AFRL-IF-RS-TR-2003-249
Final Technical Report
October 2003

CORRELATED ATTACK MODELING (CAM)

SRI International

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J665

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-249 has been reviewed and is approved for publication.

APPROVED: /s/
GLEN E. BAHR
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
OCTOBER 2003

3. REPORT TYPE AND DATES COVERED
Final May 00 – May 03

4. TITLE AND SUBTITLE
CORRELATED ATTACK MODELING (CAM)

5. FUNDING NUMBERS
C - F30602-00-C-0098
PE - 62301E
PR - J665
TA - 10
WU - 01

6. AUTHOR(S)
Ulf Lindqvist, Steven Cheung, and Rico Valdez

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
System Design Laboratory
333 Ravenswood Avenue
Menlo Park California 94025-3493

8. PERFORMING ORGANIZATION REPORT NUMBER
P10851-010/020

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-249

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Glen E. Bahr/IFGB/(315) 330-3515/ Glen.Bahr@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*
This report describes research into the nature of multistep cyber attack scenarios, aimed toward automatic detection and identification of such attacks. Previous work in intrusion detection and cyber attack analysis has been focused mainly on isolated attack steps, and less on how such steps are combined into composite scenarios. The purpose of this work is to gain a deeper understanding of multistep attack scenarios, develop models of such scenarios, and design the mechanisms needed to automatically recognize such scenarios through event monitoring. Several multistep attack scenarios were developed and are documented in great detail in this report. Based on these scenarios, an attack modeling language called the Correlated Attack Modeling Language (CAML) was designed. A library of predicates and the concept of attack patterns were also developed to support the use of CAML. To validate CAML, a scenario recognition engine was developed. Furthermore, we designed an architecture supporting novel concepts in highly dynamic monitoring and correlation functionality, and a language for specification of active component behavior.

14. SUBJECT TERMS
Cyber Defense, Attack Modeling, Alert Correlation, Scenario Recognition, Attack Language, Intrusion Detection

15. NUMBER OF PAGES
124

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Contents

1 Summary **1**

2 Introduction **3**

3 Multistep Cyber Attack Scenarios **5**

3.1 Introduction 5
3.2 Attack 1 – Through the Web server 7
3.3 Attack 2 – Insider Attack 12
3.4 Attack 3 – Trojan Application via e-mail 15
3.5 Attack 4 – Attack through a User's Home PC 16
3.6 Attack 5a – Unix 19
3.7 Attack 5b – Unix 22
3.8 Attack 6 – Reconnaissance 22
3.9 Attack 7 – Covering Tracks 26
3.10 Other Considerations 26

4 Attack Modeling **29**

4.1 Introduction 29
4.2 Modeling Attacks 30
4.3 Attack Modeling Language 31
4.4 Attack Patterns 37
4.5 Implementing a Scenario Recognition Engine 40
4.6 Related Work 42

5 Adaptive Sensing and Correlation **45**

5.1 Introduction 45
5.2 Architecture 46
5.3 Alert Correlation and Scenario Recognition 51
5.4 Creation and Deployment of Custom Sensors 54

5.5 Related Work . . . 54

6 Concluding Remarks **57**

References **59**

A Attack Scenario References **63**

B CAML Grammar **67**
B.1 Introduction . . . 67
B.2 Grammar . . . 67

C CAML Predicates **97**
C.1 Introduction . . . 97
C.2 Services: Operating Systems and Applications . . . 97
C.3 Files . . . 101
C.4 Users . . . 103
C.5 Hosts . . . 105
C.6 Know . . . 106
C.7 Temporal . . . 107

D Correlation Engine Management Protocol **111**

Index **115**

Figures and Tables

Figures

3.1 A generic network architecture . . . 6
3.2 Attack tree . . . 28

4.1 CAML module: OpenSSL buffer overflow to remote execution . . 33
4.2 CAML module: Remote execution and access violation to data theft . . . 34
4.3 Attack model for the exfiltration scenario . . . 35
4.4 Attack pattern: Bandwidth Amplifier (Part I) . . . 38
4.5 Attack pattern: Bandwidth Amplifier (Part II) . . . 39

5.1 Architecture for adaptive sensing and correlation . . . 47
5.2 Basic operating model for correlation engine management and runtime updates . . . 49
5.3 Example SLAM trigger . . . 50
5.4 SLAM trigger: Activate-Backup-NIDS . . . 52
5.5 SLAM configuration modules: Host-based IDS on solar . . . 52
5.6 SLAM: Elevating monitoring level . . . 53

Tables

4.1 CAML predicate categories . . . 37

Acknowledgments

The authors are grateful to all the people at SRI International who have contributed to this project. Specifically, Martin Fong, John Khouri, Phillip Porras, and Alfonso Valdes have made significant contributions. The following people also contributed to the project during their time at SRI: Magnus Almgren, David Farrell, Sami Saydjari, and Bradley Wood. Finally, we thank the DARPA Cyber Panel Program Manager, Catherine McCollum, for her support and encouragement.

Chapter 1

Summary

SRI International has conducted research on how to model multistep cyber attack scenarios so that such attacks can be detected and identified by automated systems that analyze a stream of security alerts. Cyber security alerts are produced mainly by intrusion detection systems, but also by other sources such as firewalls, file integrity checkers, and availability monitors. A common characteristic for these first-level security alerts is that each isolated alert is based on the observation of activity that corresponds to a single attack step (exploit, probe, or other event). The process of "connecting the dots", that is, correlating alerts from different sensors regarding different events and recognizing complex multistage attack scenarios, has traditionally been manual and ad hoc in nature, and therefore slow and unreliable.

We have produced methods and a language for modeling multistep attack scenarios, based on typical isolated alerts about attack steps. The purpose is to enable the development of abstract attack models that can be shared among developer groups and used by different alert correlation engines. To verify that the language is suitable for describing attack models to a scenario recognition engine, a prototype of such an engine was developed, using components of the EMERALD intrusion detection framework that has been developed under this and other DARPA programs. The engine consumes security alerts and makes high-level conclusions based on the scenario models.

Under this project, research has also been conducted on how the attack modeling can be extended and used to control highly adaptive attack identification mechanisms. An architecture for adaptive security sensing and correlation has been designed, supporting novel concepts in dynamic monitoring such as "inquisitive sensors" that can actively probe targets to reduce uncertainty in observations, and custom sensors that can be dynamically deployed to cover a certain portion of the attack space. A language for representing models governing the adaptive behavior of the components has been developed.

This work shows how multistep cyber attack scenarios can be analyzed, modeled, and recognized in an adaptive cyber defense environment

Chapter 2

Introduction

For almost two decades, research in intrusion detection has mainly been focused on achieving accurate detection and identification of isolated malicious events. Significant effort has been expended in the characterization and encoding of the nearly atomic events that represent misuse. In contrast, only recently have researchers started to look at how attackers combine attack steps into scenarios, and how real-world attacks often consist of many such steps. Operators are often inundated with low-level alarms from their intrusion detection systems (IDSs), and would like to be informed about how these events are related and how they fit into a high-level attack scenario.

To enable automatic detection and identification of multistep cyber attack scenarios, we must be able to *understand* the characteristics of such scenarios and we must be able to *model* the scenarios for a recognition engine. It would also be desirable to represent the attack models in a generic way that could be translated to the particular representation that a given correlation technology uses. If models could be shared among different groups, those groups could concentrate on development of correlation algorithms rather than spending time encoding attack scenarios.

The work presented in this report addresses the problems described above, with the goal of providing developers of correlation technologies with a common way to express properties of multistep attack scenarios.

In Chapter 3, we present a number of detailed example attack scenarios set against a generic network architecture. In Chapter 4, we present a methodology and a language for modeling attack scenarios. Chapter 5 describes how our

work can be extended to support highly dynamic and adaptive cyber defense framework.

Chapter 3

Multistep Cyber Attack Scenarios

A set of multistep attack scenarios were developed to serve as examples and a basis for the formal scenario modeling. The scenarios are described here in the level of detail that is often necessary for accurate modeling.

3.1 Introduction

To support the Correlated Attack Modeling (CAM) Project, the SRI Red Team was tasked with developing and stepping through attacks against a fictitious network in order to give the developers of the modeling language some insight into what likely attacks would look like and how to best communicate characteristics of the attack. The decision was made to develop a "mission ignorant" attack model and apply it to specific scenarios to develop specific attacks.

3.1.1 Battlefield

A generic network architecture, shown in Figure 3.1, was designed to represent a typical network that could be used in a variety of different scenarios.

The network contains a DMZ where publicly accessible servers such as DNS, SMTP, and HTTP would be deployed. This firewall would be configured to allow traffic necessary for this communication. The firewall to the internal network is typically configured with a stronger rule set. Communication between servers on the internal network and servers in the DMZ would be necessary for some functionality such as dynamic Web content and relaying SMTP mail to the inside.

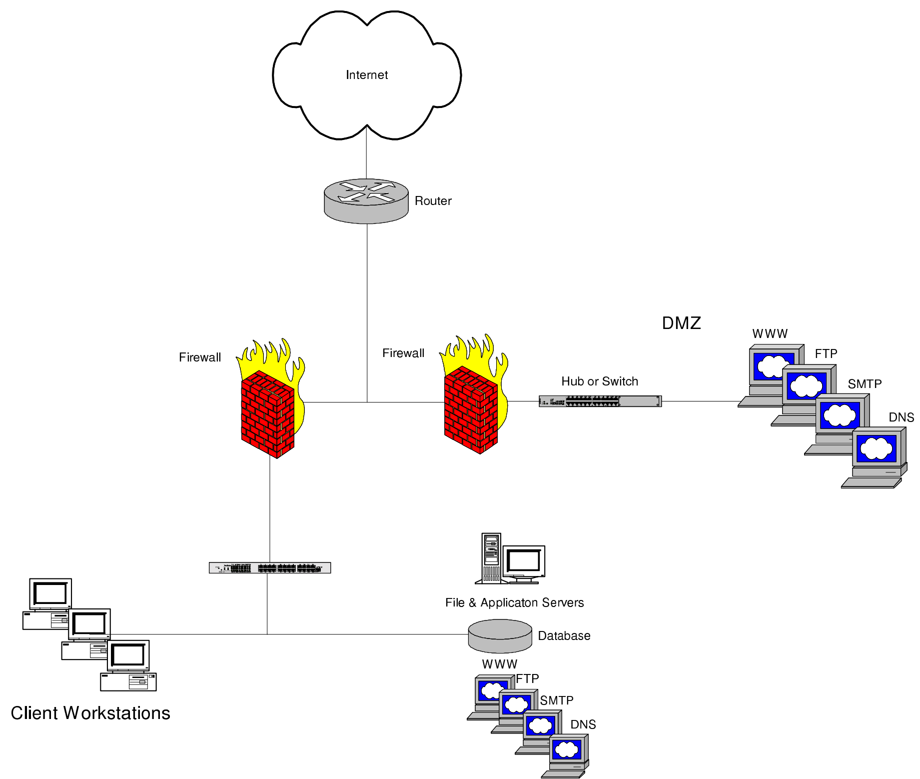


Figure 3.1: A generic network architecture

3.1.2 Scenario

The scenario was chosen to be that of an E-commerce network. In this scenario, a Web server exists in the DMZ that enables the members of the public to manage their accounts and place orders over an SSL-encrypted session. The Web server does not have content stored locally. The content resides on a file server in the internal network. The Web server also communicates with a database on the internal network for user account information, for placing orders, and so on.

3.1.3 Conventions Used

Steps typically map to target services or physical nodes. The substeps are what are required to advance through the attack to the next step. Throughout the following described attacks, bracketed [ ] steps indicate optional steps and curly braces { } indicate macro steps. A macro step is one in which the attacker has a choice of methods to complete the step.

3.2 Attack 1 – Through the Web server

3.2.1 Assumptions

- Web server platform is Windows 2000 with Service Pack 1 running IIS 5
- No content stored on Web server. Content is retrieved from internal file server through normal SMB communications. Account that accesses Web content only has read access to the content.
- Database is MS SQL 2000
- Firewalls configured to allow high ports (>1023) into network and all ports out.

3.2.2 Step 1

Goal: Obtain remote administrative access to the Web server and obtain all the usernames and passwords available on that machine.
A) The attacker tests Web site for IIS Unicode vulnerability (MS00-078) by attempting to exploit the bug to gain a directory listing. The following string input into a Web browser's address field will accomplish this.
`http://address.of.iis5.system/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir`

Prerequisites – Machine does not have the hotfix for this vulnerability installed. There exists a folder under the Web root with execute permissions assigned to anonymous Internet users (this exists by default). The anonymous user has execute permissions to cmd.exe and other operating system commands. A default Windows 2000 install with IIS5 and service pack 1 is vulnerable.

Result – Learns that external user can execute arbitrary commands under the security context of the account under which the Web service is running.

Observables – IIS Log files will show specifically malformed HTTP requests.

B) The attacker finds machine vulnerable. In order to redirect standard output by using the '`>`' character, the attacker must first copy cmd.exe to another file name. This can be done using the following command:
`http://address.of.iis5.system/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+copy+c:/winnt/system32/cmd.exe+cmd1.exe`

Prerequisites – The attacker has write and execute access to a directory.

Result – The attacker can now use the '`>`' character to redirect output from echo commands to create a file.

Observables – If the command in the example above were used, a copy of cmd.exe would now exist in the Inetpub\scripts directory with a name of cmd1.exe. IIS Log files will show specifically malformed HTTP requests.

C) The attacker uploads tools by creating a custom ASP file using echo commands via the bug. Echo is a shell command. An example of the command used would be
`http://address.of.iis5.system/scripts/cmd1.exe?/c+echo+This+is+text+to+include+in+a+file+>>+file.asp`

Prerequisites – The attacker needs to have access to a directory that allows anonymous Internet users to create files.

Result – The attacker has now created an ASP file.

Observables – IIS Log files will show malformed HTTP requests. A new ASP file will be created on the machine.

D) The attacker browses to the newly created ASP file and performs the upload(s): `http://address.of.iis5.system/scripts/file.asp`

Prerequisites – ASP file needs to reside in a directory under the Web root that is configured to allow ASP files to be executed.

Result – Files are uploaded. Files uploaded are shown in the following steps and referenced in Appendix A.

Observables – New files would be observed on server. IIS Logs will show HTTP POST requests.

E) The attacker runs PipeUpAdmin.exe to exploit vulnerability MS00-070.
`http://address.of.iis5.system/scripts/cmd1.exe?/c+PipeUpAdmin.exe`

Prerequisites – A service must exist that runs under a privileged context that the anonymous user account can start. The PipeUpAdmin exploit uses the clipbook service.

Result – The anonymous Internet user account now has administrator privileges, which gives the attacker the ability to run commands as administrator.

Observables – IIS Logs will show malformed HTTP requests, there will be a registry read on a specific key performed, a service will be started, the anonymous Internet user account will be in the administrators group.

F) {

Goal – Obtain interface of choice to run commands – note that future network-level observables would depend on choice of interface.

Option 1 – Full GUI

[VNC is installed and configured to listen on a high port (or other port the firewall allows through). See Appendix A for more information.

Prerequisite – Need to have administrator access to install.

Result – Service listening on a high port that allows VNC client to connect to target machine and get a full GUI. If a user is logged in at the console, the attacker obtains that user's screen. Otherwise, the attacker will be prompted with a standard Windows 2000 logon. The attacker will attempt to log on using the anonymous Internet user account (which has administrator rights from previous exploit). If machine is configured to disallow interactive logon to that account, the attacker may add a new administrative user with logon rights or assign the interactive logon right to the anonymous Internet user account. A new user can be added using native Windows commands (net user jdoe /add; net localgroup administrators jdoe /add). To assign the logon right, a Windows resource kit utility (or other utility) would need to be uploaded and used.

Observable – A new service would be created, VNC libraries (DLLs) and executables would be seen, the registry would be modified, and malformed HTTP requests would be seen. The use of VNC would be indicated by traffic on a high port, netstat showing a high listening port and a process appearing in the process list.]

Option 2 – Interactive shell

[Netcat could be used in conjunction with cmd.exe to provide an interactive shell to give a command line interface. An example command run on the machine would be

`http://address.of.iis5.system/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+nc.exe+-L+-e+cmd.exe+-p+1234`

Prerequisite – Need to have administrator access to configure to run with administrator privileges because it runs under the security context of the user who started it.

Result – Netcat listening on a high port configured to spawn a command shell upon connection from a telnet client (or another netcat).

Observable – A malformed HTTP request would be seen. The use of netcat would be indicated by traffic on a high port and netstat showing a high listening port. The netcat executable would be visible on the system and the process would be visible in a process list.]

Option 3 – Noninteractive shell

[Commands are issued using Unicode exploit as done earlier. Since shell is noninteractive, output of commands executed will be redirected to a file that can then be accessed via regular HTTP GET requests.

Prerequisites – Already met. Same method used until now.

Result – Continue to run commands via UNICODE exploit.

Observable – Malformed HTTP requests.]

}

G) Lsadump2.exe is executed in an attempt to extract services and associated passwords from the LSA.

Prerequisites – The attacker must have administrative privileges.

Results – Passwords for accounts used by the services running on the local machine are obtained. Lsadump2 will display the service name and clear text password.

Observables – System process queries the LSA and dumps contents.

H) Specific Registry keys (the service names) are queried to determine the usernames associated with services for which we found passwords above. This is done using the resource kit utility regdmp.exe. An example command to query the clipboard service is

```
regdmp HKEY_LOCAL_MACHINE\system\currentcontrolset\services\clipsrv
```

This would give the account name used for the service.

Prerequisites – The attacker must have an account on the system.

Results – The account names used by the services are obtained, which can now be matched with the passwords found above.

Observables – Registry read to specific keys will be observed.

I) pwdump2.exe is executed to extract usernames and hashed passwords from the SAM. Note that this step may not be necessary if domain administrator access was obtained from the previous step. We would take the output from pwdump2.exe and use it as input to L0phtcrack to obtain the cleartext passwords.

Prerequisites – The attacker must have administrative privileges.

Results – Local usernames and password are obtained.

Observables – System process queries the SAM and dumps information.

3.2.3 Step 2

Goal: Modify Web site by gaining write access to the actual Web content files and directly modifying them.

A) Users are enumerated on the file server and domain controller. This is done by using built-in Windows mechanisms that allow unauthenticated users to enumerate accounts, shares, and other information. In this case, winfo.exe will be used.

Prerequisites – Ports 139 or 445 allowed from the Web server in the DMZ to the inside network. Access to at least these ports at least on the file server would be necessary for the Web server to retrieve the Web content.

Results – Domain and local file server account names and group membership are now known.

Observables – A null session request and clear text usernames would be seen on the network originating from the Web server.

B) There exists a group called "webmasters" that we assume has read/write access through existing share. The attacker attempts to mount remote share with read/write privileges using normal commands (i.e., `net use x: \\fileserver\webshare`) using accounts in webmaster group with passwords found from running pwdump2 or lsadump2 above. The attacker finds a match.

Prerequisites – Ports 139 or 445 allowed from the Web server to the file server. This is necessary for the Web server to retrieve the Web content.

Results – The attacker now has read/write access to the Web content on the file server.

Observables – Failed logon attempts and mount requests would show up in the Windows event log if auditing had been properly enabled.

C) Web content is modified with editor of the attacker's choice. File could be modified directly or indirectly, or it could be replaced.

Prerequisites – The attacker has mounted share with read/write access.

Results – Web content is now modified.

Observables – Many different observables depending on method of modification. In all cases, a tool monitoring the Web content for modification would work.

3.2.4 Step 3

Goal: Retrieve credit card information stored in database by directly querying the database with a privileged account.

A) Connection is made to SQL server using osql.exe (which the attacker uploads). The SQL server is set up with a blank 'sa' password.

Prerequisites – TCP port 1433 (default) allowed from the Web server to the database. This would be required to incorporate database information in website. The MS SQL database has a null password for the 'sa' account.

Result – The attacker has full access to database.

Observables – Connection by 'sa' to database server from the Web server.

B) The attacker issues additional SQL commands to retrieve data. If database or table names are known, the attacker goes straight to them. If not, the attacker searches through available databases, tables, or fields using standard SQL queries until he finds credit card information.

Prerequisites – The attacker has made a connection to the database with an account that has read privileges to the desired information.

Result – The attacker now has credit card information.

Observables – Lots of cc information going into DMZ! General (as opposed to specific) SQL queries could be seen on network (i.e., 'select *' statements).

3.3 Attack 2 – Insider Attack

3.3.1 Assumptions

- The attacker has normal, unprivileged user account.
- Insider has knowledge (or can obtain information) of which machine(s) contain desired information and information-providing services.
- User does not have administrator access to his own machine.
- Network management, anti-virus or backup service running under domain administrator account on local workstations
- MS SQL2K uses Windows Authentication.

3.3.2 Step 1

Goal: Get domain administrator privileges by retrieving usernames and passwords of service accounts running on the local workstation.
A) The attacker obtains administrative privileges to his local workstation by exploiting vulnerability MS00-070 using PipeUpAdmin.

Prerequisites – The attacker has a local unprivileged account on the workstation. A service must exist that runs under a privileged context that the anonymous user account can start. The PipeUpAdmin exploit uses the clipbook service.

Result – The unprivileged user account now has administrator privileges, which gives the attacker the ability to run commands as administrator.

Observables – There will be a registry read on a specific key performed, a service will be started, and the user account will be in the administrators group.

B) The attacker runs lsadump2 and finds passwords associated with the accounts under which the services on the local machine are running.

Prerequisites – The attacker must have administrative privileges.

Results – Passwords for accounts used by the services running on the local machine are obtained. Lsadump2 will display the service name and clear text password.

Observables – System process queries the LSA and dumps contents.

C) Specific Registry keys (the service names) are queried to determine the usernames associated with services for which we found passwords above. This is done using the resource kit utility regdmp.exe. An example command to query the clipboard service is

```
regdmp HKEY_LOCAL_MACHINE\system\currentcontrolset\services\clipsrv
```

This would give the account name used for the service.

Prerequisites – The attacker must have an account on the system.

Results – The account names used by the services are obtained, which can now be matched with the passwords found above. In this case, we obtain domain administrator privileges.

Observables – Registry read to specific keys will be observed.

3.3.3 Step 2

Goal: Modify Web site by gaining write access to the actual Web content files and directly modifying them.

A) With the domain administrator privileges, the attacker mounts the default drive share on file server, which gives the attacker full access to the entire disk partition. This is done using normal commands (i.e., `net use x: \\fileserver\c$`). At this point the attacker can search the partition for the Web content.

Prerequisites – Ports 139 or 445 must be allowed. The attacker must have a username/password that has read/write permissions to the share.

Results – The attacker now has read/write access to content and is able to modify and create files as well.

Observables – Mount request and success acknowledgment could be seen on the network.

B) The attacker analyzes the Web content files that query the database which are used in the normal operation of the Web server.

Prerequisites – The attacker has at least read access to Web content.

Results – The attacker gathers database information (e.g., database names, table names, possibly usernames/passwords).

Observables – If auditing was properly enabled, the event log would show anomalous accesses to these files.

C) Web content is modified with editor of the attacker's choice. Files could be modified directly or indirectly, or could be replaced.

