges. But on the other hand, many application programs demand too many privileges, more than strictly necessary to access the data on which they operate. Chen *et al.* [6] explain why this is harmful (users demand and receive the privileges necessary to run the applications, which then gives them privileges to do harm) and have built an analysis tool for finding such situations. We believe our detailed model of Windows access-control could also be used in the context of “least-privilege-incompatibility” analysis.

Modeling vulnerabilities and their interactions can be dated back to the Kuang and COPS security analyzers for Unix [3, 12]. Recent works in this area include the one by Ramakrishnan and Sekar [22], and the one by Fithen et al [13]. A major difference between our work and these works is the application to Windows platform. There is a long line of work on network vulnerability analysis [31, 29, 24, 26, 1, 20].

Ritchey and Amman proposed using model checking for network vulnerability analysis [24]. Sheyner *et al.* extensively studied attack-graph generation based on model-checking techniques [26]. These approaches suffer from problems of state space explosion. While it is foreseeable that these approaches can be made to work, it has not been demonstrated that the approach scales for large networks. Amman *et al.* proposed graph-based search algorithms to conduct network vulnerability analysis [1]. This algorithm is adopted in Topological Vulnerability Analysis (TVA) [15], a framework that combines an exploit knowledge base with a remote network vulnerability scanner to analyze exploit sequences leading to attack goals. However, it seems building the exploit model involves manual construction, limiting the tool’s use in practice.

Intrusion detection systems have been widely deployed in networks and extensively studied in the literature [10, 19, 16].

The MulVAL system [21] used Datalog to express a simple operating-system model (Unix) combined with models of firewalls and bug advisories (in the OVAL and National Vulnerability Database formats) to reason about host-to-host propagation of attacks. The work we report in this paper can be used to extend MulVAL to heterogeneous networks of Windows and Linux machines. We are not aware of prior work that can understand the semantics of disparate operating systems.

Microsoft's *Trustworthy Computing Security Development Lifecycle* project [18] addresses many of the issues discussed in this paper, which explains why the vulnerabilities we describe in Microsoft's own software were closed in mid-2004.

10 Conclusion

Windows XP is a complex beast; to understand its configuration vulnerabilities we had to draw upon many sources. Representing this understanding as a formal model in a declarative language (Datalog) was worthwhile for several reasons. First, the rigor and conciseness of a formal specification language allows a basis on which to discuss and debate the accuracy of our understanding. The model allows us to make predictions that can be tested against the real system.

Second, running the model on real data from actual installations produced unexpected results: the pervasiveness of configuration bugs in default installations of commercial software from many vendors. It is obvious in hindsight that Windows access-control is so complex that programmers and administrators have difficulty understanding and debugging their access-control decisions without tools.

Third, by using the a Datalog framework consistent with that of the MulVAL system [21], heterogeneous multihost networks can be analyzed for problems not limited to single-host configuration vulnerabilities.

Acknowledgments. Discussions with Varugis Kurien, Adrian Oney, Brandon Baker, and Scott Field helped clarify our understanding of the Windows security model. We thank Wayne Boyer, Ed Felten, Xiaolan Zhang, Alex Halderman, John Lambert, Miles McQueen, Michael Steiner, Ruoming Pang, Daniel Dantas, Xinming Ou, and Matt Thomlinson for helpful comments on earlier drafts of this paper. An internship at Microsoft by the first author in the summer of 2005 was a valuable opportunity to gain experience with the APIs used in access control and in querying the Windows registry.

References

[1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[3] R. Baldwin. Rule based analysis of computer security. Technical Report TR-401, MIT LCS Lab, 1988.

[4] Ed Bott and Carl Siechert. *Microsoft Windows Security Inside Out: for Windows XP and Windows 2000*. Microsoft Press, 2003.

[5] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, 2002. USENIX Association.

[6] Shuo Chen, John Dunagan, Chad Verbowski, and Yi-Min Wang. A black-box tracing technique to identify causes of least-privilege incompatibilities. In *Proceedings of Network and Distributed System Security Symposium, 2005*, February 2005.

[7] Microsoft Corporation. Access rights and access masks. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_rights_and_access_masks.asp, October 2005. web page fetched October 9, 2005.

[8] Microsoft Corporation. ChangeServiceConfig. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/changeserviceconfig.asp, 2005.

[9] Microsoft Corporation. Service security and access rights. http://windowssdk.msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/service_security_and_access_rights.asp, October 2005. web page fetched October 9, 2005.

[10] Frdric Cuppens and Alexandre Mige. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 202. IEEE Computer Society, 2002.

[11] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105. IEEE Computer Society, 2002.

[12] Daniel Farmer and Eugene H. Spafford. The cops security checker system. Technical Report CSD-TR-993, Purdue University, September 1991.

[13] William L. Fithen, Shawn V. Hernan, Paul F. O'Rourke, and David A. Shinberg. Formal modeling of vulnerabilities. *Bell Labs technical journal*, 8(4):173–186, 2004.

[14] Sudhakar Govindavajhala. *A Formal Approach to Network Security Management*. PhD thesis, Princeton University, 2006 (in preparation).

[15] Sushil Jajodia, Steven Noel, and Brian O'Berry. Topological analysis of network attack vulnerabity. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challanges*, chapter 5. Kluwer Academic Publisher, 2003.

[16] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *The 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

[17] John Lambert, Matt Thomlinson, and Vishal Kumar. Microsoft Corporation, personal communication, July 2005.

[18] Steven B. Lipner. The trustworthy computing security development lifecycle. In *20th Annual Computer Security Applications Conference (ACSAC 2004)*, pages 2–13, December 2004. See also msdn.microsoft.com/security/sdl.

[19] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 245–254. ACM Press, 2002.

[20] Steven Noel, Sushil Jajodia, Brian O'Berry, and Michael Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *19th Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[21] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In *14th USENIX Security Symposium*, 2005.

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

[23] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

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

[25] Mark E. Russinovich and David A. Splomon. *Microsoft Windows Internals*. Microsoft Press, 2003.

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

[27] William R. Stanek. Windows 2000 server: Using default group accounts. http://www.microsoft.com/technet/prodtechnol/windows2000serv/evaluate/featfunc/07w2kadc.mspx. web page fetched January 28, 2006.

[28] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[29] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38. ACM Press, 2000.

[30] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In *Usenix LISA: 18th Large Installation System Administration Conference*, November 2004.

[31] Dan Zerkle and Karl Levitt. NetKuang–A multi-host configuration vulnerability checker. In *Proc. of the 6th USENIX Security Symposium*, pages 195–201, San Jose, California, 1996.