Prerequisites – The attacker has mounted share with read/write access.

Results – Web content is now modified.

Observables – Many different observables, depending on method of modification. In all cases, a tool monitoring the Web content files for modification would work.

3.3.4 Step 3

Goal: Get customer credit card information by identifying accounts with access to the data and then getting and cracking the password hashes for those accounts found on the domain controller. The database is then queried directly.

A) The attacker will enumerate the users and groups on the database server to identify accounts that have access to the data in which we are interested. In this case, winfo.exe will be used.

Prerequisites – Ports 139 or 445 allowed from the Web server in the DMZ to the inside network. Access to these ports at least on the file server would be necessary for the Web server to retrieve the Web content.

Results – Domain and local file server account names and group membership are now known.

Observables – A null session request and clear text usernames would be seen on the network originating from the Web server.

B) The attacker will locate a PDC or BDC by identifying the machine by which he was authenticated. Viewing the environment variables using the 'set' command accomplishes this.

Prerequisites – User is logged on to workstation and has been authenticated by the domain.

Results – The attacker now has identified a domain controller.

Observables – System call to read the environment variable that contains the name of the domain controller that authenticated the user.

C) The attacker extracts usernames and password hashes from the PDC using L0phtcrack and cracks the passwords of accounts that were identified to have access to the database.

Prerequisites – The attacker has domain administrator privileges and uses them to make an IPC connection to the domain controller.

Results – The attacker obtains the passwords for all accounts that have access to the database.

Observables – An administrative IPC session and clear text password hashes could be seen on the network.

D) At this point the attacker can connect to the database using osql.exe or other SQL management tools. The attacker may need to attempt connection with multiple accounts until one is found that has access to the desired data (credit card information).

Prerequisites – User can connect to the database on TCP port 1433. Usernames and accounts for the database are known.

Results – The attacker gains access to credit card information

Observables – Failed logon attempts to the database would be seen unless the attacker is able to accurately determine an account with the appropriate privileges on the first authentication attempt.

E) The attacker queries the proper tables in the proper database for the credit card information.

Prerequisites – The attacker knows database and table names where credit card information is stored. The attacker has username/password for an account that has at least read access to the credit card data.

Results – The attacker now has credit card information.

Observables – Credit card information would be seen traveling in clear text across the network.

3.4 Attack 3 – Trojan Application via e-mail

3.4.1 Assumptions

- Non-stateful firewall in use.
- Firewall allows internal users to FTP/HTTP out.

3.4.2 Step 1

Goal: Transition from an outsider to an insider by using a remote shell through a firewall. The backdoor is provided by a Trojan application sent in an e-mail.

A) The attacker makes a Trojan horse of a small popular game, by using an application called *elitewrap*. Elitewrap wraps the game and another program called Ackcmd into a binary package and runs them both when executed. The game plays as usual while the Ackcmd binary is run without the user's knowledge. Ackcmd provides a remote command execution using a TCP ACK-tunnel, which enables it to bypass some firewalls. Ackcmd also continues to run after the Trojan application is killed. This Trojan is then e-mailed to an employee.

Prerequisites – E-mail gateways do not filter executable attachments or scan for known Trojans. End user must double click the file attachment to execute. End user must not be running a virus scanner, as some virus scanners will detect this as a Trojan application.

Results – Port 1054 on the victim's machine will be listening for an ack packet. Note that a netstat would not show a listening port because a full TCP connection is never established.

Observables – Unencrypted SMTP traffic carrying an executable attachment could be observed going to the mail server. An executable attachment could be seen at the mail server. The e-mail could also be seen on the network being delivered to the victim's machine. Once executed, the Ackcmd process could be seen in the process list. Note that this could be renamed to look like a system process.

B) Once this Trojan is in place, an attacker can connect to the machine through the firewall using the Ackcmd client component.

Prerequisites – The machine running the Ackcmd server component must have a publicly accessible IP address. As stated in the assumptions above, the firewall must not be "stateful", meaning it does not keep track of TCP handshaking and allows packets that look to be part of an established connection.

Results – The attacker now has the ability to remotely execute commands with the privileges of the user who executed the Ackcmd server component.

Observables – The Ackcmd server component on the victim machine communicates over a high port (1054) with the Ackcmd client component on the attacker's machine on port 80. At first glance, it appears to be a normal HTTP session; however, system commands and responses can be seen in these packets.

C) The attacker can FTP out to get other tools.

Prerequisites – Internal users are allowed to FTP out.

Results – The attacker now has tools and exploits available locally.

Observables – Client/server Ackcmd traffic can be seen as stated above as well as FTP traffic from the victim machine. New files can be seen on the local machine in whatever directory the attacker chooses to put them.

At this point, the attacker has made the transition from outsider to insider in regard to access, and continues along the path in *Attack 2* steps 1, 2 and 3.

3.5 Attack 4 – Attack through a User's Home PC

3.5.1 Assumptions

- Network administrator's home PC is Windows 98.
- Administrator's home machine has file sharing enabled with the system drive shared.
- Administrator has access to corporate network through a VPN connection.
- Once connectivity is established with the corporation network, the administrator has Domain Administrator privileges due to use of Microsoft RRAS for VPN.
- Web application is based on Microsoft Active Server Pages technology.

3.5.2 Step 1

Goal: Identify an administrator's home PC and attack that PC to obtain VPN configuration and authentication information.

A) The attacker identifies the administrator's home PC connected to the Internet via a cable modem. This can be by sniffing packets on the cable modem segment, using Windows NETBIOS browsing mechanisms (Network Neighborhood), or through noncyber means.

Prerequisites – The attacker has access to the same cable segment as the administrator.

Results – The attacker now has an IP address of the target machine.

Observables – Because this is a passive activity, it is difficult for the attacker to be observed. However, it may be possible to use active sensor technology to detect network interface cards in promiscuous mode.

B) The attacker enumerates the shares on the PC using the built-in Windows "net" command with the "view" argument.

Prerequisites – The attacker has IP address of target, the drive is shared via Windows 98 file and print sharing, and there is no firewall or other device preventing access to TCP port 139.

Results – The attacker has a list of available shares on the machine.

Observables – SMB connections to port 139 on the victim machine could be observed.

C) The attacker mounts shares on the employee PC.

Prerequisites – The attacker has IP address of target, the drive is shared via Windows 98 file and print sharing, and there is no firewall or other device preventing access to TCP port 139.

Results – The attacker has full read/write access to the hard drive through a share of the entire drive.

Observables – Network activity would indicate an SMB connection from a remote machine.

D) After mounting the drive, the attacker first determines what VPN software is being used and the configuration. This is accomplished by directory listings to determine what software is installed and if software is found, copying over the text files with the proper information. If no third-party VPN software is discovered, the attacker assumes the administrator is using the built-in Windows VPN client. To verify, and get configuration information, the attacker copies over the registry files and analyzes them offline.

Prerequisites – The attacker has read/write access to file system.

Results – The attacker now knows which VPN software is being used and parameters required to connect.

Observables – Disk activity on the victim machine and SMB traffic to port 139 on the victim machine.

E) The username and password to connect to the corporate resources must be found. This could be done by a text search of files on the machine for the keyword "pass" and finding a file with the authentication information. If usernames/passwords are not found clear text on the machine, a keyboard logger is copied over and put in the Windows startup folder, which starts the keyboard logger the next time the system is booted. The keystrokes are then copied to a file, which the attacker returns to get at a later time.

Prerequisites – The attacker has read/write access to system drive.

Results – Username and password for VPN access into corporate network is discovered.

Observables – On the victim machine, if a keyboard logger is used, the process could be seen as a running process in the task list. Also, the file with the keystrokes that the program generates would be visible as well as the program in the startup folder. File transfers via SMB would also be observable on the network.

3.5.3 Step 2

Goal: Configure local machine to mimic behavior of administrator's home PC and establish VPN tunnel with the organization.
A) The attacker installs client VPN software if needed on his local machine and connects using the username/password for the corporation network found in Step 1 above. This gives the attacker an "in" to the network behind the firewall and domain administrator privileges as stated in the assumptions.

Prerequisites – The attacker has username and password and knowledge of how to connect to the corporation network via VPN.

Results – The attacker is connected to the corporate network and is able to operate as though he were a node on the corporate network. He has also been authenticated on the domain with domain administrator privileges.

Observables – The VPN connection would show up in the appropriate logs. This information may not be anomalous, though, if VPN connections are allowed from anywhere. If domain logons were being audited, the logon would show up here as well.

3.5.4 Step 3

Goal: Determine location of Web content and modify.
A) The attacker uses standard IIS Administration tool to examine the configuration of the IIS Web server and determine where the Web content is stored. The IP address of the machine can be retrieved via standard DNS requests.

Prerequisites – The attacker must have privileges to read the configuration of the IIS server and access to port 139 or 445.

Results – The attacker now knows on which computer and share the content is stored.

Observables – An SMB connection could be seen on the Web server. A network logon would be performed on the Web server and could be seen if auditing were configured to do so.

B) The attacker then mounts the filesystem with the Web content and determines the database name and tables used by the Web application. This information is stored in the files that make up the Web application. The attacker can easily locate information by doing a text search for a database-specific keyword such as 'DSN='.

Prerequisites – The attacker must have at least read access to the share that contains the Web content.

Results – The attacker retrieves database names and address as well as table names and other useful information.

Observables – SMB traffic to and from the Web server and disk activity on the Web server.

C) At this point, assuming the attacker has read/write access to the Web content, the content could be modified using the attacker's editor of choice. If

the attacker does not have these privileges, he uses his domain administrator privileges to mount the whole drive, which gives him the necessary access.

Prerequisites – The attacker has read/write access to the file system with the Web content.

Results – Web content has been modified.

Observables – File(s) would be modified.

3.5.5 Step 4

Goal: Get the credit card information from the database.
A) The attacker connects to SQL server identified in Step 3 using the MSSQL administration tool with current domain administrator user/pass.

Prerequisites – SQL server uses standard Windows 2000 logons for security, the administrator has the privileges necessary to connect to the SQL server, and port 1433 is reachable from his workstation.

Results – The attacker has a connection to the server that holds the database that stores credit card information.

Observables – IP traffic to TCP port 1433.

B) At this point, the attacker checks the permissions for the database that holds the credit card information, using the administration tool. If it is determined that the attacker does not have the proper privileges, he checks to see which groups have the correct privileges and makes his account a member of one of those groups.

Prerequisites – The attacker has necessary privileges to read the credit card information contained in the database.

Results – The attacker has obtained credit card information.

Observables – Credit card information passing unencrypted on wire.

3.6 Attack 5a – Unix

In these attacks, the scenario is set up in the same way as in previous attacks except that the hardware and software platforms are different. This attack can be used against any Unix platform running the vulnerable services. The exploits used may need to be modified based on the target Unix OS and hardware platforms. The attacker begins the campaign from somewhere out on the Internet.

3.6.1 Assumptions

- Apache 1.3.19 is used.
- Same root password used on DNS server and Web server.
- Bind 8.2.1 running on DNS server in DMZ.

- OpenSSH 2.2.0 running on Web and DNS servers. Root allowed to SSH in.
- MySQL running as the database to hold account information.
- Web content accessed via NFS from machine on internal network.

3.6.2 Step 1

Goal: Obtain a root shell on the Web server.
A) The attacker uses Bind exploit to get a root shell on DNS server in the DMZ.

Prerequisites – Victim is running a vulnerable version of BIND running SUID root.

Results – The attacker has an interactive remote root shell.

Observables – Exploit code could be seen on the network. DNS server could be seen making an outgoing connection to the attacker machine on port 2500 (default for the exploit). Invocation of root shell could be seen by a host IDS.

B) The attacker FTPs shadow file to himself and cracks root password offline using crack, john, or a similar tool.

Prerequisites – The attacker has root access on the server. FTP connections through the firewall from the DNS server are allowed. The attacker has a cracking tool to brute-force the encrypted passwords to find their clear-text equivalent.

Results – The attacker now has the username and password for the root account on the compromised host.

Observables – The shadow file could be detected being transferred to the Internet; a file transfer session originating from the DNS server could also be seen. If the DNS server were logging file transfers, this activity could show up in the log as well. Critical file monitoring, if in use, would detect access to the shadow file.

C) Web server is configured to use the same root password as the DNS server. The attacker SSH's into the Web server and examines Apache configuration files to determine the location of the Web content.

Prerequisites – The attacker has the password for the root account on the Web server. The attacker also has knowledge of Apache and the format of the config files. The Web server is configured to allow root to SSH or telnet in.

Results – The attacker has interactive shell with root privileges on the Web server.

Observables – An SSH connection could be seen on the network from the DNS server to the Web server and on the Web server itself. The logon to the Web server would be logged appropriately.

3.6.3 Step 2

Goal: Replace Web content with content of the attacker's choice by relocating the Web root.

A) The attacker at this point analyzes Web content to determine database and table names and find usernames and passwords for the database.

Prerequisites – The attacker has at least read access to the Web content. Note that this would be a necessary requirement for the Web server to be able to serve up the content.

Results – The attacker gains information enabling remote access to database (i.e., privileged login information, database layout to allow targeting of specific tables). The attacker has ability to log into database and retrieve any information.

Observables – Since the Web server would already have a connection established to the Web content, no new connections would be observed. However, patterns of file access on the Web server would be different from the "well understood" profile of file access from the Web server.

B) The attacker uses FTP to retrieve new Web content from an FTP server on the Internet. Next, the attacker would update the configuration files on the Web server so that the Web content now comes from the local machine. The attacker must then restart the Apache daemon.

Prerequisites – The attacker has privileges necessary to modify the Apache configuration files on the Web server and restart the Web daemon. The firewall must also allow outgoing FTP connections from the Web server.

Results – New Web content is now served from new local Web root directory.

Observables – If changes to configuration files were being watched for, this activity could be detected here. The FTP session to retrieve the new content could be detected. The restart of Apache would probably show up in a log, and there would be new files on the Web server.

3.6.4 Step 3

Goal: Obtain credit card information by creating a new Web page that queries the database and returns credit card information, and then point Web browser to it.

A) The attacker creates an HTML file that queries the database and returns the credit card information. This can be done locally using any text editor, or this can be created offline and transferred over with the rest of the Web content.

Prerequisites – The attacker knows the format to use to create a Web page that queries the database and returns the credit card information. This knowledge is mostly gained from analyzing the Web content in Step 2 above.

Results – The attacker has a Web page that returns the credit card information when he points his Web browser to it.

Observables – The query for the credit card information may be unusual and could be detected. The fact that all credit card information is queried, as op-

posed to information on one account, would probably be considered anomalous. A successful HTTP request for an unknown Web page could be seen.

3.7 Attack 5b – Unix

3.7.1 Assumptions:

- OpenSSH 2.2.0 running on Web server. Root allowed to SSH in.

3.7.2 Step 1

Goal: Obtain a root shell on the Web server.
A) The attacker exploits SSH on Web server to connect with the root account. This can be done directly without having to go through another machine. Note that steps 1b and 2b occur on the same physical node (host).

Prerequisites – Web server is running a vulnerable version of SSH and allows the root account access via SSH. The attacker has exploit modified if necessary.

Results – The attacker has a root shell on the Web server.

Observables – Exploit code could be seen in network traffic to the Web server. Log files would show anomalies in the SSH operation.

3.7.3 Step 2

At this point we continue along the same attack path as Attack 5a, beginning at Step 2.

3.8 Attack 6 – Reconnaissance

This attack or components of it can be overlaid or used as a precursor to any of the attack steps or substeps.

3.8.1 Noncyber Reconnaissance

Several methods used in noncyber reconnaissance would be very difficult to detect. An attacker could obtain insider information from an administrator or other person knowledgeable about the IT infrastructure of the target organization. Social engineering tactics can also be used whereby an attacker could call employees, pretend to be a help desk person or other IT employee, and then attempt to dupe the user into giving out privileged information. "Dumpster diving" is another common tactic where a determined attacker will rummage through the garbage of an organization, looking for any bit of information that will give him an edge. There are many more variations on these themes.

3.8.2 Publicly Available Information

Assuming an attacker has little to no prior knowledge of the target organization, the attacker would most likely scour available public databases to learn as much about the target organization as possible. IP addresses owned by the target corporation can be found by searching through the ARIN database. Names of administrators, any other domain names registered to a company or individual, and IP addresses of machines responsible for DNS information can typically be found in WHOIS databases. An attacker may also look for news articles and search databases such as EDGAR for information about newly acquired companies. These companies may prove to be an easier target to penetrate and may have hastily connected to the parent organization with poorly planned access controls. An attacker may also search technical newsgroups, looking for postings from the domain of the company to see if an administrator asked any questions that might reveal information about policies or about hardware and software in use at the organization. The Web presence of the organization will also be scoured to search for information that could be of use. The source code for the Web pages, for example, may have notes listed or IP address and database names given in clear text. Names or organizational information found on Web pages can often help to facilitate a social engineering attack.

Prerequisites – The attacker has knowledge of available sources of information and Internet access.

Results – The attacker will have more information than before about the target. This information could be helpful in any number of ways.

Observables – There would be no cyber observables, as this activity occurs almost entirely outside of the target network. The only information gathering on the target is via normal allowed Web browsing.

3.8.3 Active Reconnaissance on Organization

The attacker probably by now has a large range of IP addresses registered to the target organization. The next step will be to identify those addresses that are accessible or high-value systems. The DNS servers for the domain names identified in step 1 would be queried to determine IP addresses for mail and other servers that may provide network services to the public (e.g., FTP, WWW, News). A zone transfer may also be attempted. If the zone transfer is successful, the attacker will have obtained all the DNS records the server has for that domain. Depending on the design, these could be IP addresses for internal machines as well and may identify databases, network management stations, certificate servers, or other high-value targets.

Prerequisites – DNS servers for the domain in question are publicly accessible. This is necessary for domains that host Web sites or exchange Internet e-mail. For a zone transfer, the DNS server must be configured to allow this from unknown addresses.

Results – Servers that host network services can be identified.

Observables – If an attacker attempts a zone transfer, this will be logged.

Multiple queries for nonexistent servers from the same IP address may raise suspicion.

This is an excellent time to attempt to discover as much as possible about the network topology. Traceroutes can be used to identify routers and other network devices. Firewalk or a similar tool could be run to attempt to map the firewall rules.

Prerequisites – The IP address of the firewall or other packet-filtering device must be known, as well as the IP address of a host behind it to run Firewalk.

Results – The attacker has knowledge of the rules on the firewall and can direct attacks or possibly pass traffic to the internal network through those ports. An attacker could also identify the path packets take to reach their destination and can identify network perimeters and routers.

Observables – An unusually large number of TTL expired ICMP messages would be visible in the network traffic. Firewalk may have other signatures that could enable an intrusion detection system (IDS) to detect it.

An attacker's next step will probably be to verify which IP addresses can be reached from the Internet and, optionally, what TCP or UDP ports are listening on those machines. Nmap is a favorite tool for these types of scans, but there are many others.

Prerequisites – The attacker has access to the Internet and appropriate tools. Knowledge of firewall rules could enable more successful scans.

Results – Machines and the services running on those machines that are publicly accessible are identified.

Observables – Many different types of observables are possible, depending on the scanning method used by the attacker. Most likely, packets could be detected with a destination port that is not open on the machine toward which the packet is directed.

If not done in conjunction with the scanning phase, operating systems will be identified next. Nmap is again the favorite tool for this activity; however, Queso and other tools can perform the same function.

Prerequisites – The attacker must be able to reach a machine via the Internet and receive replies from it.

Results – The attacker has a very good guess of the operating system running on the machine.

Observables – IP packets with unique options set could be detected in the network traffic. Most of these tools have a unique signature associated with them.

3.8.4 Finding Vulnerabilities

Once machines have been enumerated and services have been identified, an attacker will most likely look further at each identified service to determine the vendor and version of each. To accomplish this, an attacker will connect to a listening port to see what information is sent back. In some cases, the attacker may need to attempt connection with a client tool (such as a pcAnywhere client) to identify the service and version running. Note that the attacker could create

a custom tool to send special queries to the listening port. The content of the query would depend on the service the attacker expects to find running there. The attacker will then use this information to determine which machines and services are likely to have vulnerabilities.

Prerequisites – The attacker has access to Internet and appropriate tools.

Results – The attacker may be able to determine the software and version used to run the services on the target machine.

Observables – A full TCP connection to listening ports could be detected. The data sent to the port may not be what was expected by the listening application, so this could be used to trigger a flag.

The attacker may also run noisy or blatantly overt vulnerability-scanning tools such as Cybercop or ISS Internet Scanner (commercial tools), or Nessus (open-source). These are general tools that will look for a variety of vulnerabilities on a machine. These tools are usually easily detected and have unique signatures.

Prerequisites – The attacker has access to the Internet and appropriate tools.

Results – The attacker has information on what vulnerabilities are likely to exist on the target machine with little effort put forth by him.

Observables – Again, the observables would be dependent on how the attacker customized the scanning software, but there would be telltale signs if any of these automated tools were run. In almost every case, there would be a flurry of activity that most current IDSs would detect and correctly identify.

If a Web server is running, the attacker may also run scanners that look for vulnerable CGI scripts or other well-known Web server or Web content vulnerabilities. Whisker is a popular tool to use for this type of scanning. Other application-specific scanning tools may be used. These tools are much more specific and less noisy than the ones just previously described, mainly because they target a specific application and do not try to look for vulnerabilities in services that do not exist. However, they are still fairly easy to detect if someone is looking.

Prerequisites – The attacker has access to the Internet and appropriate tools.

Results – The attacker has information on what vulnerabilities are likely to exist in a specific application on the target machine with little effort put forth by him.

Observables – Observables would be dependent on the software used and how the software works. In most cases, though, there will be a flurry of network activity to a specific port, and logs for the application being scanned will show a variety of random activity. Many IDSs will detect signatures that these scanners will generate.

3.8.5 Iterative Reconnaissance

If an attacker has penetrated the perimeter and has root or equivalent access on a machine on the internal network, tools will usually be uploaded to allow further exploration. For example, a sniffer will typically be installed to glean passwords from the wire and analyze data flow to determine important servers.

Vulnerability scanners may now be run on more machines and may uncover more vulnerabilities than previously identified outside the perimeter. Typically, there is much unencrypted traffic inside the network perimeter and policy is more relaxed. Each time the attacker gains more access, scanning, topology mapping and other reconnaissance techniques may be employed again to gain more information.

3.9 Attack 7 – Covering Tracks

This attack or components of it can be overlaid or used as a postactivity in conjunction with any of the attack steps or substeps.

Once an attacker has some level of access to a machine, there are many ways for him to cover his tracks. The higher the privilege the attacker has gained, the more he can do to obscure or eliminate records of his activity. For example, an attacker may

- Hide processes by renaming (or otherwise)
- Modify and/or delete logfiles
- Use encryption to obscure data
- Use covert methods to thwart network monitoring
- Launch attacks from machines difficult to trace back to attacker
- Hide files and directories
- Disable auditing
- Hide functionality (i.e., Trojans, modified system files)
- Spoof identification
- And many others. . . .

3.10 Other Considerations

It is important to note that many options can be taken along an attack path and the option taken is a function of the attacker's skill, aversion to detection, style, and perception of the defenses in place. An attack diagram can be useful to see where a decision in the attack path can be made and where those decisions can lead. Figure 3.2 on page 28 depicts an attack tree for all the attack scenarios described in this chapter, with attack steps, options, and goals.

The "reconnaissance" and "covering tracks" attacks can be used in conjunction with any of the other attacks. The degree to which the information gathering would be used is dependent on the attacker and the amount of information already available to him. The degree to which an attacker would be

concerned about covering his tracks is dependent upon the attacker's aversion to detection and level of sophistication. Methods of covering tracks can also be used in the reconnaissance phase.

28

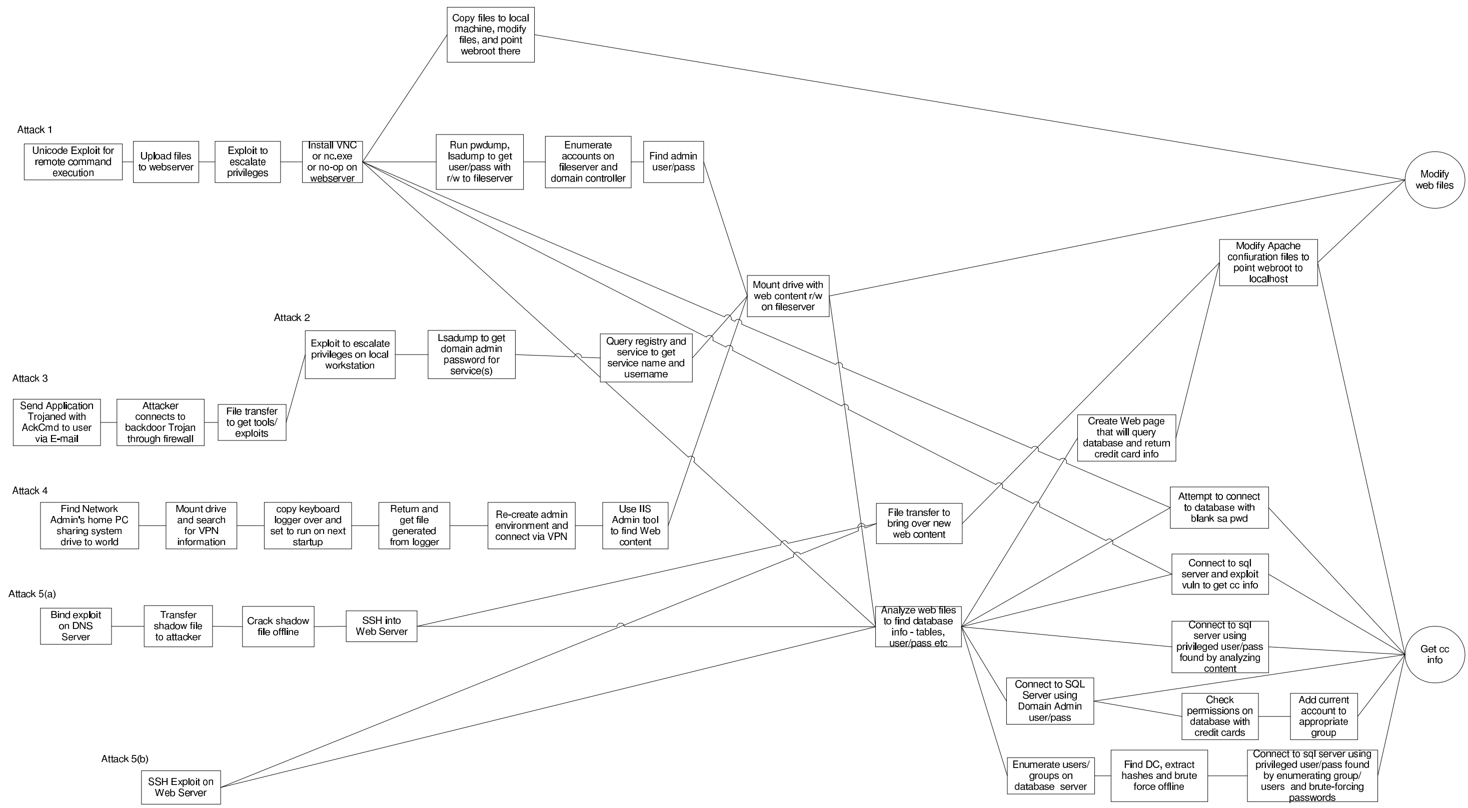


Figure 3.2: Attack tree

Chapter 4

Attack Modeling

SRI International has developed a methodology and a language called the Correlated Attack Modeling Language (CAML) for modeling of multistep cyber attack scenarios [9]. CAML uses a modular approach, where a module represents an inference step and modules can be linked together to detect multistep scenarios. CAML is accompanied by a library of predicates, which functions as a vocabulary to describe the properties of system states and events. The concept of attack patterns is introduced to facilitate reuse of generic modules in the attack modeling process. CAML is used in a prototype implementation of a scenario recognition engine that consumes first-level security alerts in real time and produces reports that identify multistep attack scenarios discovered in the alert stream.

4.1 Introduction

Security alerts are produced mainly by intrusion detection sensors, but also by other sources such as firewalls, file integrity checkers, and availability monitors. A common characteristic for these first-level security alerts is that each isolated alert is based on the observation of activity that corresponds to a single attack step (exploit, probe, or other event). The process of "connecting the dots", that is, correlating alerts from different sensors regarding different events and recognizing complex multistage attack scenarios, is typically manual and ad hoc in nature, and therefore slow and unreliable.

It would be highly desirable to automate the attack scenario recognition process, but there are several challenges facing such efforts:

- Knowledge representing attack scenarios needs to be modeled, preferably in a way that is decoupled from the specifics of a particular correlation technology.
- Producers of first-level security alerts are heterogeneous, and the alert content may vary.

- Attacks belonging to the same scenario could be spatially and temporally distributed.

- First-level alerts could be produced in very high numbers as a result of false positives, repeated probes, or as an attacker-induced smokescreen.

- An attack scenario could be executed in different ways that are equivalent with respect to the attackers' goal. For example, the temporal ordering of some attacks could be changed, or one attack could be substituted for a functionally equivalent one.

- Some attacks that constitute part of a scenario will not be represented in the alert stream. This could be due to missing sensor coverage or because the attack—albeit part of an attack scenario—is indistinguishable from normal benign activity.

We have developed methods and a language for modeling multistep attack scenarios, based on typical isolated alerts about attack steps. The purpose is to enable the development of abstract attack models that can be shared among developer groups and used by different alert correlation engines. To verify that the language is suitable for describing attack models to a scenario recognition engine, we have developed such an engine that consumes low-level alerts and makes high-level conclusions based on the scenario models.

4.2 Modeling Attacks

Our discussion of attacks and attack steps is guided by the following key definitions:

Vulnerability A condition in a system, or in the procedures affecting the operation of the system, that makes it possible to perform an operation that violates the explicit or implicit security (or survivability) policy of the system

Exploit Single-step (atomic) exploitation of a single vulnerability

Attack step An exploit or other activity performed by an adversary as part of a campaign toward the adversary's goal

(Composite) Attack A collection of one or several attack steps

Attack models for scenario recognition are related to attack trees/graphs used by red teams. However, the purpose of the attack models is not to provide details on how each attack is to be carried out. Instead, the emphasis is on how the attacks are detected and reported. Our modeling methodology includes the following tasks:

- Identify logical attacks in an attack scenario: These attacks may correspond to attack subgoals, and each of them may be further decomposed until it can be detected by a sensor.

- Characterize these logical attacks from the detection point of view: These attacks may be detected by observing certain events, observing certain system states, or performing inferences.

- Specify relationships among these attacks: In particular, there are temporal relationships (e.g., one attack happens before another one), attribute-value relationships (e.g., the target of one attack is the same as the source of another one), and prerequisite relationships (e.g., one attack enables another one to occur).

An attack modeling language must be able to express the knowledge compiled in the modeling tasks described above. In addition, an attack modeling language should fulfill the following requirements to efficiently support attack scenario recognition:

- *Extensible* to handle new attacks and sensors

- *Expressive* to cover the range of attacks in which we are interested

- *Unambiguous* to enable mechanization

- Enabling *event reduction* to identify a high-level security event from a large number of low-level incident reports

- Enabling *efficient* implementations

- *Independent* of sensor technologies other than assuming that sensors and correlators use a standard means to communicate

4.3 Attack Modeling Language

The Correlated Attack Modeling Language (CAML) enables one to specify multistage attack scenarios in a modular fashion. A CAML specification contains a set of *modules*, which specify an inference step. Moreover, the relationships among modules are specified through pre- and post-conditions.

Let us consider the following multistep attack scenario as an example: An attacker first exploits a buffer overflow vulnerability of a secure socket layer (SSL) implementation, on which a Web server depends, to obtain remote execution capability. From the Web server, the attacker mounts a file system to access some sensitive data. Then a file corresponding to a Web page is modified to include this data, which the attacker can download using an HTTP request.

The individual attacks of this attack scenario may be observed by different sensors. For instance, a signature-based network IDS may detect the buffer overflow attack step, an anomaly detection component may detect the unusual file access, and a file integrity checker may detect the modification of the Web page. To recognize the “exfiltration” attack scenario, one needs to correlate the pieces.

This scenario has many different variations. For example, instead of using an attack that exploits the SSL vulnerability, the Web server may have other vulnerabilities (e.g., a buffer overflow vulnerability in Windows IIS indexing service [7]) that, when exploited, would enable the remote attacker to run arbitrary code on it. Instead of accessing a file, the attacker may perform network sniffing or query an SQL server to steal data. The large number of different combinations of the attacks makes it difficult to recognize multistage attack scenarios manually. Moreover, explicitly enumerating all these combinations makes attack models less extensible. When a new attack is known, one may have to revisit and modify many previously defined multistage attack models to incorporate it.

4.3.1 CAML Examples

Figure 4.1 shows a CAML module for the SSL buffer overflow attack step. In this example, one may first notice the similarity between the structures of some CAML constructs and the Intrusion Detection Message Exchange Format (IDMEF) [11]. This design facilitates CAML to interoperate with different types of sensors that can generate reports in IDMEF. See Appendix B for the CAML grammar.

In Figure 4.1, the activity section (cf. Lines 2–11) specifies an event template that could match with intrusion detection reports (which correspond to event instances) for the buffer overflow attack. When a match occurs, the variables in the template (s, t, and tp) are instantiated with the corresponding values in the event instance. For example, s will be instantiated with the source IP address reported by a sensor. Moreover, if a literal is used in the template (e.g., "CAN-2002-0656" as classification name), an event instance must have that value in the corresponding field to match the template.

The pre-condition section (cf. Lines 12–29) specifies the set of conditions that must be met by the event instance and the system state to trigger the inference of this module. The inference results are specified in the post-condition section (cf. Lines 30–40). Specifically, the two predicates — HasService and Depends — and the function VersionCmp()[1] specify that the target host must provide a service associated with port tp that depends on a vulnerable implementation of SSL. Moreover, every event and predicate in CAML is associated with a time interval during which it is valid (start and end time for an event). The temporal predicate Subset() is used to specify that the HasService and the Depends predicates hold when the matching event instance, denoted by the label $r1$, occurs. If the activity and the pre-condition sections are satisfied, the post-condition section says that a remote execution event may result.

The inference result of the OpenSSL-Handshake-BO-2-Remote-Exec module — that is, remote execution — may be used as an input for another module. Figure 4.2 shows an example of inferring a "data theft" event from a "remote

[1]The function VersionCmp(a, b) compares two strings a and b. It returns an integer less than, equal to, or greater than zero if the version number a is before, the same as, or after version number b.

```
1   module OpenSSL-Handshake-BO-2-Remote-Exec (
2   activity:
3       r1: Event(
4           Source(
5               Node(Address(s: address)))
6           Target(
7               Node(Address(t: address))
8               Service(tp: port))
9           Classification(
10              origin == "cve"
11              name == "CAN-2002-0656"))
12  pre:
13      p1: HasService(
14          Node(Address(address == t))
15          Service(
16              imp: implement
17              ver1: version
18              port == tp))
19      p2: Depends(
20          Source(Service(
21              implement == imp
22              version == ver1
23              port == tp))
24          Target(Service(
25              implement == "OpenSSL"
26              ver2: version)))
27      VersionCmp(ver2, "0.9.6") < 0
28      Subset(r1, p1)
29      Subset(r1, p2)
30  post:
31      Event(
32          starttime == r1.starttime
33          endtime == DEFAULT_ENDTIME
34          Source(
35              Node(Address(address == s)))
36          Target(
37              Node(Address(address == t)))
38          Classification(
39              origin == "vendor-specific"
40              name == "CAM-Remote-Exec"))
41  )
```

Figure 4.1: CAML module: OpenSSL buffer overflow to remote execution

```
1  module Remote-Exec-Access-Violation-2-Data-Theft (
2  activity:
3      r1: Event(
4          Source(
5              Node(Address(a: address)))
6          Target(
7              Node(Address(b: address)))
8          Classification(
9              origin == "vendor-specific"
10             name == "CAM-Remote-Exec"))
11     r2: Event(
12         Source(
13             Node(Address(address == b)))
14         Target(
15             Node(Address(c: address)))
16         Classification(
17             origin == "vendor-specific"
18             name == "CAM-Access-Violation"))
19 pre:
20     StartsBefore(r1, r2)
21 post:
22     Event(
23         starttime == r1.starttime
24         endtime == r2.endtime
25         Source(
26             Node(Address(address == a)))
27         Target(
28             Node(Address(address == c)))
29         Classification(
30             origin == "vendor-specific"
31             name == "CAM-Data-Theft"))
32 )
```

Figure 4.2: CAML module: Remote execution and access violation to data theft

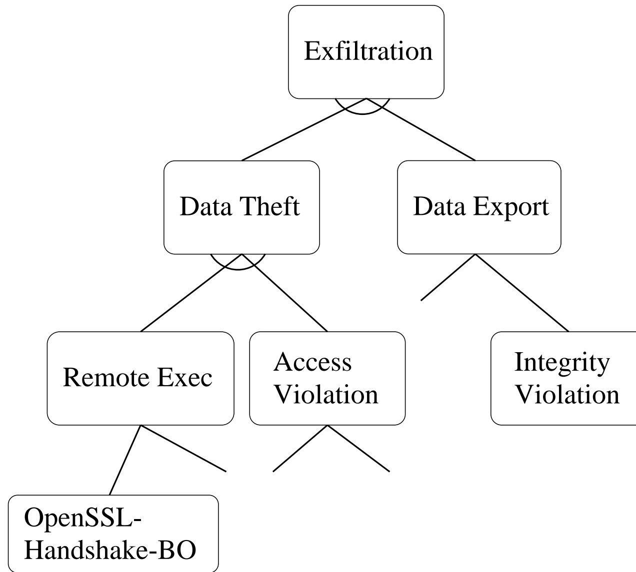


Figure 4.3: Attack model for the exfiltration scenario

execution" event and an "access violation" event. (The latter may be inferred by other sensor reports such as one corresponding to a suspicious file system mount request.) Another module (not shown here) may correlate a "data theft" event with a "data export" event (which may in turn be inferred from an "integrity violation" event corresponding to an unauthorized Web page modification) to detect the exfiltration scenario. The attack model for this scenario is depicted by an AND/OR tree [39] in Figure 4.3.

4.3.2 CAML Modules

A module is the basic unit for specifying correlation steps in CAML. A module specification consists of three sections, namely, *activity*, *pre-condition*, and *post-condition*. To support event-driven inferences, the activity section is used to specify a list of events needed to trigger the module. These events include observed events (corresponding to sensor reports) and inferred events. These events are specified using *event templates*, which describe the requirements for the candidate event instances. The structure of CAML events is based on IDMEF. For instance, the top-level elements of events, similar to those of IDMEF alerts, are Analyzer, Source, Target, Classification, Assessment, and Correlation. CAML also has other fields that do not have counterparts in IDMEF. In particular, there are fields for indicating the start time as well as the end time of an attack (for evaluating temporal interval relationships), for reporting alert counts (to facilitate threshold-based analyses), for reporting thread id's

(to facilitate alert threading), and for reporting correlated results (to support correlating of correlated results).

Labels may be associated with matching event instances or their fields so that they can be referenced in another part of the module. Moreover, simple constraints in the form of event field comparisons may be used to specify the event sets needed by a module. For example, in Figure 4.2, the label b is used as a handle for the target IP address of a "remote execution" event matched by the first event template. It is subsequently used in a constraint for the source IP address of an event instance matching the second event template. CAML can handle the situations in which an event instance may or may not provide data for a field by means of the *optional field* construct. A label preceded by a "?" means that the field is optional. An event instance does not need to have the fields marked optional to match an event template; however, if it does, these values will be used when the corresponding labels are referenced.

For specifying constraints on the system states and the event instances, predicates may be used in the pre-condition section of a module. (We will discuss these predicates in Section 4.3.3.) Examples of system state constraints include restrictions on host or service configurations and the state of server integrity. Examples of event constraints that can be specified in this section include temporal interval relationships among events.

If the activity and the pre-condition sections are met, the inference result specified in the post-condition section will hold (in our model). In particular, a module may derive new system states (in the form of predicates) and inferred events. The derived information may then be used to trigger the inference of other modules. As a result, multistep attack scenarios can be detected by chaining the inferences of CAML modules.

4.3.3 CAML Predicates

To support attack model extension (i.e., ability to incorporate new attack knowledge in attack models) and module composition, we need a uniform way to represent objects and to express their relationships. For the former, CAML uses an IDMEF-based representation for events and predicates. For the latter, we have developed a library of predicates, which functions as a vocabulary to describe the properties of system states and events. Predicate instances may be fed to a correlation engine at startup time; they are specified using a construct called *init section* in CAML. Predicate instances may also be acquired dynamically. In particular, the post-condition section of a module may specify inferred predicates. Every predicate instance has an associated time interval during which it holds. When information needed to determine the truth value of a predicate is not available, the correlation engine could let the evaluation return a default value based on a policy. For example, the policy could state that missing information should not prevent a module pre-condition from being satisfied, and the default truth values could be calculated accordingly.

Currently, several dozen predicate types have been defined in CAML, and they are divided into six categories, namely, *Temporal*, *Hosts*, *Services*, *Files*,

Table 4.1: CAML predicate categories

Category	Description	Example
Temporal	Relationships between two time intervals, based on Allen's work on temporal intervals [1]	*IsBefore(r1,r2)* indicates time interval *r1* ends strictly before the start of time interval *r2*
Hosts	Properties or states of a host	*SuspiciousHost* indicates suspicious activities originated from a specified host have been observed
Services	Properties or states of an operating system or an application instance	*HasService* indicates a specified host provides a specified service
Files	Relationships and properties pertaining to files	*HasFile* indicates a specified host has a file with a specified name
Users	Relationships and properties pertaining to users	*SwitchUser* indicates a specified user at a specified host can "become" another specified user at another specified host
Know	Specifying the predicate instances known by a particular user	Indicates that a specified user knows the password of another specified user at a specified host

Users, and *Know*. The predicate categories are summarized in Table 4.1. All CAML predicates are described in detail in Appendix C.

4.4 Attack Patterns

Developing attack models for multistep attack scenarios could be time-consuming. Moreover, the quality of the models depends heavily on the specifier's experience. Thus, it is important to identify methods for building new attack models based on previously defined ones.

Attack patterns facilitate attack model reuse. These attack patterns correspond to high-level reusable modules that characterize common attack techniques from the detection point of view. The concept of attack patterns is inspired by design patterns [17], which address reuse of software designs and architectures. In particular, software designs that are proven to be effective for solving certain recurring problems in a context are distilled to form design patterns. Similarly, attack patterns are developed to capture the essence of commonly occurring techniques used by attackers. However, the focus for attack patterns is not to facilitate attack development, but to facilitate detection.

A specification of an example attack pattern, called Bandwidth Amplifier, is shown in Figures 4.4 and 4.5. The specification consists of several parts. The *Attack Goal* and the *Considerations* sections provide the context

Pattern Name: Bandwidth Amplifier

Attack Goal: To generate more traffic to jam a target to reduce its availability.
Considerations:
— The communication channel between the source and the target of an attack may have limited bandwidth.
— One may not be able to break in and use other hosts that have high-bandwidth channels to directly attack the target.
— The break-ins on other hosts could be detected and traced.
— The packets sent to the target should have certain varieties. Otherwise, the defender may be able to detect and to block the attack.
Approach: Using an intermediate node that takes a "small" input and generates a "large" output to flood the target. (The size of the input/output may be measured by the number of bytes or the number of packets.) Moreover, by sending requests with a forged source address, equal to the target's address, to this intermediate node, the intermediate node will send the responses to the target. As a result, the amount of traffic going to the target can be increased. Because of the anonymous nature of this technique, it is difficult to trace the attack back to its true source.
Examples:

1. Sending DNS requests with a forged source address to cause (large) DNS responses to be sent to the target. The DNS server is the intermediate node. See AusCERT Advisory AL-1999.004 [3].

2. Sending an ICMP echo request whose source address equals the target's address to a broadcast address. In this case, the intermediate node is the network corresponding to this broadcast address. See CERT Advisory CA-1998-01 [4].

Figure 4.4: Attack pattern: Bandwidth Amplifier (Part I)

for the attack pattern. The former describes the issue the pattern addresses, and the latter discusses the main considerations to determine whether to use this pattern. The *Approach* section describes the attack pattern itself. Known instances of this pattern are described in the *Examples* section. The *Detection* section characterizes this attack pattern from the detection point of view and shows CAML specifications for detecting it. Finally, the *Related Patterns* section describes relationships between this pattern and other attack patterns.

When a new attack is discovered and understood, one may be able to factor the attack into attack patterns. As a result, detecting this attack can be reduced to detecting instances of the attack patterns and their relationships. To illustrate this concept, let us consider two other attack patterns:

Commander-Soldier: This pattern corresponds to a technique to increase

Pattern Name: Bandwidth Amplifier

Detection: A characteristic of this pattern is that one may observe a large amount of (unsolicited) network traffic going to a node. Moreover, the source(s) of this traffic have the small-input-large-output property. A CAML module for detecting this pattern is as follows:

```
module Packet-Flood-2-Bandwidth-Amplifier (
activity:
    r1: Event(
        Source(
            Node(Address(s: address))
            Service(n: name))
        Target(
            Node(Address(t: address)))
        Classification(
            origin == "vendor-specific"
            name == "CAM-Packet-Flood"))
pre:
    p1: SmallInputLargeOutput(
        Node(Address(address == s))
        Service(name == n))
    Intersects(r1, p1)
post:
    Event(
        starttime == r1.starttime
        endtime == r1.endtime
        Source(
            Node(Address(address == s))
            Service(name == n))
        Target(
            Node(Address(address == t)))
        Classification(
            origin == "vendor-specific"
            name == "CAM-Bandwidth-Amplifier"))
)
```

Related Patterns: Depending on the amplification ratio, the amplified traffic may not be sufficient to jam the target. *Commander-Soldier* may be used with *Bandwidth Amplifier* to consume more network bandwidth of the target.

Figure 4.5: Attack pattern: Bandwidth Amplifier (Part II)

the attack power and to hide the true source of an attack by managing a set of nodes to attack a target. This pattern has two types of components, namely, commanders and soldiers. A commander coordinates a number of soldiers to attack the same target, and the soldiers carry out the attack actions.

PERSISTENT INQUIRER: This pattern corresponds to a technique that attempts to consume the resources of a node by continuously sending requests to it to prevent it from serving legitimate requests. An example is TCP SYN flooding.

Using these attack patterns, one could model distributed denial-of-service (DDOS) attacks like mstream [6] and TFN [5]. The mstream DDOS attack uses TCP ACK flooding to consume the CPU time of the target. Moreover, multiple machines are used to generate more TCP ACK packets. The mstream attack can be characterized by the attack patterns PERSISTENT INQUIRER and COMMANDER-SOLDIER and their relationships. Another DDOS attack called TFN uses different tactics including TCP SYN flooding and smurf, a DOS attack based on ICMP directed broadcast. A model for TFN may be constructed using three attack patterns, namely, PERSISTENT INQUIRER, BANDWIDTH AMPLIFIER, and COMMANDER-SOLDIER. Depending on which tactic(s) an attacker uses, an instance of TFN may exhibit only behavior pertaining to PERSISTENT INQUIRER or BANDWIDTH AMPLIFIER. Thus, the TFN model should specify an "or" relationship between PERSISTENT INQUIRER and BANDWIDTH AMPLIFIER.

4.5 Implementing a Scenario Recognition Engine

To validate the practical usefulness of the CAML modeling language, we implemented a scenario recognition engine that uses attack specifications written in CAML. The implementation integrates two advanced technologies developed in the EMERALD program [27, 31], *P-BEST* and *eflowgen*. P-BEST is an expert system shell for building real-time forward-reasoning expert systems based on production rules [23]. Developed as a platform for the EMERALD M-correlator [30], eflowgen is an extensible application framework where application-specific processing modules are triggered in response to events (e.g., messages, timers, and file and network I/O). These processing modules may be dynamically created, modified, reordered, and destroyed. In the scenario recognition engine, eflowgen receives and processes incoming reports and asserts them as facts into the P-BEST factbase. When a fact has been asserted, eflowgen calls the P-BEST inference engine.

4.5.1 Translating CAML to P-BEST

The first step in building a P-BEST inference engine based on a CAML model is to translate the CAML specification into the P-BEST language. In the described

pilot implementation, this translation was performed manually. A CAML module maps fairly well into a P-BEST rule, by letting the activity and pre-condition sections form the antecedent of the rule, while the post-condition section becomes the consequent. Predicates are implemented as facts, each representing a specific predicate, using a generic fact type (P-BEST *ptype*) with a large number of member fields. Each different predicate typically uses a small subset of the available fields. In the traditional version of P-BEST, this would have been unusable, because it required every field of a fact to be populated. We have modified P-BEST to allow sparsely populated facts and added a function that can test if a given field is populated or not. Report events are naturally mapped into P-BEST facts.

4.5.2 Validation Scenario

In DARPA's Cyber Panel program, a project called the Grand Challenge Problem (GCP) has developed example attack scenarios that can be used for testing and evaluation of detection and correlation technologies. The example mission system in the GCP consists of multiple enclaves with heterogeneous subsystems that are used jointly to perform a mission-critical function.

For the validation of our scenario recognition engine, we chose an attack scenario from GCP version 2.0. The scenario is composed of several coordinated attacks, some of which must occur in a certain temporal order. The resulting CAML specification consists of 14 modules.

The GCP provides alerts in IDMEF (XML) format from intrusion detection sensors enabled at various locations in the example mission system during the attack scenario. The eflowgen component of our scenario recognition engine was instrumented to directly consume the IDMEF alerts and map the information into P-BEST facts.

4.5.3 Results

The scenario recognition engine could correctly identify the modeled attack scenario from the alert reports. However, the processing latency on a regular desktop computer was in the order of 3 minutes, which is too slow for real-time deployment. Analysis of the runtime behavior of the engine showed that certain parts of the translated CAML code caused combinatorial explosions in P-BEST (nested loops) with pessimistic evaluation. Basically, all expressions were placed in the innermost loop. This problem has not been observed in the extensive use of P-BEST in development of intrusion detection sensors, because such P-BEST code typically has very few fact-matching clauses in rule antecedents. CAML specifications, on the other hand, tend to result in complex antecedents causing this problem to manifest.

We addressed this problem by developing realistic (as opposed to pessimistic) loop evaluation. The P-BEST language was extended with hints that tell the P-BEST compiler on which nesting level a clause with implicit fact references should be placed. We also added explicit syntactical constructs to the P-BEST

language for placing selected evaluations outside the nested loops. For example, this can be used for global variables that are independent of facts.

The optimizations resulted in the processing time for the example scenario being reduced from 3 minutes to 1 second, which makes it feasible to deploy the scenario recognition engine in real-time situations.

4.6 Related Work

As observed by Eckmann *et al.*[14], several distinct classes of languages are used to encode different aspects of attacks. In their terminology, CAML and other languages that are used to analyze security alerts belong to the correlation language category.

Alert correlation has been proposed to address the difficult problems of analyzing a large number of alerts (possibly generated by heterogeneous sensors), identifying the security-critical ones and discounting the false alarms, and producing high-level reports to summarize and to explain the alerts. Exemplary intrusion correlation work includes probabilistic correlation based on attribute similarity [38], mission-impact-based correlation that employs common-attribute aggregation, topology analysis, and mission-criticality analysis to perform incident ranking [30], alarm clustering to support root cause analysis [19], IBM Zurich's Tivoli Enterprise Console that employs common-attribute clustering, alert duplication recognition, and event consequence detection [12], Honeywell's Scyllarus correlation framework [18], and a simulated-annealing-based approach for detecting stealthy portscans [35].

For detecting multistep attack scenarios, a naive approach is to use an attack signature that explicitly specifies the constituent attacks and the ordering among them. In fact, this approach is commonly used in signature-based intrusion detection for detecting attacks that involve multiple events. However, extending this approach to recognize (complex) attack scenarios has weaknesses. In particular, because some of the attacks may be substituted by other functionally equivalent ones and the ordering of the attacks could be changed without affecting the outcome, there may be many different variations of an attack scenario. Also, it is difficult to extend these attack signatures to incorporate new attack information.

To address these weaknesses, an attack modeling approach based on specifying the pre-condition and the post-condition of individual attacks has been proposed in Jigsaw [37], LAMBDA [10], ADeLe [25], and by Ning *et al.* [28]. Attacks are related to each other through matching the post-condition of an attack with the pre-condition of another. Moreover, this approach facilitates the specification of functionally equivalent attacks and of new attacks as these attacks can be specified individually. CAML also uses this modular approach for modeling attack scenarios. To support module composition and attack model extension, it is very important to have a uniform interface among modules. This work differs from prior work in that it focuses on a uniform representation of objects and their relationships and on attack model reuse. We have developed a

library of predicates, which functions as a vocabulary to describe system states and events. Developing attack models for multistep scenarios could be time-consuming and complicated. To facilitate reuse of the results of prior modeling efforts, we present a method based on characterizing common attack techniques and using them as higher-level abstractions in attack scenario modeling.

Attack scenario recognition shares many similarities with vulnerability analysis for complex computer and network systems (e.g., [32–34, 40]). In particular, to discover the vulnerabilities of a network, one may need to reason about the configurations of individual hosts, the vulnerabilities of the hosts, and the connectivity and interdependencies among them. Moreover, in vulnerability analysis, characterizing attacks in terms of pre- and post-conditions has also been found useful to infer attack sequences that violate a security policy.

Chapter 5

Adaptive Sensing and Correlation

We have developed extensions to the CAML scenario recognition engine that will introduce dynamic and adaptive functionality. We envision an intelligent component that not only listens to sensors, but actively tunes sensors and other components based on its global view of the cyber battlefield. For example, in a situation of increased threat level, it could invoke additional analysis engines, trigger response components, or update configurations of sensors to extend their monitoring or make sensors switch into an active probing mode [22]. Furthermore, the engine will be able to dynamically accept new or updated CAML models and update its knowledge base with those models during runtime. We present an architecture that supports highly dynamic monitoring and correlation and a language for specification of active component behavior.

5.1 Introduction

Like so many other activities, cyber security can be viewed as an optimization problem—the defender seeks to minimize costs while maximizing benefits. There is a cost associated with a successful attack against your system, and there are costs associated with the defense mechanisms you deploy. If defense mechanisms were very inexpensive in all respects, you would probably deploy them generously and ubiquitously in order to minimize the risk of a costly attack. However, real-world defense mechanisms always come with a nonzero price tag, so we need strategies and tactics for how and when to deploy them. Examples of the types of costs that defense mechanisms could present include hardware, software, bandwidth, labor, degradation or loss of services, loss of goodwill (as a consequence of misdirected response), and even help to the attackers as the mechanism could itself be vulnerable or leak information.

There are limitations to the amount of optimization that can be done statically, with respect to the deployment and configuration of defense mechanisms.

An attack or a legitimate change to the protected system could completely change the parameters upon which the optimization was based. If we could automatically and dynamically adapt to changes in the cyber battlefield situation, we could continuously optimize the cost-benefit ratio of our defense mechanisms. The adaptation could also include adjustment to parameters such as the maximum cost we are willing to allow security mechanisms to impose—this value is likely to be higher after an attack has inflicted some damage.

Although our discussion so far applies to all kinds of defense mechanisms, the primary focus of this text is on mechanisms that provide monitoring and reporting, that is, intrusion detection sensors and event correlators. To a certain extent, we also look at response actuators that are triggered by the monitors.

Why do we need to optimize monitoring—can't we just monitor everything that goes on everywhere in our system? To monitor every event on every host, we would most likely need to double or triple our hardware capacity, as every host computer would need to be accompanied by one or more monitoring computers that do nothing else than analyze every single operation taking place on the monitored host. In general, it takes more computing power to perform intelligent analysis of an event record than to produce one. For network monitoring, we could cover a network segment from a strategically chosen location, but if traffic speeds and volumes are high, the hardware costs could be considerable.

We conclude that in most cases it is infeasible to monitor everything everywhere, so we need to decide *where* to monitor and *what* to monitor. This decision needs to be made dynamically so that we can adapt to changes in the monitored system and the current threat situation. In this text we describe an architecture and design that will support such adaptive capabilities.

5.2 Architecture

The architecture for adaptive sensing and correlation, depicted in Figure 5.1, includes an adaptive correlation engine and a set of distributed, dynamically deployable and tunable sensors. Based on observed activities and knowledge of the protected system, the correlation engine adaptively manages a set of sensors to optimize the overall detection performance.

The adaptive correlation engine has two main parts: knowledge base and control. The knowledge base consists of attack models, system models, and models for adaptive behavior. The attack models characterize attacks (including multistep attack scenarios) from the detection point of view. The system models describe the states and configurations of the protected systems and the criticality of the system components with respect to the missions they support. The models for adaptive behavior specify what actions to perform and when to perform them.

The control part of the correlation engine is responsible for interacting with a control client, managing sensors, and performing inferences to correlate sensor reports and detect multistep attacks. The control client is used by a security administrator to change the behavior of the correlation engine, for example,

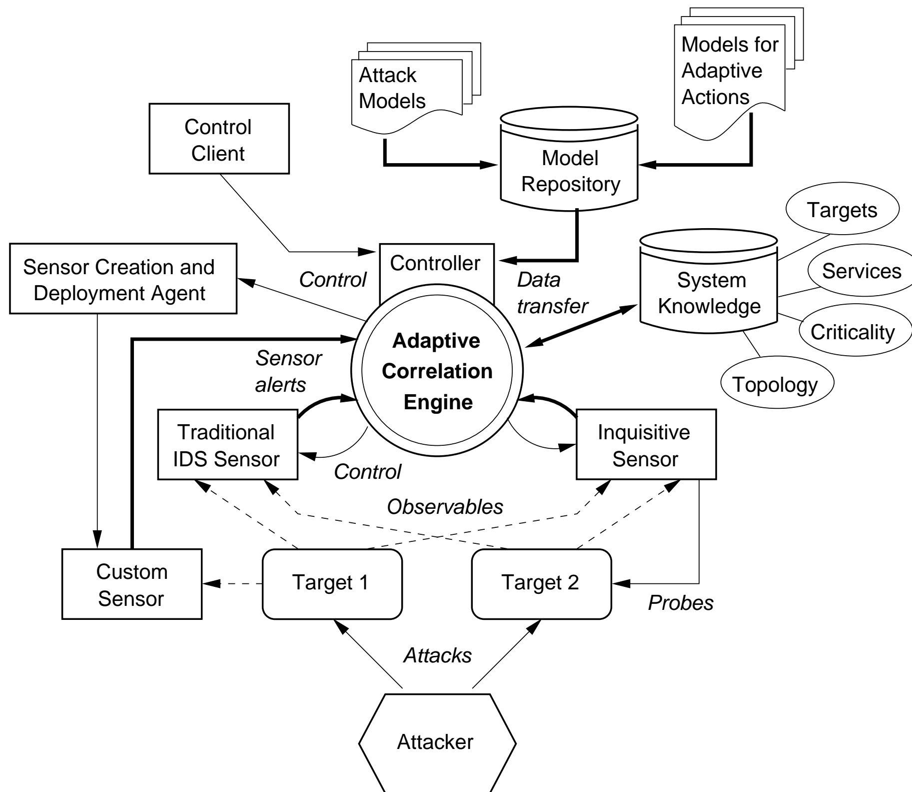


Figure 5.1: Architecture for adaptive sensing and correlation

loading a new attack model to the engine. Based on the current status of the operating environment and knowledge for the protected systems, the correlation engine may change the sensor landscape by reconfiguring, (de)activating, creating, and deploying sensors. The correlation engine accepts reports from heterogeneous sensors and correlates them to produce higher-level reports.

The adaptive correlation engine dynamically deploys sensors at selected locations to optimize the overall detection performance. These sensors include traditional sensors that passively monitor system activities, *inquisitive sensors* [22] that may actively probe system components to diagnose a situation based on observed activities, and custom sensors that are dynamically created and deployed to detect new or specialized attacks.

5.2.1 Engine Management Protocol

A fundamental aspect of adaptive sensing and correlation is the ability to dynamically update the correlation engine with new attack models as those models become available. New models could be the result of new types of attack scenarios becoming known and encoded, or the library of models being extended with larger numbers of known attack scenarios.

We have designed and implemented a communications protocol and operating procedures for how the CAM scenario recognition engine should be updated with new attack scenario models during runtime. The basic operating model is illustrated in Figure 5.2. The complete specification of the protocol can be found in Appendix D.

An attack model described in CAML is translated into a P-BEST rules file. The P-BEST rules are translated into C by pbcc and compiled into object code by the C compiler (gcc). The object code is specific to a selected hardware architecture and operating system platform.

A Control Client is used to send commands to the CAM Engine Controller. The set of available commands enables the client to perform the following operations:

Status query When the client sends a status query, the controller reports a number of parameters. The status report includes a flag indicating whether the engine is running or stopped, the architecture on which the engine is residing, and a listing of the attack models currently loaded.

Start engine This command instructs the controller to start the correlation engine.

Stop engine This command instructs the controller to stop the correlation engine. When the engine is stopped, it discards incoming messages.

Load module This command instructs the controller to retrieve a new object code file for the correlation engine and load the new code into the engine. The order in which object code files are loaded is the order in which the models will be applied to incoming events. The loading of new object

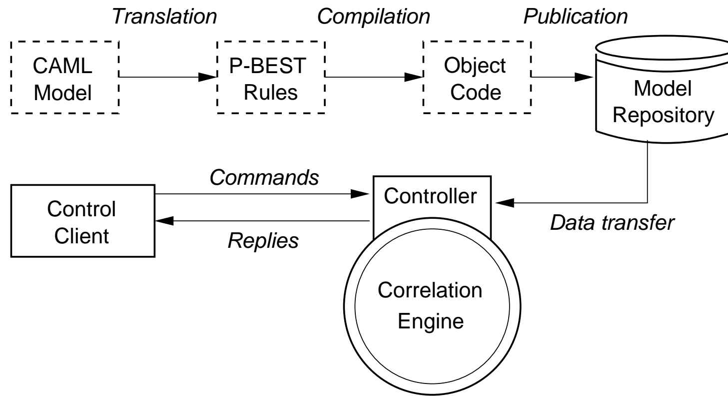


Figure 5.2: Basic operating model for correlation engine management and run-time updates

code files will not cause any previously loaded files to be unloaded — the unload command must be used to explicitly unload a file.

Unload module This command is used to remove models from the correlation engine.

5.2.2 Models for Adaptive Behavior

To support the active behavior of an adaptive monitoring system, one needs a model to specify what actions to perform and when to perform them. We have developed a language, called *Specification Language for Active Monitoring (SLAM)*, for this purpose. SLAM uses an event-driven approach for specifying the behavior of an adaptive monitoring system.

A SLAM specification consists of a set of *triggers* and *configuration modules*. Basically, SLAM triggers specify a set of events, the context in which they occur, and the corresponding actions to perform. Moreover, SLAM configuration modules specify assignments from triggers to monitoring components and the command-invocation relationships among these components (e.g., a component may invoke another component to perform a certain action).

A SLAM trigger contains an activity section, a pre-condition section, and an action section. The activity section and the pre-condition section are used for specifying a set of events and the context in which they occur respectively; they are the same as those of CAML (see Chapter 4). The action section contains *commands*. Every command has a command name and a list of arguments. If a set of events can satisfy the activity section and the constraints in the pre-condition section are met, the commands in the action section will be performed. A command may return a result. For more complex action section specifications, control-flow statements may be used. (Currently, only C-style

"if-else" statements are allowed.) To support commands that have optional arguments or allow different combinations of arguments, tagged arguments may be used.

Figure 5.3 shows a SLAM trigger example. It specifies that a potential denial-of-service event against port p of host t will trigger a Probe-Service-Status action to verify the status of the service at port p of host t. If the service is determined to be unavailable, an IDMEF alert is sent to a correlation engine, which may increase the confidence level of the denial-of-service alert.

```
trigger Check-Service-Avail (
activity:
    Event(
    Target(
        Node(Address(t: address))
        Service(p: port
            proto: protocol))
    Assessment(
        Impact(type == "dos")
        Confidence(rating != "high")))
action:
    if (Probe-Service-Status(targetIP=t, targetPort=p, protocol=proto)
        == SVC_DOWN)
        Send-IDMEF (recp = correlator, idmef =
            Event(
                Target(
                    Node(Address(address == t))
                    Service(port == p
                        protocol == proto))
                Classification(
                    origin == "vendor-specific"
                    name == "SVC_DOWN")
                Assessment(
                    Confidence(rating == "high"))))
)
```

Figure 5.3: Example SLAM trigger

SLAM configuration modules serve two main purposes: specifying trigger-component assignments and specifying delegation relationships among components. An assignment of triggers to monitoring components is specified using the *task* construct. A task groups a set of SLAM triggers to form a higher-level entity, which can be deployed as a unit. An assignment can be viewed as a many-many mapping between triggers and components. That is, a trigger may be mapped to multiple components, and a component may be mapped to multiple triggers. To facilitate building complex tasks, a task can be defined to include the trigger-component assignment of another task.

A monitoring component may need to delegate the responsibility of performing an action to another component. For example, a correlation engine may be incapable of probing the status of a server and need to invoke another component to perform that action. To specify these delegation relationships, SLAM configuration modules may be used. Specifically, the following types of con-

figuration modules are used—component definitions, command interfaces, and command configuration rules. A component definition specifies the attributes of the component and associates a unique component identifier to it. A command interface specifies the name of the command, the type of the return value, and a list of command arguments and their types. A command configuration rule specifies the commands supported by a component and the conditions in which another component can invoke them.

5.3 Alert Correlation and Scenario Recognition

An adaptive monitoring system can increase the robustness of the detection system by adaptive sensor management, improve the quality of detection reports by deploying runtime checks to verify the outcome and the relevance of an attack and by incorporating new attack models, and optimize the use of resources through analyzing the system activities and incorporating information about the current threat level.

5.3.1 Increasing Robustness

To avoid being detected, an adversary may deploy evasion tactics or even attack the monitoring system itself. Thus it is important to build a robust detection system. Techniques for increasing the robustness of the detection system include (1) increasing the difficulty for an adversary to determine the current state of the detection system, for example, by hiding or migrating sensors, and (2) building an attack-tolerant system through the use of redundancy and diversity.

We use an example to illustrate how a correlation engine can employ adaptive sensor management to make the detection system more robust. Let us assume that the correlation engine has received sensor reports indicating that a certain host may been compromised. Suppose that host is used for performing network intrusion detection for a subnet. The alert stream from this NIDS (network intrusion detection system) may no longer be reliable. If a static monitoring system is used, the issue may be fixed only after a security administrator receives and analyzes the alerts and performs a recovery action. Thus, there will be a relatively large time window during which the attack detection capability is compromised. On the other hand, an adaptive correlation engine may activate a replacement NIDS (on another host) to maintain detection coverage.

This behavior of an adaptive correlation engine can be expressed using the SLAM trigger specification as shown in Figure 5.4. The pre-condition section specifies two predicates—there is a suspicious host and an NIDS runs on the host—that need to be satisfied to activate the trigger. The action section specifies the commands that will be performed—for example, locating and activating the replacement NIDS—if the pre-condition section is satisfied.

```
trigger Activate-Backup-NIDS (
pre:
    SuspiciousHost(Node(Address(a: address)))
    HasService(
        Node(Address(address == a))
        Service(name == "NID"))
action:
    backupNID = LocateBackup (ipaddr = a, service = "NID")
    Activate-Sensor (comp = backupNID)
    Deactivate-Sensor (ipaddr = a, service = "NID")
)
```

Figure 5.4: SLAM trigger: Activate-Backup-NIDS

5.3.2 Adapting to System Changes

From an intrusion detection viewpoint, changes may come in different forms. First, there may be changes to the monitored system, such as new services added to a host. Second, the importance of a resource may change because of changes in security policies, user interest levels, or the availability of related resources. Third, the model repository may be updated, for example, when there is a new attack to detect.

A monitoring system can adapt to changes by modifying the sensor landscape. For instance, when a model for a new multistep attack scenario, as described in [9], is developed and deployed, a correlation engine may activate additional sensors, reconfigure existing sensors, or even create new specialized sensors to ensure that the sensor reports needed by the correlation engine will be available when the attack scenario occurs.

As an example, suppose an attack step of the new multistep attack scenario is not currently detected, and the detection system may detect this step either reconfiguring an NIDS to increase its coverage or activating a host-based IDS to detect the step. To support this behavior, the correlation engine needs to have the knowledge about the status and the capabilities of the components in the system. This knowledge of component capabilities can be specified using the SLAM configuration modules shown in Figure 5.5.

```
// Component definitions:
host solar : dname = "sun.cam.org"
process myCorrelator : dname = "cam-corr.cam.org",
  tcpport = DEFAULT-CAMCORR-LISTEN-PORT

// Command interfaces:
int Start-Sensor (string sensor)

// Command configuration rules:
r1: myCorrelator -> solar; Start-Sensor;
    sensor == "eXpert-BSM"
```

Figure 5.5: SLAM configuration modules: Host-based IDS on solar

In Figure 5.5, the component definitions associate the identifier solar to the host whose domain name is sun.cam.org and the identifier myCorrelator to the correlation engine. The command interface Start-Sensor defines the argument(s) and the return value type for the command. The command configuration rule r1 specifies that myCorrelator can invoke solar to start the host-based IDS, eXpert-BSM [24] there.

5.3.3 Adapting to Attacks

The operational costs of running sensors can be reduced if one can adapt the sensing landscape to the attack activities. Through dynamic sensor management, some sensors can be turned off or put in a low-cost mode most of the time, and are activated only when certain attack precursors are observed. Later, when a reduced level of attack activities is perceived, the monitoring system can return to a lower level to conserve resources.

For instance, when there is an increase in threat level, for example, detected by an early warning system, one may want to raise the monitoring level across the whole administrative domain. An adaptive detection system may then activate additional intrusion detection components and reconfigure certain components to run in a more secure albeit costly mode.

```
task SecLevel3 {
    Check-Service-Avail -> inq-IDS-set
    include SecLevel2
}

trigger Elevate-SecLevel2 (
pre:
    p1: AggregateAttackLevel(val == "High")
    p2: CurrentSecurityLevel(val == "2")
    OverlapWith(p1,p2)
action:
    Deactivate-Task("SecLevel2")
    Activate-Task("SecLevel3")
)
```

Figure 5.6: SLAM: Elevating monitoring level

Figure 5.6 shows an example SLAM specification for increasing the monitoring level from 2 to 3 when there is a high level of aggregate attack activities in the current time period. The task module, which assigns triggers to components, SecLevel3 includes the assignments specified in the task module SecLevel2. Moreover, it also enables the active behavior pertaining to the Check-Service-Avail trigger (see Section 5.2.2) for the set of inquisitive sensors inq-IDS-set to obtain more informative sensor reports.

5.4 Creation and Deployment of Custom Sensors

Situations exist where the most cost-effective adaptive action would be to dynamically create a highly specialized sensor and deploy it in a selected location. The degree of sophistication and novelty could vary greatly in this category of dynamically created sensors. It is hard to envision automated creation of sensors for completely new event streams, but if we assume that a basic event collection framework exists for the event stream in question, it should be possible to conjure and deploy new detection mechanisms. Examples of such mechanisms range from patterns for event string matching to algorithmic specifications such as state machines for protocol analysis and sets of heuristics for detection of complex event combinations.

5.5 Related Work

Lindqvist's [22] paper proposed using active sensors to aid the determination of attack outcome and alert relevance. IDS reconfiguration to tradeoff performance with workload has been investigated in Lee et al.'s paper [21] and Feiertag et al.'s papers [15, 16]. Moreover, Feiertag et al. also suggested adding new detection capabilities to an IDS as another form of adaptation. Mounji and Le Charlier [26] proposed coupling an IDS with a configuration analysis subsystem to conserve IDS resource and to ignore activities that cannot affect the system by enabling only those IDS rules that are relevant to the current system configuration. Anomaly intrusion detection systems—for example, NIDES [2]—may adapt user profiles to reflect changes in usage patterns to avoid false alarms. To improve detection coverage, techniques for learning and blocking unknown attacks have been presented in Just et al.'s paper [20]. For more specific uses, active detection techniques have been proposed by Cheung and Levitt [8] for identifying invalid DNS data, by Templeton and Levitt [36] to detect packet spoofing, and for identifying attack sources (see Dunigan's survey [13]). Outside the computer security domain, active systems have also been studied for databases (see Paton's and Diaz's survey [29]).

- *(Attack outcome and alert relevance)* Lindqvist [22] proposed the concept of inquisitive sensors. The main applications of these sensors discussed in the paper are confirming compromise and determining alert relevance, for example, checking if a NIC is in promiscuous mode, querying a network server to determine its availability, identifying the status of critical processes, on-the-fly vulnerability scanning, and network topology discovery.
- *(Performance tradeoffs)* Lee et al. [21] presented IDSs that autonomously configure themselves on-the-fly to trade off detection coverage against workload. The authors formulated performance tradeoff as an optimization problem that maximizes the detection value of an IDS configuration subject to a set of resource constraints.

- *(New detection capabilities and performance adaptation)* Feiertag et al.'s papers [15, 16] describe two classes of situations in which an IDS may need to adapt to the changing environment: (1) The IDS acquires a new capability, for example, a new attack signature or a new component. (2) The IDS is overloaded, for example, overwhelmed by a flooding attack. For the latter, the IDS may want to reconfigure the event collection module to reduce the scope of its analysis, or to invoke a response to cut off the source of the attack.

- *(Resource conservation and alert relevance)* Mounji and Le Charlier [26] proposed coupling an IDS with a configuration analysis subsystem to tune the IDS based on continuous assessment of the system configuration. In particular, certain IDS rules are enabled/disabled to reflect the current system state. A major potential benefit of their approach is conserving resources of the IDS. Moreover, because only IDS rules relevant to the current system configuration are enabled, activities that are not likely to affect the system will not be reported.

- *(User adaptation)* In anomaly intrusion detection systems, such as NIDES, a profile to characterize normal user activity is constructed using previously observed events for the subject. When a statistically significant deviation is detected between observed activity and the profile, an alert is generated. To cope with changes in usage patterns over time, the user profile is continuously updated to reduce false alarms.

- *(Detecting and blocking unknown attacks)* Just et al.'s paper [20] presented an approach to cope with unknown attacks that involves learning an attack based on observing system failures and developing filter rules to block future occurrences of the attack.

- *(Detecting invalid DNS data)* In Cheung's and Levitt's paper [8] on protecting DNS servers, a security wrapper is used to ensure that the content of response packets received is consistent with the "authoritative answers" (i.e., the DNS data maintained by the corresponding authoritative DNS servers). If data in a DNS response comes from a server that is not authoritative, instead of generating an alert, the wrapper will locate an authoritative server and query it to obtain the authoritative answer for detecting invalid DNS data.

- *(Packet spoofing detection)* In Templeton's and Levitt's paper [36] on detecting packet spoofing, active methods are proposed to verify the source of a packet. Examples include probing the host corresponding to the source address specified in the packet to obtain information about TTL, IP identification number, and operating system deployed. For TCP traffic, one may also inject special packets in a TCP stream to cause certain flow control behavior or retransmissions, and observe whether the incoming traffic pattern is consistent with that to detect packet spoofing.

- *(Packet tracing)* Dunigan's report [13] surveys techniques and systems for packet tracing to identify the source of an attack. Some of these techniques are effective only for ongoing attacks. Having an active sensor or correlation system that invokes packet tracing when an attack is observed is useful, especially for attacks that are too short-lived for human response.

- *(Active database systems)* In an active database system, actions may be performed automatically to respond to events. Paton's and Diaz's paper [29] surveys active database systems. The behavior of active database systems may be expressed by event-condition-action rules. In particular, the event part of the rule describes a happening (for example, insertion of a new record); the condition part describes the context to trigger the rule; the action part describes the actions to perform when both the event part and the condition part are satisfied.

Chapter 6

Concluding Remarks

We have presented methods and a language, called CAML, for modeling multistep attack scenarios in a way that enables correlation engines to use these models to recognize attack scenarios. CAML uses a modular approach for specifying attack scenarios, making the models expressive and extensible. Each module represents an inference step, and these modules can be linked together to recognize attack scenarios. To facilitate module interfacing, CAML has a set of predicates for specifying the properties of system states and events, and employs a uniform representation for events and predicates. A concept called *attack patterns* facilitates reuse of modules and attack models. To validate our approach, we implemented a prototype scenario recognition engine that uses CAML specifications to identify an attack scenario in a stream of IDMEF-encoded alerts.

Furthermore, we have presented a cyber defense architecture centerered around the scenario recognition engine. This architecture allows for dynamic and adaptive monitoring that can be continuously optimized for cost efficiency and survivability. We have developed a language in which the active behavior of an adaptive monitoring system can be specified.

These results are important for the emerging fields of research in alert correlation and management, coordinated cyber defense, cyber situational awareness, and cyber indications and warnings and attack prediction and response.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion-detection expert system (NIDES). Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, Menlo Park, California, May 1995.

[3] Australian Computer Emergency Response Team. *Denial of Service (DoS) attacks using the Domain Name System (DNS)*, Aug. 13, 1999. AusCERT Advisory AL-1999.004.

[4] CERT Coordination Center. *Smurf IP Denial-of-Service Attacks*, Jan. 5, 1998. CERT Advisory CA-1998-01.

[5] CERT Coordination Center. *Distributed Denial of Service Tools*, Nov. 18, 1999. CERT Incident Note IN-99-07.

[6] CERT Coordination Center. *"mstream" Distributed Denial of Service Tool*, May 2, 2000. CERT Incident Note IN-2000-05.

[7] CERT Coordination Center. *Buffer Overflow In IIS Indexing Service DLL*, June 17, 2001. CERT Advisory CA-2001-13.

[8] S. Cheung and K. N. Levitt. A formal-specification based approach for protecting the domain name system. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 641–651, New York, New York, June 25–28, 2000.

[9] S. Cheung, U. Lindqvist, and M. W. Fong. Modeling multistep cyber attacks for scenario recognition. In *DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 284–292, Washington, D.C., Apr. 22–24, 2003.

[10] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In H. Debar, L. Mé, and S. F. Wu, editors, *Recent Advances in Intrusion Detection (RAID 2000)*, volume 1907 of *LNCS*, pages 197–216, Toulouse, France, Oct. 2–4, 2000. Springer-Verlag.

[11] D. Curry and H. Debar. *Intrusion Detection Message Exchange Format: Data Model and Extensible Markup Language (XML) Document Type Definition*. Intrusion Detection Working Group, June 20, 2002. Work in progress, IETF Internet-Draft draft-ietf-idwg-idmef-xml-07.txt.

[12] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In W. Lee, L. Mé, and A. Wespi, editors, *Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 85–103, Davis, California, Oct. 10–12, 2001. Springer-Verlag.

[13] T. Dunigan. Backtracking spoofed packets. Technical Report ORNL/TM-2001/114, Network Research Group, Oak Ridge National Laboratory, June 2001.

[14] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10:71–103, 2002.

[15] R. Feiertag, L. Benzinger, S. Rho, S. Wu, K. Levitt, D. Peticolas, M. Heckman, S. Staniford-Chen, and C. Zhang. Intrusion detection inter-component adaptive negotiation. In *Web proceedings of the Recent Advances in Intrusion Detection (RAID 1999)*, 1999. `http://www.raid-symposium.org/raid99/`.

[16] R. Feiertag, S. Rho, L. Benzinger, S. Wu, T. Redmond, C. Zhang, K. Levitt, D. Peticolas, M. Heckman, S. Staniford-Chen, and J. McAlerney. Intrusion detection inter-component adaptive negotiation. *Computer Networks*, 34(4):605–621, Oct. 2000.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] R. P. Goldman, W. Heimerdinger, S. A. Harp, C. W. Geib, V. Thomas, and R. L. Carter. Information modeling for intrusion report aggregation. In *DARPA Information Survivability Conference and Exposition (DISCEX II)*, volume 1, pages 329–342, Anaheim, California, June 12–14, 2001.

[19] K. Julisch. Mining alarm clusters to improve alarm handling efficiency. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, pages 12–21, New Orleans, Louisiana, Dec. 10–14, 2001.

[20] J. E. Just, J. C. Reynolds, L. A. Clough, M. Danforth, K. N. Levitt, R. Maglich, and J. Rowe. Learning unknown attacks — a start. In A. Wespi, G. Vigna, and L. Deri, editors, *Recent Advances in Intrusion Detection (RAID 2002)*, volume 2516 of *LNCS*, pages 158–176, Zurich, Switzerland, Oct. 16–18, 2002. Springer-Verlag.

[21] W. Lee, J. B. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In A. Wespi, G. Vigna, and L. Deri, editors, *Recent Advances in Intrusion Detection*

(RAID 2002), volume 2516 of *LNCS*, pages 252–273, Zurich, Switzerland, Oct. 16–18, 2002. Springer-Verlag.

[22] U. Lindqvist. The inquisitive sensor: A tactical tool for system survivability. In *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pages C–14–C–16, Göteborg, Sweden, July 1–4, 2001.

[23] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 9–12, 1999.

[24] U. Lindqvist and P. A. Porras. eXpert-BSM: A host-based intrusion detection solution for Sun Solaris. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, pages 240–251, New Orleans, Louisiana, Dec. 10–14, 2001.

[25] C. Michel and L. Mé. ADeLe: An attack description language for knowledge-based intrusion detection. In M. Dupuy and P. Paradinas, editors, *Trusted Information: The New Decade Challenge: IFIP TC11 16th International Conference on Information Security (IFIP/SEC'01)*, pages 353–368, Paris, France, June 11–13, 2001.

[26] A. Mounji and B. L. Charlier. Continuous assessment of a Unix configuration: Integrating intrusion detection and configuration analysis. In *Proceedings of the ISOC 1997 Symposium on Network and Distributed Systems Security*, San Diego, California, Feb. 10–11, 1997.

[27] P. G. Neumann and P. A. Porras. Experience with EMERALD to date. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, Apr. 9–12, 1999.

[28] P. Ning, S. Jajodia, and X. S. Wang. Abstraction-based intrusion detection in distributed environments. *ACM Transactions on Information and System Security*, 4(4):407–452, Nov. 2001.

[29] N. W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[30] P. A. Porras, M. W. Fong, and A. Valdes. A mission-impact-based approach to INFOSEC alarm correlation. In A. Wespi, G. Vigna, and L. Deri, editors, *Recent Advances in Intrusion Detection (RAID 2002)*, volume 2516 of *LNCS*, pages 95–114, Zurich, Switzerland, Oct. 16–18, 2002. Springer-Verlag.

[31] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland, Oct. 7–10, 1997.

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

[33] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 156–165, Oakland, California, May 14–17, 2000.

[34] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 273–284, Oakland, California, May 12–15, 2002.

[35] S. Staniford, J. A. Hoagland, and J. M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10:105–136, 2002.

[36] S. Templeton and K. Levitt. Detecting spoofed packets. In *DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 164–175, Washington, D.C., Apr. 22–24, 2003.

[37] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 New Security Paradigms Workshop*, pages 31–38, Ballycotton Co., Cork, Ireland, Sept. 18–21, 2000.

[38] A. Valdes and K. Skinner. Probabilistic alert correlation. In W. Lee, L. Mé, and A. Wespi, editors, *Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 54–68, Davis, California, Oct. 10–12, 2001. Springer-Verlag.

[39] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1977.

[40] D. Zerkle and K. Levitt. NetKuang–a multi-host configuration vulnerability checker. In *Proceedings of the Sixth USENIX Security Symposium*, pages 195–204, San Jose, California, July 22–25, 1996.

Appendix A

Attack Scenario References

Ackcmd

Tool that provides a remote shell that can bypass some firewalls by using packets with the 'ack' bit set to create an 'ack tunnel'.
`http://www.ntsecurity.nu/toolbox/ackcmd`

ASP file to upload tools over HTTP

Bugtraq posting
`http://marc.theaimsgroup.com/?l=bugtraq\&m=98040935006042\&w=2`

BIND Exploit

Exploits bind TSIG vulnerability outlined in CVE-2001-0010. Exploit returns a root shell.
Link to vulnerability information:
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0010`
Link to exploit:
`http://www.hack.co.za/download.php?sid=1186`

Elitewrap

Tool that allows executable files to be packaged together as one file and then executed synchronously or asynchronously and can be hidden from the user. Used to create Trojaned applications.
`http://www.holodeck.f9.co.uk/elitewrap/`

Firewalk

Tool that allows an attacker to map out a firewall rule set using traceroute-like techniques.
`http://www.packetfactory.net/projects/firewalk/`

IIS Unicode Vulnerability

"Web Server Folder Traversal" Vulnerability
`http://www.microsoft.com/technet/security/bulletin/MS00-078.asp`
"Web Server File Request Parsing" Vulnerability
`http://www.microsoft.com/technet/security/bulletin/MS00-086.asp`
Bugtraq posting
`http://marc.theaimsgroup.com/?l=bugtraq&m=97180137413891&w=2`

lsadump2.exe and LSA

The Local Security Authority (LSA) provides security services for Windows NT. Among other tasks, it authenticates all logon requests, adjudicates users' privileges and determines whether they can access requested resources, and oversees the security auditing functions.
`http://razor.bindview.com/tools/desc/lsadump2_readme.html`

L0phtcrack

`http://www.securitysoftwaretech.com/lc3/faq25.html`

Nessus

Open-source vulnerability scanner. It scans hosts for vulnerabilities and reports on them. This tool is very similar to CyberCop or Internet Scanner.
`http://www.nessus.org/`

Netcat

Home Page for Windows port
`http://www.l0pht.com/~weld/netcat/`

Nmap

Scanning tool used to identify open ports and operating systems on TCP/IP systems. Can also be used to scan ranges of IP addresses to find machines that are alive or listening on certain ports.
`http://www.insecure.org/nmap/index.html`

osql.exe

MS utility used for command line access to database.

PipeUpAdmin.exe

Guardent Advisory
`http://www.guardent.com/A0108022000.html`
Proof-of-concept example exploit source code
`http://marc.theaimsgroup.com/?l=ntbugtraq\&m=96523736401097\&w=2`

Ports 139, 445 and SMB

Windows uses TCP ports 139 and 445 for Server Message Block (SMB) communications. This is the communications mechanism used to provide file and other resource sharing.
The Common Internet File System (CIFS) is the standard way that computer users share files across corporate intranets and the Internet in a Windows network. The CIFS is an enhanced version of the SMB protocol. CIFS is an open, cross-platform implementation of SMB that is currently a draft Internet standard. CIFS was introduced in Service Pack 3 for Windows NT 4.0 and is the native file sharing protocol for Windows 2000. Extensions to CIFS and NetBT (NetBIOS over TCP/IP) now allow connections directly over TCP/IP with the use of TCP port 445.

pwdump2.exe

The Security Account Manager (SAM) database is the storage point for user passwords in Windows NT. The passwords are stored in a hashed form in the database.
`http://razor.bindview.com/tools/desc/pwdump2_readme.html`

Queso

Tool designed to identify operating systems. It does this by sending specially crafted IP packets and analyzing the response to these packets.
`http://www.apostols.org/projectz/queso/`

regdmp.exe

MS command line utility used to dump contents of registry. Provided with the Windows Resource Kit(s). Sample output:
usage: REGDMP [-m \\machinename | -h hivefile hiveroot | -w Windows 95 Directory]
[-i n] [-o outputWidth]
[-s] [-o outputWidth] registryPath
where: -m specifies a remote Windows NT machine whose registry is to be manipulated.
-h specifies a local hive to manipulate.
-w specifies the paths to a Windows 95 system.dat and user.dat files
-i n specifies the display indentation multiple. Default is 4
-o outputWidth specifies how wide the output is to be. By default the outputWidth is set to the width of the console window if standard output has not been redirected to a file. In the latter case, an outputWidth of 240 is used.
-s specifies summary output: value names, type, and first line of data
registryPath specifies where to start dumping.
If REGDMP detects any REG_SZ or REG_EXPAND_SZ that is missing the trailing null character, it will prefix the value string with the following text: (*** MISSING TRAILING NULL CHARACTER ***)
The REGFIND tool can be used to clean up these omissions that are com-

mon programming errors.
In specifying a registry path, either on the command line or in an input file, the following prefix strings can be used:
HKEY_LOCAL_MACHINE
HKEY_USERS
HKEY_CURRENT_USER
USER:
Each of these strings can stand alone as the key name or be followed by a backslash and a subkey path.

SSH Exploit

Exploits vulnerability outlined in CVE-2001-0144. Enables the attacker to run arbitrary code. Exploit returns a root shell to the attacker.
Link to vulnerability information:
`http://razor.bindview.com/publish/advisories/adv_ssh1crc.html`
Link to exploit(s):
`http://www.hack.co.za/index.php?osid=253`

VNC

Home Page
`http://www.uk.research.att.com/vnc/`

Whisker

CGI scanner designed to find vulnerabilities in Web applications.
`http://www.wiretrip.net/rfp/p/doc.asp?id=21&iface=2`

winfo.exe

`http://ntsecurity.nu/toolbox/winfo/`

Appendix B

CAML Grammar

B.1 Introduction

This document describes the grammar for Correlated Attack Modeling Language (CAML). A CAML specification contains a global section, an init section, and a list of module definitions. A module can be viewed as a step in the correlation process. A module definition consists of an activity section, a pre-condition section, and a post-condition section. The activity section is used to specify a list of events needed to trigger the module. These events are specified using event templates, which describes the requirements for candidate event instances. The structure of CAML events is based on the IDMEF data model [11]. The pre-condition section is used to specify constraints on the system states and the event instances. Predicates may be used to specify these constraints. If the activity and the pre-condition sections are met, the inference result specified in the post-condition section will hold. In particular, the module may derive new system states (in the form of predicates) and inferred events.

B.2 Grammar

⟨cam-spec⟩
- ⟨global-sec-opt⟩ ⟨init-sec-opt⟩ ⟨module-list⟩

⟨global-sec-opt⟩
- ⟨global-sec⟩
- ⟨empty⟩

⟨global-sec⟩
- **global (** ⟨const-def-list-opt⟩ ⟨alt-def-list-opt⟩ **)**

⟨const-def-list-opt⟩
- ⟨const-def-list⟩
- ⟨empty⟩

⟨const-def-list⟩
- ⟨const-definition⟩

⟨const-def-list⟩ ⟨const-definition⟩
⟨const-definition⟩
const ⟨type-specifier⟩ ⟨identifier⟩ = ⟨constant⟩
⟨alt-def-list-opt⟩
⟨alt-def-list⟩
⟨empty⟩
⟨alt-def-list⟩
⟨alt-definition⟩
⟨alt-def-list⟩ ⟨alt-definition⟩
⟨alt-definition⟩
alt (nested-ident-list **)**
⟨nested-ident-list⟩
⟨nested-ident⟩
⟨nested-ident-list⟩ **,** ⟨nested-ident⟩
⟨nested-ident⟩
⟨identifier⟩
⟨identifier⟩ **(** ⟨nested-ident⟩ **)**

⟨init-sec-opt⟩
⟨init-sec⟩
⟨empty⟩
⟨init-sec⟩
init (⟨event-or-predicate-list⟩ **)**
⟨event-or-predicate-list⟩
⟨event-or-predicate⟩
⟨event-or-predicate-list⟩ ⟨event-or-predicate⟩
⟨event-or-predicate⟩
⟨event⟩
⟨predicate⟩

⟨module-list⟩
⟨module-definition⟩
⟨module-list⟩ ⟨module-definition⟩
⟨module-definition⟩
module ⟨identifier⟩ **(** ⟨activity-sec-opt⟩ ⟨pre-sec-opt⟩ ⟨post-sec⟩
)

⟨activity-sec-opt⟩
activity: ⟨event-list⟩
⟨empty⟩
⟨pre-sec-opt⟩
pre: ⟨constraint-list⟩
⟨empty⟩
⟨constraint-list⟩
⟨constraint⟩

⟨constraint-list⟩ ⟨constraint⟩
⟨post-sec⟩
post: ⟨event-or-predicate-mod-list⟩
⟨event-or-predicate-mod-list⟩
⟨event-or-predicate-mod⟩
⟨event-or-predicate-mod-list⟩ ⟨event-or-predicate-mod⟩
⟨event-or-predicate-mod⟩
⟨event-mod⟩
⟨predicate-mod⟩
// **add or delete an event/predicate instance,**
// **or modify an existing event/predicate instance**
⟨event-mod⟩
⟨event⟩
- ⟨event-label⟩
***** ⟨event-label⟩ **(** ⟨new-attr-val-list⟩ **)**
⟨predicate-mod⟩
⟨predicate⟩
- ⟨predicate-label⟩
***** ⟨predicate-label⟩ **(** ⟨new-attr-val-list⟩ **)**
⟨new-attr-val-list⟩
⟨new-attr-val⟩
⟨new-attr-val-list⟩ ⟨new-attr-val⟩
⟨new-attr-val⟩
⟨identifier⟩ **<-** ⟨additive-expr⟩
⟨event-label⟩
⟨entity-label⟩
⟨entity-label⟩
⟨identifier⟩
⟨predicate-label⟩
⟨identifier⟩

⟨event-list⟩
⟨labeled-event-definition⟩
⟨event-list⟩ ⟨labeled-event-definition⟩
⟨labeled-event-definition⟩
⟨event-label⟩ **:** ⟨event-definition⟩
⟨event-definition⟩
⟨event-definition⟩
Event (⟨event-attr-list⟩ ⟨analyzer-opt⟩ ⟨source-list-and-or-target-list⟩ ⟨classification-list⟩ ⟨assessment-opt⟩ ⟨correlationalert-opt⟩ ⟨additionaldata-list-opt⟩ **)**
⟨event-attr-list⟩
⟨starttime-opt⟩ ⟨endtime-opt⟩ ⟨thread-id-opt⟩ ⟨alert-count-opt⟩
⟨starttime-opt⟩
⟨starttime⟩

⟨empty⟩
⟨starttime⟩
starttime ⟨comp-op⟩ ⟨time-val⟩
⟨identifier⟩ **: starttime**
⟨endtime-opt⟩
⟨endtime⟩
⟨empty⟩
⟨endtime⟩
endtime ⟨comp-op⟩ ⟨time-val⟩
⟨identifier⟩ **: endtime**
⟨thread-id-opt⟩
⟨thread-id⟩
⟨empty⟩
⟨thread-id⟩
thread-id ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: thread-id**
⟨?identifier⟩
? ⟨identifier⟩
⟨identifier⟩
⟨alert-count-opt⟩
⟨alert-count⟩
⟨empty⟩
⟨alert-count⟩
alert-count ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: alert-count**

⟨analyzer-opt⟩
⟨labeled-analyzer⟩
⟨empty⟩
⟨labeled-analyzer⟩
⟨analyzer⟩
⟨entity-label⟩ **:** ⟨analyzer⟩
⟨analyzer⟩
Analyzer (⟨analyzer-attr-list⟩ ⟨node-opt⟩ ⟨process-opt⟩ **)**
⟨analyzer-attr-list⟩
⟨analyzerid-opt⟩ ⟨manufacturer-opt⟩ ⟨model-opt⟩ ⟨version-opt⟩
⟨class-opt⟩ ⟨ostype-opt⟩ ⟨osversion-opt⟩
⟨analyzerid-opt⟩
⟨analyzerid⟩
⟨empty⟩
⟨analyzerid⟩
analyzerid ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: analyzerid**
⟨manufacturer-opt⟩
⟨manufacturer⟩

⟨empty⟩

⟨manufacturer⟩
  manufacturer ⟨eq-op⟩ ⟨str-val⟩
  ⟨?identifier⟩ **: manufacturer**

⟨model-opt⟩
  ⟨model⟩
  ⟨empty⟩

⟨model⟩
  model ⟨eq-op⟩ ⟨str-val⟩
  ⟨?identifier⟩ **: model**

⟨version-opt⟩
  ⟨version⟩
  ⟨empty⟩

⟨version⟩
  version ⟨comp-op⟩ ⟨str-val⟩
  ⟨?identifier⟩ **: version**

⟨class-opt⟩
  ⟨class⟩
  ⟨empty⟩

⟨class⟩
  class ⟨eq-op⟩ ⟨str-val⟩
  ⟨?identifier⟩ **: class**

⟨ostype-opt⟩
  ⟨ostype⟩
  ⟨empty⟩

⟨ostype⟩
  ostype ⟨eq-op⟩ ⟨str-val⟩
  ⟨?identifier⟩ **: ostype**

⟨osversion-opt⟩
  ⟨osversion⟩
  ⟨empty⟩

⟨osversion⟩
  osversion ⟨comp-op⟩ ⟨str-val⟩
  ⟨?identifier⟩ **: osversion**

⟨source-list-and-or-target-list⟩
  ⟨source-list⟩
  ⟨target-list⟩
  ⟨source-list⟩ ⟨target-list⟩

⟨source-list⟩
  ⟨labeled-source⟩
  ⟨source-list⟩ ⟨labeled-source⟩

⟨labeled-source⟩
  ⟨source⟩
  ⟨entity-label⟩ **:** ⟨source⟩

⟨source⟩
    Source (⟨node-opt⟩ ⟨user-opt⟩ ⟨process-opt⟩ ⟨service-opt⟩ ⟨filelist-opt⟩ **)**

⟨target-list⟩
    ⟨labeled-target⟩
    ⟨target-list⟩ ⟨labeled-target⟩

⟨labeled-target⟩
    ⟨target⟩
    ⟨entity-label⟩ **:** ⟨target⟩

⟨target⟩
    Target (⟨node-opt⟩ ⟨user-opt⟩ ⟨process-opt⟩ ⟨service-opt⟩ ⟨filelist-opt⟩ **)**

⟨node-opt⟩
    ⟨labeled-node⟩
    ⟨empty⟩

⟨labeled-node⟩
    ⟨node⟩
    ⟨entity-label⟩ **:** ⟨node⟩

⟨node⟩
    Node (⟨node-attr-list⟩ ⟨location-opt⟩ ⟨name-or-address⟩ ⟨address-list-opt⟩ **)**

⟨node-attr-list⟩
    ⟨ident-opt⟩ ⟨node-category-opt⟩

⟨ident-opt⟩
    ⟨ident⟩
    ⟨empty⟩

⟨ident⟩
    ident ⟨eq-op⟩ ⟨str-val⟩
    ⟨?identifier⟩ **: ident**

⟨node-category-opt⟩
    ⟨node-category⟩
    ⟨empty⟩

⟨node-category⟩
    category ⟨eq-op⟩ ⟨node-category-enum-val⟩
    ⟨?identifier⟩ **: category**

⟨node-category-enum-val⟩
    ⟨identifier⟩
    ⟨node-category-enum-const⟩

⟨node-category-enum-const⟩
    unknown
    ads
    afs
    coda
    dfs

dns
kerberos
nds
nis
nisplus
nt
wfs
⟨location-opt⟩
⟨location⟩
⟨empty⟩
⟨location⟩
location ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: location**
⟨name-or-address⟩
⟨name⟩
⟨address⟩
⟨empty⟩
⟨name⟩
name ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: name**

⟨address-list-opt⟩
⟨address-list⟩
⟨empty⟩
⟨address-list⟩
⟨labeled-address⟩
⟨address-list⟩ ⟨labeled-address⟩
⟨labeled-address⟩
⟨address⟩
⟨entity-label⟩ **:** ⟨address⟩
⟨address⟩
Address (⟨address-attr-list⟩ ⟨address-clause⟩ ⟨netmask-opt⟩ **)**
⟨address-attr-list⟩
⟨ident-opt⟩ ⟨address-category-opt⟩ ⟨vlan-name-opt⟩ ⟨vlan-num-opt⟩
⟨address-category-opt⟩
⟨address-category⟩
⟨empty⟩
⟨address-category⟩
category ⟨eq-op⟩ ⟨address-category-enum-val⟩
⟨?identifier⟩ **: category**
⟨address-category-enum-val⟩
⟨identifier⟩
⟨address-category-enum-const⟩
⟨address-category-enum-const⟩

unknown
atm
e-mail
lotus-notes
mac
sna
vm
ipv4-addr
ipv4-addr-hex
ipv4-net
ipv4-net-mask
ipv6-addr
ipv6-addr-hex
ipv6-net
ipv6-net-mask

⟨vlan-name-opt⟩
⟨vlan-name⟩
⟨empty⟩

⟨vlan-name⟩
vlan-name ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: vlan-name**

⟨vlan-num-opt⟩
⟨vlan-num⟩
⟨empty⟩

⟨vlan-num⟩
vlan-num ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: vlan-num**

⟨address-clause⟩
address ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: address**

⟨netmask-opt⟩
⟨netmask⟩
⟨empty⟩

⟨netmask⟩
netmask ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: netmask**

⟨user-opt⟩
⟨labeled-user⟩
⟨empty⟩

⟨labeled-user⟩
⟨user⟩
⟨entity-label⟩ **:** ⟨user⟩

⟨user⟩
User (⟨user-attr-list⟩ ⟨userid-list⟩ **)**

⟨user-attr-list⟩
⟨passwd-opt⟩ ⟨hashedpasswd-opt⟩ ⟨ident-opt⟩ ⟨usercat-opt⟩
⟨passwd-opt⟩
⟨passwd⟩
⟨empty⟩
⟨passwd⟩
passwd ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: passwd**
⟨hashedpasswd-opt⟩
⟨hashedpasswd⟩
⟨empty⟩
⟨hashedpasswd⟩
hashedpasswd ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: hashedpasswd**
⟨usercat-opt⟩
⟨usercat⟩
⟨empty⟩
⟨usercat⟩
category ⟨eq-op⟩ ⟨usercat-enum-val⟩
⟨?identifier⟩ **: category**
⟨usercat-enum-val⟩
unknown
application
os-device
⟨userid-list⟩
⟨labeled-userid⟩
⟨userid-list⟩ ⟨labeled-userid⟩
⟨labeled-userid⟩
⟨userid⟩
⟨entity-label⟩ **:** ⟨userid⟩
⟨userid⟩
UserId (⟨userid-attr-list⟩ ⟨name-and-or-number⟩ **)**
⟨userid-attr-list⟩
⟨ident-opt⟩ ⟨userid-type-opt⟩
⟨userid-type-opt⟩
⟨userid-type⟩
⟨empty⟩
⟨userid-type⟩
type ⟨eq-op⟩ ⟨userid-type-enum-val⟩
⟨?identifier⟩ **: type**
⟨userid-type-enum-val⟩
⟨identifier⟩
⟨userid-type-enum-const⟩
⟨userid-type-enum-const⟩
current-user
original-user

target-user
user-privs
current-group
group-privs
⟨name-and-or-number⟩
⟨name⟩
⟨number⟩
⟨name⟩ ⟨number⟩
⟨number⟩
number ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: number**

⟨process-opt⟩
⟨labeled-process⟩
⟨empty⟩
⟨labeled-process⟩
⟨process⟩
⟨entity-label⟩ **:** ⟨process⟩
⟨process⟩
Process (⟨process-attr-list⟩ ⟨name⟩ ⟨pid-opt⟩ ⟨path-opt⟩ ⟨arg-list-opt⟩ ⟨env-list-opt⟩ **)**
⟨process-attr-list⟩
⟨ident-opt⟩
⟨pid-opt⟩
⟨pid⟩
⟨empty⟩
⟨pid⟩
pid ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: pid**
⟨path-opt⟩
⟨path⟩
⟨empty⟩
⟨path⟩
path ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: path**
⟨arg-list-opt⟩
⟨arg-list⟩
⟨empty⟩
⟨arg-list⟩
⟨arg⟩
⟨arg-list⟩ ⟨arg⟩
⟨arg⟩
arg ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: arg**
⟨env-list-opt⟩

⟨env-list⟩
⟨empty⟩
⟨env-list⟩
⟨env⟩
⟨env-list⟩ ⟨env⟩
⟨env⟩
env ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: env**

⟨service-opt⟩
⟨labeled-service⟩
⟨empty⟩
⟨labeled-service⟩
⟨service⟩
⟨entity-label⟩ **:** ⟨service⟩
⟨service⟩
Service (⟨service-attr-list⟩ ⟨patch-list-opt⟩ ⟨userid-opt⟩ ⟨name-port-or-portlist⟩ ⟨protocol-opt⟩ ⟨snmpservice-opt⟩ ⟨webservice-opt⟩ **)**
⟨service-attr-list⟩
⟨patchlevel-opt⟩ ⟨implement-opt⟩ ⟨version-opt⟩ ⟨vendor-opt⟩ ⟨ident-opt⟩
⟨patchlevel-opt⟩
⟨patchlevel⟩
⟨empty⟩
⟨patchlevel⟩
patchlevel ⟨comp-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: patchlevel**
⟨implement-opt⟩
⟨implement⟩
⟨empty⟩
⟨implement⟩
implement ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: implement**
⟨vendor-opt⟩
⟨vendor⟩
⟨empty⟩
⟨vendor⟩
vendor ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: vendor**
⟨patch-list-opt⟩
⟨patch-list⟩
⟨empty⟩
⟨patch-list⟩
patch ⟨eq-op⟩ ⟨list-val⟩

⟨?identifier⟩ **: patch**
⟨userid-opt⟩
⟨userid⟩
⟨empty⟩
⟨name-port-or-portlist⟩
⟨name-and-or-port⟩
⟨portlist⟩
⟨name-and-or-port⟩
⟨name⟩
⟨port⟩
⟨name⟩ ⟨port⟩
⟨port⟩
port ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: port**
⟨portlist⟩
portlist ⟨eq-op⟩ ⟨list-val⟩
⟨?identifier⟩ **: portlist**
⟨protocol-opt⟩
⟨protocol⟩
⟨empty⟩
⟨protocol⟩
protocol ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: protocol**

⟨snmpservice-opt⟩
⟨labeled-snmpservice⟩
⟨empty⟩
⟨labeled-snmpservice⟩
⟨snmpservice⟩
⟨entity-label⟩ **:** ⟨snmpservice⟩
⟨snmpservice⟩
SNMPService (⟨oid-opt⟩ ⟨community-opt⟩ ⟨command-opt⟩ **)**
⟨oid-opt⟩
⟨oid⟩
⟨empty⟩
⟨oid⟩
oid ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: oid**
⟨community-opt⟩
⟨community⟩
⟨empty⟩
⟨community⟩
community ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: community**
⟨command-opt⟩

⟨command⟩
⟨empty⟩
⟨command⟩
command ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: command**

⟨webservice-opt⟩
⟨labeled-webservice⟩
⟨empty⟩
⟨labeled-webservice⟩
⟨webservice⟩
⟨entity-label⟩ **:** ⟨webservice⟩
⟨webservice⟩
WebService (⟨url⟩ ⟨cgi-opt⟩ ⟨http-method-opt⟩ ⟨arg-list-opt⟩
)
⟨url⟩
url ⟨eq-op⟩ ⟨str-val⟩
⟨identifier⟩ **: url**
⟨cgi-opt⟩
⟨cgi⟩
⟨empty⟩
⟨cgi⟩
cgi ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: cgi**
⟨http-method-opt⟩
⟨http-method⟩
⟨empty⟩
⟨http-method⟩
http-method ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: http-method**
⟨arg-list-opt⟩
⟨arg-list⟩
⟨empty⟩
⟨arg-list⟩
⟨arg⟩
⟨arg-list⟩ ⟨arg⟩
⟨arg⟩
arg ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: arg**

⟨filelist-opt⟩
FileList (⟨file-list⟩ **)**
⟨empty⟩
⟨file-list⟩
⟨labeled-file⟩

⟨file-list⟩ ⟨labeled-file⟩

⟨labeled-file⟩

⟨file⟩

⟨entity-label⟩ **:** ⟨file⟩

⟨file⟩

File (⟨file-attr-list⟩ ⟨path⟩ ⟨name⟩ ⟨create-time-opt⟩ ⟨modify-time-opt⟩ ⟨access-time-opt⟩ ⟨data-size-opt⟩ ⟨disk-size-opt⟩ ⟨fileaccess-list-opt⟩ ⟨linkage-opt⟩ ⟨inode-opt⟩ **)**

⟨file-attr-list⟩

⟨ident-opt⟩ ⟨file-category⟩ ⟨fstype⟩

⟨file-category⟩

category ⟨eq-op⟩ ⟨file-category-enum-val⟩

⟨identifier⟩ **: category**

⟨file-category-enum-val⟩

⟨identifier⟩

⟨file-category-enum-const⟩

⟨file-category-enum-const⟩

current

original

⟨fstype⟩

fstype ⟨eq-op⟩ ⟨str-val⟩

⟨identifier⟩ **: fstype**

⟨create-time-opt⟩

⟨create-time⟩

⟨empty⟩

⟨create-time⟩

create-time ⟨comp-op⟩ ⟨time-val⟩

⟨?identifier⟩ **: create-time**

⟨modify-time-opt⟩

⟨modify-time⟩

⟨empty⟩

⟨modify-time⟩

modify-time ⟨comp-op⟩ ⟨time-val⟩

⟨?identifier⟩ **: modify-time**

⟨access-time-opt⟩

⟨access-time⟩

⟨empty⟩

⟨access-time⟩

access-time ⟨comp-op⟩ ⟨time-val⟩

⟨?identifier⟩ **: access-time**

⟨data-size-opt⟩

⟨data-size⟩

⟨empty⟩

⟨data-size⟩

data-size ⟨comp-op⟩ ⟨int-val⟩

⟨?identifier⟩ **: data-size**

⟨disk-size-opt⟩
  ⟨disk-size⟩
  ⟨empty⟩
⟨disk-size⟩
  disk-size ⟨comp-op⟩ ⟨int-val⟩
  ⟨?identifier⟩ **: disk-size**
⟨fileaccess-list-opt⟩
  ⟨fileaccess-list⟩
  ⟨empty⟩
⟨fileaccess-list⟩
  ⟨labeled-fileaccess⟩
  ⟨fileaccess-list⟩ ⟨labeled-fileaccess⟩
⟨labeled-fileaccess⟩
  ⟨fileaccess⟩
  ⟨entity-label⟩ **:** ⟨fileaccess⟩
⟨fileaccess⟩
  FileAccess (⟨userid⟩ ⟨permission-list⟩ **)**
⟨permission-list⟩
  ⟨permission⟩
  ⟨permission-list⟩ ⟨permission⟩
⟨permission⟩
  permission ⟨eq-op⟩ ⟨str-val⟩
  ⟨identifier⟩ **: permission**
⟨linkage-opt⟩
  ⟨labeled-linkage⟩
  ⟨empty⟩
⟨labeled-linkage⟩
  ⟨linkage⟩
  ⟨entity-label⟩ **:** ⟨linkage⟩
⟨linkage⟩
  Linkage (⟨linkage-attr⟩ ⟨name⟩ ⟨path⟩ ⟨labeled-file⟩ **)**
⟨linkage-attr⟩
  ⟨linkage-category⟩
⟨linkage-category⟩
  category ⟨eq-op⟩ ⟨linkage-category-enum-val⟩
  ⟨identifier⟩ **: category**
⟨linkage-category-enum-val⟩
  ⟨identifier⟩
  ⟨linkage-category-enum-const⟩
⟨linkage-category-enum-const⟩
  hard-link
  mount-point
  reparse-point
  shortcut
  stream
  symbolic-link

⟨inode-opt⟩
⟨labeled-inode⟩
⟨empty⟩
⟨labeled-inode⟩
⟨inode⟩
⟨entity-label⟩ **:** ⟨inode⟩
⟨inode⟩
Inode (⟨change-time-opt⟩ ⟨number-major-minor-opt⟩ ⟨c-major-minor-opt⟩ **)**
⟨change-time-opt⟩
⟨change-time⟩
⟨empty⟩
⟨change-time⟩
change-time ⟨comp-op⟩ ⟨time-val⟩
⟨?identifier⟩ **: change-time**
⟨number-major-minor-opt⟩
⟨number-major-minor⟩
⟨empty⟩
⟨number-major-minor⟩
⟨number⟩ ⟨major-device⟩ ⟨minor-device⟩
⟨major-device⟩
major-device ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: major-device**
⟨minor-device⟩
minor-device ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: minor-device**
⟨c-major-minor-opt⟩
⟨c-major-minor⟩
⟨empty⟩
⟨c-major-minor⟩
⟨c-major-device⟩ ⟨c-minor-device⟩
⟨c-major-device⟩
c-major-device ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: c-major-device**
⟨c-minor-device⟩
c-minor-device ⟨eq-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: c-minor-device**

⟨classification-list⟩
⟨labeled-classification⟩
⟨classification-list⟩ ⟨labeled-classification⟩
⟨labeled-classification⟩
⟨classification⟩
⟨entity-label⟩ **:** ⟨classification⟩
⟨classification⟩

Classification (⟨origin⟩ ⟨name⟩ ⟨url⟩ **)**
⟨origin⟩
origin ⟨eq-op⟩ ⟨origin-enum-val⟩
⟨identifier⟩ **: origin**
⟨origin-enum-val⟩
⟨identifier⟩
⟨origin-enum-const⟩
⟨origin-enum-const⟩
unknown
bugtraq
cve
vendor-specific

⟨assessment-opt⟩
⟨labeled-assessment⟩
⟨empty⟩
⟨labeled-assessment⟩
⟨assessment⟩
⟨entity-label⟩ **:** ⟨assessment⟩
⟨assessment⟩
Assessment (⟨impact-opt⟩ ⟨confidence-opt⟩ ⟨anomaly-score-opt⟩ **)**
⟨impact-opt⟩
⟨labeled-impact⟩
⟨empty⟩
⟨labeled-impact⟩
⟨impact⟩
⟨entity-label⟩ **:** ⟨impact⟩
⟨impact⟩
Impact (⟨severity-opt⟩ ⟨completion-opt⟩ ⟨impact-type-opt⟩ **)**
⟨severity-opt⟩
⟨severity⟩
⟨empty⟩
⟨severity⟩
severity ⟨comp-op⟩ ⟨severity-enum-val⟩
⟨?identifier⟩ **: severity**
⟨severity-enum-val⟩
⟨identifier⟩
⟨severity-enum-const⟩
⟨severity-enum-const⟩
low
medium
high
⟨completion-opt⟩
⟨completion⟩

  ⟨empty⟩
⟨completion⟩
  completion ⟨eq-op⟩ ⟨completion-enum-val⟩
  ⟨?identifier⟩ **: completion**
⟨completion-enum-val⟩
  ⟨identifier⟩
  ⟨completion-enum-const⟩
⟨completion-enum-const⟩
  failed
  succeeded
⟨impact-type-opt⟩
  ⟨impact-type⟩
  ⟨empty⟩
⟨impact-type⟩
  type ⟨eq-op⟩ ⟨impact-type-enum-val⟩
  ⟨?identifier⟩ **: type**
⟨impact-type-enum-val⟩
  ⟨identifier⟩
  ⟨impact-type-enum-const⟩
⟨impact-type-enum-const⟩
  admin
  dos
  file
  recon
  user
  other
⟨confidence-opt⟩
  ⟨labeled-confidence⟩
  ⟨empty⟩
⟨labeled-confidence⟩
  ⟨confidence⟩
  ⟨entity-label⟩ **:** ⟨confidence⟩
⟨confidence⟩
  Confidence (⟨confidence-rating⟩ ⟨confidence-numeric-opt⟩ **)**
⟨confidence-rating⟩
  rating ⟨comp-op⟩ ⟨confidence-rating-enum-val⟩
  ⟨identifier⟩ **: rating**
⟨confidence-rating-enum-val⟩
  ⟨identifier⟩
  ⟨confidence-rating-enum-const⟩
⟨confidence-rating-enum-const⟩
  low
  medium
  high
  numeric
⟨confidence-numeric-opt⟩

⟨confidence-numeric⟩
⟨empty⟩
⟨confidence-numeric⟩
val ⟨comp-op⟩ ⟨confidence-numeric-val⟩
⟨?identifier⟩ **: val**
⟨confidence-numeric-val⟩
⟨identifier⟩
⟨float-constant⟩
⟨anomaly-score-opt⟩
⟨anomaly-score⟩
⟨empty⟩
⟨anomaly-score⟩
anomaly-score ⟨comp-op⟩ ⟨anomaly-score-val⟩
⟨?identifier⟩ **: anomaly-score**
⟨anomaly-score-val⟩
⟨float-val⟩

⟨correlationalert-opt⟩
⟨labeled-correlationalert⟩
⟨empty⟩
⟨labeled-correlationalert⟩
⟨correlationalert⟩
⟨entity-label⟩ **:** ⟨correlationalert⟩
⟨correlationalert⟩
Correlation (⟨correlation-attr-list-opt⟩ ⟨name⟩ ⟨alertident-list⟩
)
⟨correlation-attr-list-opt⟩
⟨correlation-attr-list⟩ ⟨empty⟩
⟨correlation-attr-list⟩
⟨alert-priority-opt⟩ ⟨alert-relevance-opt⟩ ⟨alert-rank-opt⟩ ⟨activity-measure-opt⟩ ⟨merge-policy-opt⟩ ⟨merge-count-opt⟩ ⟨correlated-alert-outcome-opt⟩ ⟨model-confidence-min-opt⟩ ⟨model-confidence-max-opt⟩ ⟨model-confidence-avg-opt⟩ ⟨model-confidence-std-opt⟩ ⟨anomaly-score-min-opt⟩ ⟨anomaly-score-max-opt⟩ ⟨anomaly-score-avg-opt⟩ ⟨anomaly-score-std-opt⟩ ⟨priority-min-opt⟩ ⟨priority-max-opt⟩ ⟨priority-avg-opt⟩ ⟨priority-std-opt⟩ ⟨relevance-min-opt⟩ ⟨relevance-max-opt⟩ ⟨relevance-avg-opt⟩ ⟨relevance-std-opt⟩ ⟨rank-min-opt⟩ ⟨rank-max-opt⟩ ⟨rank-avg-opt⟩ ⟨rank-std-opt⟩
⟨alert-priority-opt⟩
⟨alert-priority⟩
⟨empty⟩
⟨alert-priority⟩
alert-priority ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: alert-priority**
⟨alert-relevance-opt⟩

⟨alert-relevance⟩
⟨empty⟩
⟨alert-relevance⟩
alert-relevance ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: alert-relevance**
⟨alert-rank-opt⟩
⟨alert-rank⟩
⟨empty⟩
⟨alert-rank⟩
alert-rank ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: alert-rank**
⟨activity-measure-opt⟩
⟨activity-measure⟩
⟨empty⟩
⟨activity-measure⟩
activity-measure ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: activity-measure**
⟨merge-policy-opt⟩
⟨merge-policy⟩
⟨empty⟩
⟨merge-policy⟩
merge-policy ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: merge-policy**
⟨merge-count-opt⟩
⟨merge-count⟩
⟨empty⟩
⟨merge-count⟩
merge-count ⟨comp-op⟩ ⟨int-val⟩
⟨?identifier⟩ **: merge-count**
⟨correlated-alert-outcome-opt⟩
⟨correlated-alert-outcome⟩
⟨empty⟩
⟨correlated-alert-outcome⟩
correlated-alert-outcome ⟨eq-op⟩ ⟨completion-enum-val⟩
⟨?identifier⟩ **: correlated-alert-outcome**
⟨model-confidence-min-opt⟩
⟨model-confidence-min⟩
⟨empty⟩
⟨model-confidence-min⟩
model-confidence-min ⟨comp-op⟩ ⟨confidence-numeric-val⟩
⟨?identifier⟩ **: model-confidence-min**
⟨model-confidence-max-opt⟩
⟨model-confidence-max⟩
⟨empty⟩
⟨model-confidence-max⟩
model-confidence-max ⟨comp-op⟩ ⟨confidence-numeric-val⟩

⟨?identifier⟩ **: model-confidence-max**
⟨model-confidence-avg-opt⟩
⟨model-confidence-avg⟩
⟨empty⟩
⟨model-confidence-avg⟩
model-confidence-avg ⟨comp-op⟩ ⟨confidence-numeric-val⟩
⟨?identifier⟩ **: model-confidence-avg**
⟨model-confidence-std-opt⟩
⟨model-confidence-std⟩
⟨empty⟩
⟨model-confidence-std⟩
model-confidence-std ⟨comp-op⟩ ⟨confidence-std-val⟩
⟨?identifier⟩ **: model-confidence-std**
⟨confidence-std-val⟩
⟨identifier⟩
⟨float-constant⟩
⟨anomaly-score-min⟩
anomaly-score-min ⟨comp-op⟩ ⟨anomaly-score-val⟩
⟨?identifier⟩ **: anomaly-score-min**
⟨anomaly-score-max-opt⟩
⟨anomaly-score-max⟩
⟨empty⟩
⟨anomaly-score-max⟩
anomaly-score-max ⟨comp-op⟩ ⟨anomaly-score-val⟩
⟨?identifier⟩ **: anomaly-score-max**
⟨anomaly-score-avg-opt⟩
⟨anomaly-score-avg⟩
⟨empty⟩
⟨anomaly-score-avg⟩
anomaly-score-avg ⟨comp-op⟩ ⟨anomaly-score-val⟩
⟨?identifier⟩ **: anomaly-score-avg**
⟨anomaly-score-std-opt⟩
⟨anomaly-score-std⟩
⟨empty⟩
⟨anomaly-score-std⟩
anomaly-score-std ⟨comp-op⟩ ⟨anomaly-score-std-val⟩
⟨?identifier⟩ **: anomaly-score-std**
⟨anomaly-score-std-val⟩
⟨identifier⟩
⟨float-constant⟩
⟨priority-min-opt⟩
⟨priority-min⟩
⟨empty⟩
⟨priority-min⟩
priority-min ⟨comp-op⟩ ⟨priority-val⟩
⟨?identifier⟩ **: priority-min**

⟨priority-val⟩
    ⟨float-val⟩
⟨priority-max-opt⟩
    ⟨priority-max⟩
    ⟨empty⟩
⟨priority-max⟩
    priority-max ⟨comp-op⟩ ⟨priority-val⟩
    ⟨?identifier⟩ **: priority-max**
⟨priority-avg-opt⟩
    ⟨priority-avg⟩
    ⟨empty⟩
⟨priority-avg⟩
    priority-avg ⟨comp-op⟩ ⟨priority-val⟩
    ⟨identifier⟩ **: priority-avg**
⟨priority-std-opt⟩
    ⟨priority-std⟩
    ⟨empty⟩
⟨priority-std⟩
    priority-std ⟨comp-op⟩ ⟨priority-std-val⟩
    ⟨?identifier⟩ **: priority-std**
⟨priority-std-val⟩
    ⟨float-val⟩
⟨relevance-min-opt⟩
    ⟨relevance-min⟩
    ⟨empty⟩
⟨relevance-min⟩
    relevance-min ⟨comp-op⟩ ⟨relevance-val⟩
    ⟨?identifier⟩ **: relevance-min**
⟨relevance-val⟩
    ⟨float-val⟩
⟨relevance-max-opt⟩
    ⟨relevance-max⟩
    ⟨empty⟩
⟨relevance-max⟩
    relevance-max ⟨comp-op⟩ ⟨relevance-val⟩
    ⟨?identifier⟩ **: relevance-max**
⟨relevance-avg-opt⟩
    ⟨relevance-avg⟩
    ⟨empty⟩
⟨relevance-avg⟩
    relevance-avg ⟨comp-op⟩ ⟨relevance-val⟩
    ⟨?identifier⟩ **: relevance-avg**
⟨relevance-std-opt⟩
    ⟨relevance-std⟩
    ⟨empty⟩
⟨relevance-std⟩

relevance-std ⟨comp-op⟩ ⟨relevance-std-val⟩
⟨?identifier⟩ **: relevance-std**
⟨relevance-std-val⟩
⟨float-val⟩
⟨rank-min-opt⟩
⟨rank-min⟩
⟨empty⟩
⟨rank-min⟩
rank-min ⟨comp-op⟩ ⟨rank-val⟩
⟨?identifier⟩ **: rank-min**
⟨rank-val⟩
⟨float-val⟩
⟨rank-max-opt⟩
⟨rank-max⟩
⟨empty⟩
⟨rank-max⟩
rank-max ⟨comp-op⟩ ⟨rank-val⟩
⟨?identifier⟩ **: rank-max**
⟨rank-avg-opt⟩
⟨rank-avg⟩
⟨empty⟩
⟨rank-avg⟩
rank-avg ⟨comp-op⟩ ⟨rank-val⟩
⟨?identifier⟩ **: rank-avg**
⟨rank-std-opt⟩
⟨rank-std⟩
⟨empty⟩
⟨rank-std⟩
rank-std ⟨comp-op⟩ ⟨rank-std-val⟩
⟨?identifier⟩ **: rank-std**
⟨rank-std-val⟩
⟨float-val⟩
⟨alertident-list⟩
⟨labeled-alertident⟩
⟨alertident-list⟩ ⟨labeled-alertident⟩
⟨labeled-alertident⟩
⟨alertident⟩
⟨entity-label⟩ **:** ⟨alertident⟩
⟨alertident⟩
Alertident (⟨alertident-attr-list⟩ ⟨alertident-val⟩ **)**
⟨alertident-attr-list⟩
⟨alertident-analyzerid-opt⟩
⟨alertident-val⟩
val ⟨eq-op⟩ ⟨str-val⟩
⟨identifier⟩ **: val**
⟨alertident-analyzerid-opt⟩

⟨alertident-analyzerid⟩
⟨empty⟩
⟨alertident-analyzerid⟩
analyzerid ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: analyzerid**

⟨additionaldata-list-opt⟩
⟨additionaldata-list⟩
⟨empty⟩
⟨additionaldata-list⟩
⟨labeled-additionaldata⟩
⟨additionaldata-list⟩ ⟨labeled-additionaldata⟩
⟨labeled-additionaldata⟩
⟨additionaldata⟩
⟨entity-label⟩ **:** ⟨additionaldata⟩
⟨additionaldata⟩
AdditionalData (⟨additionaldata-type⟩ ⟨meaning-opt⟩ ⟨additionaldata-val⟩ **)**
⟨additionaldata-type⟩
type ⟨eq-op⟩ ⟨additionaldata-type-val⟩
⟨identifier⟩ **: type**
⟨additionaldata-type-val⟩
boolean
byte
char
datetime
int
ntpstamp
portlist
float
string
⟨meaning-opt⟩
⟨meaning⟩
⟨emtpy⟩
⟨meaning⟩
meaning ⟨eq-op⟩ ⟨str-val⟩
⟨?identifier⟩ **: meaning**
⟨additionaldata-val⟩
val ⟨comp-op⟩ ⟨str-val⟩
⟨identifier⟩ **: val**

⟨labeled-predicate⟩
⟨predicate-label⟩ **:** ⟨predicate⟩
⟨predicate⟩
⟨predicate⟩

⟨host-based-predicate⟩
⟨net-based-predicate⟩
⟨know-predicate⟩
⟨temporal-predicate⟩

⟨host-based-predicate⟩
⟨host-predicate-identifier⟩ **(** ⟨predicate-attr-list⟩ ⟨node⟩ ⟨service-opt⟩ ⟨host-source-list-opt⟩ ⟨host-target-list-opt⟩ **)**
// if (host-predicate-identifier is not ⟨identifier⟩) **name** *is mandatory*

⟨host-predicate-identifier⟩
⟨identifier⟩
HostBasedPredicate

⟨predicate-attr-list⟩
⟨name-opt⟩ ⟨predicate-val-opt⟩ ⟨optype-opt⟩ ⟨starttime-opt⟩ ⟨endtime-opt⟩

⟨name-opt⟩
⟨name⟩
⟨empty⟩

⟨predicate-val-opt⟩
⟨predicate-val⟩
⟨empty⟩

⟨predicate-val⟩
val ⟨eq-op⟩ ⟨str-val⟩
⟨identifier⟩ **: val**

⟨optype-opt⟩
⟨optype⟩
⟨empty⟩

⟨optype⟩
optype ⟨eq-op⟩ ⟨str-val⟩
⟨identifier⟩ **: optype**

⟨host-source-list-opt⟩
⟨host-source-list⟩
⟨empty⟩

⟨host-target-list-opt⟩
⟨host-target-list⟩
⟨empty⟩

⟨host-source-list⟩
⟨labeled-host-source⟩
⟨host-source-list⟩ ⟨labeled-host-source⟩

⟨host-target-list⟩
⟨labeled-host-target⟩
⟨host-target-list⟩ ⟨labeled-host-target⟩

⟨labeled-host-source⟩
⟨host-source⟩

⟨entity-label⟩ **:** ⟨host-source⟩
⟨host-source⟩
Source (⟨user-opt⟩ ⟨process-opt⟩ ⟨filelist-opt⟩ **)**
⟨labeled-host-target⟩
⟨host-target⟩
⟨entity-label⟩ **:** ⟨host-target⟩
⟨host-target⟩
Target (⟨user-opt⟩ ⟨process-opt⟩ ⟨filelist-opt⟩ **)**

⟨net-based-predicate⟩
⟨net-predicate-identifier⟩ **(** ⟨predicate-attr-list⟩ ⟨source-list-and-or-target-list⟩ **)**
⟨net-predicate-identifier⟩
⟨identifier⟩
NetBasedPredicate

⟨know-predicate⟩
Know (⟨predicate-attr-list⟩ ⟨node⟩ ⟨userid⟩ ⟨predicate⟩ **)**

⟨temporal-predicate⟩
⟨temporal-predicate-identifier⟩ **(** ⟨object-label⟩ **,** ⟨object-label⟩ **)**
⟨object-label⟩
⟨predicate-label⟩
⟨event-label⟩
⟨time-interval-label⟩
⟨time-interval-label⟩
⟨identifier⟩

⟨constraint⟩
⟨cond-expr⟩
⟨cond-expr⟩
⟨or-expr⟩
⟨or-expr⟩
⟨and-expr⟩
⟨or-expr⟩ || ⟨and-expr⟩
⟨and-expr⟩
⟨equality-expr⟩
⟨and-expr⟩ **,** ⟨equality-expr⟩
⟨equality-expr⟩
⟨membership-expr⟩
⟨equality-expr⟩ ⟨eq-op⟩ ⟨membership-expr⟩
⟨membership-expr⟩
⟨relational-expr⟩
⟨membership-expr⟩ ⟨member-op⟩ ⟨relational-expr⟩
⟨relational-expr⟩

⟨additive-expr⟩
⟨relational-expr⟩ ⟨relational-op⟩ ⟨additive-expr⟩
⟨additive-expr⟩
⟨multiplicative-expr⟩
⟨additive-expr⟩ **+** ⟨multiplicative-expr⟩
⟨additive-expr⟩ **-** ⟨multiplicative-expr⟩
⟨multiplicative-expr⟩
⟨unary-expr⟩
⟨multiplicative-expr⟩ ***** ⟨unary-expr⟩
⟨multiplicative-expr⟩ **/** ⟨unary-expr⟩
⟨unary-expr⟩
⟨postfix-expr⟩
⟨unary-op⟩ ⟨unary-expr⟩
⟨postfix-expr⟩
⟨primary-expr⟩
⟨postfix-expr⟩ **.** ⟨identifier⟩
⟨postfix-expr⟩ **(** ⟨cond-expr-list-opt⟩ **)**
⟨primary-expr⟩
⟨identifier⟩
⟨constant⟩
⟨labeled-predicate⟩
(⟨cond-expr⟩ **)**
⟨labeled-time-interval-expr⟩
⟨cond-expr-list-opt⟩
⟨cond-expr-list⟩
⟨empty⟩
⟨cond-expr-list⟩
⟨cond-expr⟩
⟨cond-expr-list⟩ **,** ⟨cond-expr⟩
⟨labeled-time-interval-expr⟩
⟨time-interval-label⟩ **:** ⟨time-interval-op⟩ **(** ⟨object-label⟩ **,** ⟨object-label⟩ **)**
⟨time-interval-op⟩
ExtUnion
Union
Intersection
StartToStart
EndToEnd
Gap

⟨type-specifier⟩
int
float
boolean
char

string
byte
portlist
⟨enum-type-specifier⟩
datetime
ntpstamp
time
timerange
ipv4-addr
ipv4-addr-hex
ipv4-net
ipv4-net-mask
ipv4
ipv4list
atm
e-mail
lotus-notes
mac
sna
vm
ipv6-addr
ipv6-addr-hex
ipv6-net
ipv6-net-mask
ipv6
ipv6list
address

⟨enum-type-specifier⟩
node-category
address-category
userid-type
origin
severity
completion
impact-type
confidence-rating
linkage-category
file-category

⟨constant⟩
⟨int-constant⟩
⟨float-constant⟩
⟨char-constant⟩
⟨string-constant⟩
⟨byte-constant⟩
⟨list-constant⟩
⟨enum-constant⟩

⟨time-constant⟩
⟨time-range-constant⟩
⟨address-constant⟩
⟨enum-constant⟩
⟨node-category-enum-const⟩
⟨address-category-enum-const⟩
⟨userid-type-enum-const⟩
⟨origin-enum-const⟩
⟨severity-enum-const⟩
⟨completion-enum-const⟩
⟨impact-type-enum-const⟩
⟨confidence-rating-enum-const⟩
⟨linkage-category-enum-const⟩
⟨file-category-enum-const⟩
⟨int-val⟩
⟨identifier⟩
⟨int-constant⟩
⟨float-val⟩
⟨identifier⟩
⟨float-constant⟩
⟨str-val⟩
⟨identifier⟩
⟨string-constant⟩
⟨list-op⟩
⟨eq-op⟩
⟨member-op⟩
⟨member-op⟩
in
notin
⟨list-val⟩
⟨identifier⟩
⟨list-constant⟩
⟨list-constant⟩
[⟨list-constant2⟩]
[ ]
⟨list-constant2⟩
⟨list-item⟩
⟨list-constant2⟩ **,** ⟨list-item⟩
⟨list-item⟩
⟨string-constant⟩
⟨int-constant⟩
⟨int-range⟩
⟨int-range⟩
⟨int-constant⟩ **-** ⟨int-constant⟩
⟨time-val⟩
⟨identifier⟩

  - ⟨time-const⟩

- ⟨eq-op⟩
  - **==**
  - **!=**
- ⟨relational-op⟩
  - >
  - >=
  - <
  - <=
- ⟨comp-op⟩
  - ⟨eq-op⟩
  - ⟨relational-op⟩
- ⟨unary-op⟩
  - **+**
  - **-**
  - **!**

Appendix C

CAML Predicates

C.1 Introduction

This document describes the predicates used in CAML (Correlated Attack Modeling Language). For each predicate class, the following information is provided: the parent predicate class, a description, the arguments that are applicable, the mandatory arguments, and an example that illustrates this predicate class. The predicate classes are partitioned into several sections depending on whether they pertain to services, files, users, hosts, knowledge of an entity, or a temporal property between two time intervals.

C.2 Services: Operating Systems and Applications

C.2.1 HasOS

Parent: HostBasedPredicate
Description: A host runs a specified OS.
Arguments: Node(Address), Service
Mandatory: Node(Address(address)), Service(implement)

```
Example:
  HasOS(
    Node(Address(address == hostaddr))
    Service(
      implement == "Windows"
      version == "2000"))
```

C.2.2 HasService

Parent: HostBasedPredicate
Description: A service is provided by a host. The privileges of the server pro-

cess is specified in Service(UserId), and the client's requests are executed with privileges specified in Source(User)
Arguments: Node(Address), Service(UserId), Source(User(UserId))
Mandatory: Node(Address(address)), Service(name)

```
Example:
  HasService(
    Node(Address(address == hostaddr))
    Service(
      implement == "Apache"
      version == "1.3.20"
      UserId(
        type == "user-privs"
        number == "root")
      name == "http"
      port == 80)
    Source(User(UserId(number == webuser))))
```

C.2.3 WebRoot

Parent: HostBasedPredicate
Description: A specified path is the root directory of the specified web server.
Arguments: Node(Address), Service, Source(FileList(File))
Mandatory: Node(Address(address)), Source(FileList(File(path)))

```
Example:
  WebRoot(
    Node(Address(address == hostaddr))
    Service(
      implement == "iPlanet"
      version == "4.1"
      name == "http"
      port == servport)
    Source(FileList(File(path == webrootdir))))
```

C.2.4 WebExecutableDir

Parent: HostBasedPredicate
Description: A specified directory under the "web" directory is executable. When a web server receives a request whose parent directory is executable, the web server will attempt to execute the file instead of downloading it.
Arguments: Node(Address), Service, Source(FileList(File))
Mandatory: Node(Address(address)), Source(FileList(File(path)))

```
Example:
  WebExecutableDir(
    Node(Address(address == hostaddr))
```

```
    Service(
      name == "http"
      port == servport)
    Source(FileList(File(path == scriptdir))))
```

C.2.5 HasPatch

Parent: HostBasedPredicate
Description: A patch identified by a patch level or by a patch id has been applied to a specified service.
Arguments: Node(Address), Service
Mandatory: Node(Address(address)), Service(patch, name)

```
Example:
  HasPatch(
    Node(Address(address == hostaddr))
    Service(
      patch == patchid
      name == servname
      port == servport))
```

C.2.6 CorruptServerConfig

Parent: NetBasedPredicate
Description: A specified user has corrupted the configuration setup of a specified server.
Arguments(Source): Node(Address), User(UserId)
Arguments(Target): Node(Address), Service
Mandatory: Target(Node(Address(address)), Service(name))

```
Example:
  CorruptServerConfig(
    Source(
      Node(Address(address == h1))
      User(UserId(number == u)))
    Target(
      Node(Address(address == h2))
      Service(
        name == "http"
        port == webservport)))
```

C.2.7 CorruptServerContent

Parent: NetBasedPredicate
Description: A specified user has corrupted the data content managed by a specified server.
Arguments(Source): Node(Address), User(UserId)

Arguments(Target): Node(Address), Service
Mandatory: Target(Node(Address(address)), Service(name))

```
Example:
  CorruptServerContent(
    Source(
      Node(Address(address == h1))
      User(UserId(number == u)))
    Target(
      Node(Address(address == h2))
      Service(
        name == "http"
        port == webservport)))
```

C.2.8 Depends

Parent: NetBasedPredicate
Description: A service (implementation) depends on another service (implementation).
Arguments(Source): Node(Address), Service
Arguments(Target): Node(Address), Service
Mandatory: Source(Service), Target(Service)

```
Example:
  Depends(
    Source(
      Node(Address(address == "1.2.3.4"))
      Service(implement == "Apache")
    Target(Service(
      implement == "OpenSSL"
      version == "0.9")))
```

C.2.9 SmallInputLargeOutput

Parent: HostBasedPredicate
Description: A specified service has the property that it takes a small input and generates a large output.
Arguments: Node(Address), Service
Mandatory: Node(Address(address)), Service(name $\vee$ port)

```
Example:
  SmallInputLargeOutput(
    Node(Address(address == dnsserver))
    Service(name == "domain"))
```

C.3 Files

C.3.1 HasFile

Parent: HostBasedPredicate
Description: A host has a specified file.
Arguments: Node(Address), Source(FileList(File))
Mandatory: Node(Address(address)), Source(FileList(File(path, name)))

```
Example:
  HasFile(
    Node(Address(address == hostaddr))
    Source(FileList(File(
      path == "/etc/"
      name == "shadow"))))
```

C.3.2 FileEq

Parent: NetBasedPredicate
Description: Two files are the same (e.g., one is a copy of the other).
Arguments(Source): Node(Address), FileList(File)
Arguments(Target): Node(Address), FileList(File)
Mandatory: Source(Node(Address(address)), FileList(File(path, name))), Target(Node(Address(address)), FileList(File(path, name)))

```
Example:
  FileEq(
    Source(
      Node(Address(address == h1))
      FileList(File(
        path == file1-path
        name == file1-name)))
    Target(
      Node(Address(address == h2))
      FileList(File(
        path == file2-path
        name == file2-name))))
```

C.3.3 HasSpecialFile

Parent: HostBasedPredicate
Description: A file is a copy of another special file (e.g., Windows' cmd.exe).
Arguments: Node(Address), Source(FileList(File)), Target(FileList(File))
Mandatory: Node(Address(address)), Source(FileList(File(path, name))), Target(FileList(File(name)))

```
Example:
  HasSpecialFile(
```

```
    Node(Address(address == hostaddr))
    Source(FileList(File(
      path == cmdexepath
      name == cmdexename)))
    Target(FileList(File(
      name == "cmd.exe"))))
```

C.3.4 FilePerm

Parent: HostBasedPredicate
Description: A user has a specified file access right to a file.
Arguments: Node(Address), Source(FileList)
Mandatory: Node(Address), Source(FileList(File(path, name, FileAccess(UserId(type, number), permission))))

```
Example:
  FilePerm(
    Node(Address(address == hostaddr))
    Source(FileList(File(
      category == "current"
      fstype == "nfs"
      path == fpath
      name == fname
      FileAccess(
        UserId(
          type == "user-privs"
          number == uid)
        permission == "read")))))
```

C.3.5 CorruptedFile

Parent: HostBasedPredicate
Description: A specified file may be corrupted by an attack.
Arguments: Node(Address), Source(FileList(File))
Mandatory: Node(Address(address)), Source(FileList(File(path, name)))

```
Example:
  CorruptedFile(
    Node(Address(address == hostaddr))
    Source(FileList(File(
      path == "/etc/"
      name == "password"))))
```

C.4 Users

C.4.1 HasUser

Parent: HostBasedPredicate
Description: A host has a specified user account.
Arguments: Node(Address), Source(User(UserId))
Mandatory: Node(Address(address)), Source(UserId(number))

```
Example:
  HasUser(
    Node(Address(address == hostaddr))
    Source(
      User(UserId(number == uid))))
```

C.4.2 HasAppUser

Parent: HostBasedPredicate
Description: A host has a specified user account for a specified application. Note that OS-level user id's are used in HasUser and application-level user id's are used in HasAppUser.
Arguments: Node(Address), Service, Source(User(UserId))
Mandatory: Node(Address(address)), Service(name), Source(User(UserId(number)))

```
Example:
  HasAppUser(
    Node(Address(address == hostaddr))
    Service(
      name == "sql"
      port == dbservport)
    Source(User(
      passwd == ""
      category == "application"
      UserId(number == sa))))
```

C.4.3 SuspiciousUser

Parent: HostBasedPredicate
Description: A specified user account has exhibited suspicious behaviour.
Arguments: Node(Address), Source(User(UserId))
Mandatory: Node(Address(address)), Source(UserId(number))

```
Example:
  SuspiciousUser(
    Node(Address(address == hostaddr))
    Source(
      User(UserId(number == uid))))
```

C.4.4 AdminPriv

Parent: NetBasedPredicate
Description: A user has administrative privileges, e.g., Unix's "root", at a specified host.
Arguments(Source): Node(Address), User(UserId)
Arguments(Target): Node(Address)
Mandatory: Source(Node(Address(address)), User(UserId(number))), Target(Node(Address(address)))

```
Example:
  AdminPriv(
    Source(
      Node(Address(address == h1))
      User(UserId(number == uid)))
    Target(
      Node(Address(address == h2))))
```

C.4.5 UserPasswd

Parent: HostBasedPredicate
Description: A user has a specified password and hashed password.
Arguments: Node(Address), Source(User(UserId))
Mandatory: Node(Address(address)), Source(passwd, User(UserId(number)))

```
Example:
  UserPasswd(
    Node(Address(address == hostaddr))
    Source(
      passwd == x
      hashedpasswd == y
      User(UserId(number == uid))))
```

C.4.6 UserHashedPasswd

Parent: HostBasedPredicate
Description: A user has a specified hashed password.
Arguments: Node(Address), Source(User(UserId))
Mandatory: Node(Address(address)), Source(hashedpasswd, User(UserId(number)))

```
Example:
  UserHashedPasswd(
    Node(Address(address == hostaddr))
    Source(
      hashedpasswd == y
      User(UserId(number == uid))))
```

C.4.7 SwitchUser

Parent: NetBasedPredicate
Description: User u1 at node saddr can become user u2 at node taddr.
Arguments(Source): Node(Address), User(UserId)
Arguments(Target): Node(Address), User(UserId)
Mandatory: Source(Node(Address(address)), User(UserId(number))),
Target(Node(Address(address)), User(UserId(number)))

```
Example:
  SwitchUser(
    Source(Node(Address(address == saddr))
      User(UserId(number == u1)))
    Target(Node(Address(address == taddr))
      User(UserId(number == u2))))
```

C.5 Hosts

C.5.1 SuspiciousHost

Parent: HostBasedPredicate
Description: Suspicious activities originated from host hostaddr have been detected.
Arguments: Node(Address)
Mandatory: Node(Address(address))

```
Example:
  SuspiciousHost(
    Node(Address(address == hostaddr)))
```

C.5.2 InDomain

Parent: HostBasedPredicate
Description: A host hostaddr belongs to a specified domain.
Arguments: Node(Address)
Mandatory: val, Node(Address(address))

```
Example:
  InDomain(
    val == "Internal"
    Node(Address(address == hostaddr)))
```

C.5.3 BiComm

Parent: NetBasedPredicate
Description: Bi-directional communication channels possible between two network end points.

Arguments(Source): Node(Address), Service
Arguments(Target): Node(Address), Service
Mandatory: Source(Node(Address(address)), Service(port)),
Target(Node(Address(address)), Service(port))

```
Example:
  BiComm(
    Source(
      Node(Address(address == srcaddr))
      Service(port == srcport))
    Target(
      Node(Address(address == dstaddr))
      Service(port == dstport)))
```

C.5.4 HostsSameSecurityDomain

Parent: NetBasedPredicate
Description: Nodes at saddr and taddr belong to the same security domain.
e.g., Nodes under the same administrative control.
Arguments(Source): Node(Address)
Arguments(Target): Node(Address)
Mandatory: Source(Node(Address(address))), Target(Node(Address(address)))

```
Example:
  HostsSameSecurityDomain(
    Source(Node(Address(address == saddr)))
    Target(Node(Address(address == taddr))))
```

C.6 Know

C.6.1 Know

Parent: Predicate
Description: User u at host h knows a specified predicate instance.
Arguments: Node(Address), UserId, Predicate
Mandatory: Node(Address(address)), UserId(number), Predicate

```
Example:
  Know(
    Node(Address(address == h))
    UserId(number == u)
    HasPasswd(
      Node(Address(address == t))
      Source(User(
        passwd == "terces"
        UserId(number == tuid)))))
```

C.7 Temporal

C.7.1 IsBefore

Parent: Predicate
Description: Time interval r1 ends strictly before time interval r2 starts.
Arguments: object label (i.e., predicate/event/time interval label), object label
Mandatory: object label, object label

```
Example:
  IsBefore(r1,r2)
```

C.7.2 IsAfter

Parent: Predicate
Description: r1 begins strictly after r2 starts.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  IsAfter(r1,r2)
```

C.7.3 IsEqual

Parent: Predicate
Description: r1 and r2 start and end at the same times.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  IsEqual(r1,r2)
```

C.7.4 IsMet

Parent: Predicate
Description: The end time of r1 is the same as the start time of r2.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  IsMet(r1,r2)
```

C.7.5 IsMetBy

Parent: Predicate
Description: The start time of r1 is the same as the end time of r2.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  IsMetBy(r1,r2)
```

C.7.6 OverlapWith

Parent: Predicate
Description: The start time of r2 is strictly inside r1, and the end time of r2 is not in r1.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  OverlapWith(r1,r2)
```

C.7.7 OverlapBy

Parent: Predicate
Description: The start time of r1 is strictly inside r2, and the end time of r1 is not in r2.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  OverlapBy(r1,r2)
```

C.7.8 StartsWith

Parent: Predicate
Description: r1 and r2 have the same start time, and r1 ends strictly before r2 ends.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  StartWith(r1,r2)
```

C.7.9 StartsBy

Parent: Predicate
Description: r1 and r2 have the same start time, and r1 ends strictly after r2 ends.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  StartBy(r1,r2)
```

C.7.10 IsDuring

Parent: Predicate
Description: r1 starts and ends strictly inside r2.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  IsDuring(r1,r2)
```

C.7.11 IsContained

Parent: Predicate
Description: r2 starts and ends strictly inside r1.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  IsContained(r1,r2)
```

C.7.12 EndsWith

Parent: Predicate
Description: r1 and r2 have the same end time, and r1 starts strictly after r2 starts.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  EndsWith(r1,r2)
```

C.7.13 EndsBy

Parent: Predicate
Description: r1 and r2 have the same end time, and r1 starts strictly before r2 starts.
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  EndsBy(r1,r2)
```

C.7.14 Intersects

Parent: Predicate
Description: r1 and r2 has a non-null intersection: Intersects(r1,r2) $\Leftrightarrow$ Intersection(r1,r2) $\neq \emptyset$
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  Intersects(r1,r2)
```

C.7.15 StartsBefore

Parent: Predicate
Description: The start time of r1 $<$ the start time of r2
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  StartsBefore(r1,r2)
```

C.7.16 StartsAfter

Parent: Predicate
Description: The start time of r1 $>$ the start time of r2
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  StartsAfter(r1,r2)
```

C.7.17 TrueSubset

Parent: Predicate
Description: r1 is a sub-interval of r2, and r1 $\neq$ r2
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  TrueSubset(r1,r2)
```

C.7.18 Subset

Parent: Predicate
Description: r1 is a sub-interval of r2: Subset(r1,r2) $\Leftrightarrow$ Intersection(r1,r2) = r1
Arguments: object label, object label
Mandatory: object label, object label

```
Example:
  Subset(r1,r2)
```

Appendix D

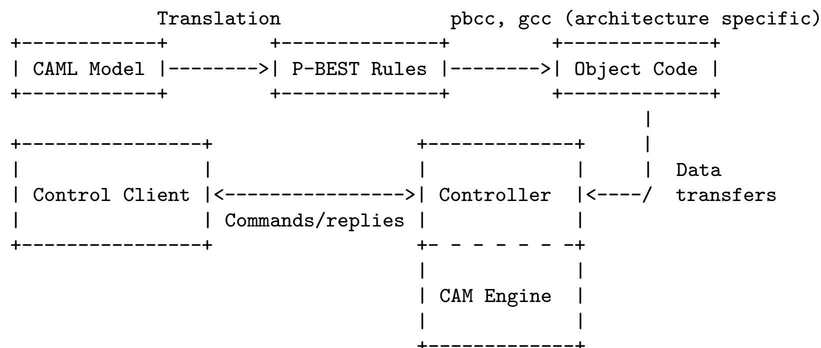
Correlation Engine Management Protocol

CAM ENGINE MANAGEMENT PROTOCOL
==============================

$Id: cam-engine-management.txt,v 1.6 2002/10/30 23:03:36 ulf Exp $

This document describes the architecture, protocol and procedures for controlling how the CAM scenario recognition engine (based on P-BEST and eflowgen) can be updated with new attack scenario models during run-time.

1. BASIC MODEL



An attack model described in CAML is translated into a P-BEST rules file. The P-BEST rules are translated into C by pbcc and compiled into object code by the C compiler (gcc). The object code is specific to a selected hardware architecture and operating system platform.

A Control Client is used to send commands to the CAM Engine Controller. The supported set of commands is: STATUS (report status), START (start the engine), STOP (stop the engine), LOAD (retrieve and load a new model in object code form), and UNLOAD (unload a specified model).

2. COMMAND PROTOCOL

The command protocol between the the Control Client ("the client") and the the CAM Engine Controller ("the server") is inspired by other simple text-based command-reply protocols such as FTP and SMTP.

The client initiates a TCP connection to a port on which the server listens. When the connection is established, the client can send a line consisting of a command code-word followed by parameters (where applicable). The server replies with a line beginning with a 3-digit numerical code (transmitted as three alphanumeric characters), followed by human-readable text.

The end-of-line sequence <EOL> consists of carriage return and linefeed, that is, the two ASCII characters '\r' (0x0d) and '\n' (0x0a).

A command line always begins with a command code-word, which is an alphanumeric string terminated by the character <SP> (Space 0x20) if parameters follow and <EOL> otherwise. Parameters are separated by <SP> and the line is terminated by <EOL> after the last parameter. Each command code-word is described in detail below.

A reply is defined to contain the 3-digit code, followed by Space <SP>, followed by one line of text (where some maximum line length has been specified), and terminated by <EOL>.

Example (lines formatted for clarity):

```
Client sends: STATUS <EOL>
Server sends: 211 <SP> Engine running since 2002-08-27 17:04:48 PDT,
                       arch "FreeBSD-4.5 i386",
                       tag "SRI worm 2002-08-06 v1.1",
                       tag "SRI ddos 2002-08-15 v1.3" <EOL>
```

2.1 STATUS

The STATUS command is used to query the server about its current status. The command takes no parameters.

The reply from the server indicates whether the CAM Engine is running or not (indicated both by the numerical code and by the text).

The reply also reports the architecture on which the CAM Engine is running, both with respect to operating system and hardware platforms, typically as produced by the Unix command "uname -msr".

The reply also reports the tag(s) associated with the currently loaded object code module(s), if any. The purpose of the tag is to identify the CAM model(s) currently loaded, for example so that the user can determine whether an update is required. The loaded modules are listed in the order in which they are applied to incoming events.

Possible reply codes:

```
   211 Engine running since <time>, arch <arch>,
       tag <tag> [, tag <tag> ...],
```

```
       number of event messages processed: <num>
   212 Engine stopped on request at <time>, arch <arch>,
       [ tag <tag> [, tag <tag> ...] ],
       number of event messages processed: <num>
   213 Engine stopped due to error at <time>, arch <arch>,
       [ tag <tag> [, tag <tag> ...] ],
       number of event messages processed: <num>
   510 Failed to return status
```

2.2 START

The START command will instruct the controller to start the CAM Engine. The command takes no parameters.

```
Possible reply codes: 221 Engine successfully started at <time>
                      222 Engine already running since <time>
                      520 Failed to start engine
                      521 Failed to start engine, no model loaded
```

2.3 STOP

The STOP command will instruct the controller to stop the CAM Engine. The command takes no parameters. When the engine is stopped, it will discard incoming messages.

```
Possible reply codes: 231 Engine successfully stopped at <time>
                      232 Engine was not running
                      530 Failed to stop engine
```

2.4 LOAD

The LOAD command will instruct the controller to retrieve a new object code file for the CAM Engine and load the new code into the engine. The order in which object code files are loaded is the order in which the models will be applied to incoming events. The loading of new object code files will not cause any previously loaded files to be unloaded - the UNLOAD command must be used to explicitly unload a file.

The first parameter to this command is a tag that becomes associated with the object code file. If the tag contains spaces, it must be enclosed within double quotes ("). If the tag matches an already loaded module, the load will fail. To update a loaded module, the module must first be unloaded with the UNLOAD command.

The second parameter to this command is the method by which the controller should retrieve the code image. Following the method are parameters that are specific to the given method, identifying the location and other information needed to retrieve the image with the specified method.

```
Supported methods:    ANON-FTP
                      Anonymous FTP, takes two parameters:
                      <hostname> hostname or IP address of FTP server
                      <path> full path and name of the file on FTP server

                      FTP
```

Non-anonymous FTP, takes four parameters:
<username> username for FTP login
<password> password for FTP login
<hostname> hostname or IP address of FTP server
<path> full path and name of the file on FTP server

Possible reply codes: 241 Successful retrieval and load, tag <tag>
540 LOAD failed
541 LOAD parameter error
542 LOAD parameter error, tag exists
543 LOAD parameter error, unknown method
544 Retrieval failed
545 Retrieval failed, authentication error
546 Retrieval failed, file not found
547 Load of object code failed
548 Load of object code failed, architecture mismatch

Example: LOAD <SP> "SRI ddos 2002-10-15 v1.4" <SP> ANON-FTP <SP>
ftp.cam-models.org <SP> /pub/i386/FreeBSD-4.5/ddos-1-4.o <EOL>

2.5 UNLOAD

The UNLOAD command is used to remove models from the CAM Engine.
The engine must be stopped for the command to succeed.

This command takes one parameter, which is the tag that identifies
the model to be unloaded. Regular expressions can be used, for example
to identify several models with a single command.

Possible reply codes: 251 Successful unload of model,
tag <tag> [, <tag> ...]
550 UNLOAD failed
551 UNLOAD parameter error
552 <reserved>
553 Nothing to unload, no model loaded with tag <tag>
554 Nothing to unload, no model loaded

Example: UNLOAD <SP> .* <EOL>
(to unload all loaded modules)

3. SECURITY

In the current version of this specification, there are no mechanisms
for protection of the communication between the Control Client and the
Controller. The currently supported code retrieval methods are also
insecure. Therefore, it is strongly recommended that all communication
takes place over protected network links for an implementation of this
version.

Index

*, 69
-, 69
?, 70

access-time, 80
activity, 68
activity-measure, 86
adaptive sensing and correlation, 45
AdditionalData, 90
Address, 73
address, 74
AdminPriv, 104
alert, 29
alert-count, 70
alert-priority, 85
alert-rank, 86
alert-relevance, 86
Alertident, 89
alt, 68
Analyzer, 70
analyzerid, 70, 90
anomaly, 85
anomaly-score-avg, 87
anomaly-score-max, 87
anomaly-score-min, 87
anomaly-score-std, 87
arg, 76, 79
Assessment, 83
attack, **30**
    composite, **30**
attack modeling, 29
attack patterns, 37
attack scenarios, 5
attack step, **30**
attack tree, 26

BiComm, 105

c-major-device, 82
c-minor-device, 82
CAML, 29, **31**
    activity section, 32
    examples, 32
    grammar, **67**
    module, 31, **35**
    post-condition section, 32
    pre-condition section, 32
    predicates, 36, **97**
    requirements, 31
category, 72, 73, 75, 80, 81
cgi, 79
change-time, 82
class, 71
Classification, 83
command, 79
community, 78
completion, 84
composite attack, **30**
Confidence, 84
const, 68
correlated-alert-outcome, 86
Correlation, 85
correlation engine, 40, 46
    management protocol, 48, **111**
CorruptedFile, 102
CorruptServerConfig, 99
CorruptServerContent, 99
create-time, 80
custom sensor, 48, 54

data-size, 80
Depends, 100
disk-size, 81

EndsBy, 109
EndsWith, 109
endtime, 70
engine, correlation, 40
env, 77
Event, 69
exploit, **30**

File, 80
FileAccess, 81
FileEq, 101
FileList, 79
FilePerm, 102
fstype, 80

global, 67

HasAppUser, 103

HasFile, 101
hashedpasswd, 75
HasOS, 97
HasPatch, 99
HasService, 97
HasSpecialFile, 101
HasUser, 103
HostBasedPredicate, 91
HostsSameSecurityDomain, 106
http, 79

ident, 72
Impact, 83
implement, 77
InDomain, 105
init, 68
Inode, 82
inquisitive sensor, 48
Intersects, 109
IsAfter, 107
IsBefore, 107
IsContained, 109
IsDuring, 109
IsEqual, 107
IsMet, 107
IsMetBy, 107

Know, 92, 106

Linkage, 81
location, 73

major-device, 82
manufacturer, 71
meaning, 90
merge-count, 86
merge-policy, 86
minor-device, 82
model, 71
model-confidence-avg, 87
model-confidence-max, 86
model-confidence-min, 86
modify-time, 80
module, 68

name, 73
NetBasedPredicate, 92
netmask, 74
Node, 72
number, 76

oid, 78
optype, 91
origin, 83
ostype, 71
osversion, 71
OverlapBy, 108
OverlapWith, 108

passwd, 75
patch, 77
patchlevel, 77
path, 76
permission, 81
pid, 76
port, 78
portlist, 78
post, 69
pre, 68
priority-max, 88
priority-min, 87
priority-avg, 88
Process, 76
protocol, 78

rank-avg, 89
rank-max, 89
rank-min, 89
rank-std, 89
rating, 84
reconnaissance, 22
references, 59
relevance-avg, 88
relevance-max, 88
relevance-min, 88
relevance-std, 89

sensor
    custom, 48, 54
    inquisitive, 48
Service, 77
severity, 83
SLAM, 49
SmallInputLargeOutput, 100
SNMPService, 78
Source, 72, 92
StartBy, 108
StartsAfter, 110
StartsBefore, 110
starttime, 70
StartWith, 108
Subset, 110
SuspiciousHost, 105
SuspiciousUser, 103
SwitchUser, 105

Target, 72, 92
thread, 70
TrueSubset, 110
type, 75, 84, 90

url, 79
User, 74
UserHashedPasswd, 104

UserId, 75
UserPasswd, 104

val, 85, 89–91
vendor, 77
version, 71
vlan-name, 74
vlan-num, 74
vulnerability, 24, **30**

WebExecutableDir, 98
WebRoot, 98
WebService, 79