1996

IDIOT - Users Guide

Mark Crosbie

Bryn Dole

Todd Ellis

Ivan Krsul

Eugene H. Spafford
Purdue University, spaf@cs.purdue.edu

Report Number:
96-050
IDIOT - USERS GUIDE

Mark Crosbie
Bryn Dole
Todd Ellis
Ivan Krsul
Eugene H. Spafford

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 96-050
September 1996
IDIOT - Users Guide.

Mark Crosbie mcrosbie@cs.purdue.edu
Bryn Dole dole@cs.purdue.edu
Todd Ellis ellista@cs.purdue.edu
Ivan Krsul krsul@cs.purdue.edu
Eugene Spafford spaf@cs.purdue.edu

Technical Report TR-96-050

September 4, 1996
Abstract

This manual gives a detailed technical description of the IDIOT intrusion detection system from the COAST Laboratory at Purdue University. It is intended to help anyone who wishes to use, extend or test the IDIOT system. Familiarity with security issues, and intrusion detection in particular, is assumed.
Chapter 1

Introduction

This document is the user's guide for the IDIOT intrusion detection system developed at the COAST Laboratory.

This section will remain short because a much better description of what IDIOT is, the design goals and the model it works under can be found in the documents included in the doc/IDIOT directory in the IDIOT distribution.

The files in that directory are:

- **kumar-spaf-overview.ps** [KS94] This report examines and classifies the characteristics of signatures used in misuse intrusion detection. The document describes a generalized model for matching intrusion signatures based on Colored Petri Nets. This is the first document you should read. We recommend that you stop reading this guide now and return here when you have finished reading that document.

- **kumar-intdet-phddiss.ps** [Kum95] Sandeep Kumar’s original Ph.D. thesis. An in-depth description of intrusion detection, the theoretical considerations behind IDIOT, and a description of the model that was used to implement IDIOT.

- **taxonomy.ps** [KS95] This report classifies UNIX vulnerabilities based on the signatures required to detect them, and gives the best overview on how to write IDIOT patterns with examples from real UNIX vulnerabilities. We recommend that you read this document last before you start writing patterns of your own.

- **IDIOT_work.ps** [ES96b] This report outlines the structure of IDIOT, explaining how the components fit and work together. It also describes results from profiling IDIOT against two audit trails and suggests possible approaches to optimizing the program.

- **debugging_IDIOT.ps** [ES96a] This report describes changes made to IDIOT code which allow a greater flexibility in the amount and type of debugging information generated. It also describes a sample IDIOT program that has been included and a utility for separating the debugging information based on its origin, pattern or server.
Chapter 2

Quick Start

The information in this chapter will help you get IDIOT running as fast as possible. However, we suggest that you read the rest of the material in this document before you attempt to use IDIOT.

The following steps must be executed to install IDIOT correctly:

- Read this document at least once
- Edit the Makefile to set the appropriate values for your site
- Give the command "make C2_app1"
- Make sure that praudit is in your path
- Run IDIOT

For the rest of this section we give an example of how to install and run IDIOT for the first time.

We start in a directory that contains only the IDIOT tar file:

```
solaris 51 $ pwd
/.../home/dollum/IDIOT
solaris 52 $ ls
didot.tar
```

Extract the files from the tar file:

```
solaris 52 $ tar xvf didot.tar
x doc, 0 bytes, 0 tape blocks  
x doc/manual, 0 bytes, 0 tape blocks  
x doc/manual/docs.ps, 12896 bytes, 266 tape blocks  
x doc/IDIOT, 0 bytes, 0 tape blocks  
x doc/IDIOT/know-how-intel-phyllis.ps, 203856 bytes, 1768 tape blocks  
x doc/IDIOT/knew-eapf-overview.ps, 672780 bytes, 1316 tape blocks  
x doc/IDIOT/asnemony.ps, 645109 bytes, 1260 tape blocks  
x C2_patterns, 0 bytes, 0 tape blocks  
x C2_patterns/creating-acid-scripts, 640 bytes, 1 tape blocks  
x C2_patterns/creating-init-flags, 1618 bytes, 3 tape blocks  
x C2_patterns/creating-get-props, 1295 bytes, 3 tape blocks  
x C2_patterns/print-hash, 936 bytes, 2 tape blocks  
x C2_patterns/secur-write-secud, 3026 bytes, 5 tape blocks  
x C2_patterns/writing-to-executable-files, 2672 bytes, 6 tape blocks  
x C2_patterns/writing-to-nommed-files, 1711 bytes, 4 tape blocks  
x apps, 0 bytes, 0 tape blocks  
x apps/profile,C, 1831 bytes, 4 tape blocks  
x apps/jig.C, 1265 bytes, 4 tape blocks  
x other_C2_patterns, 0 bytes, 0 tape blocks  
x other_C2_patterns/3-fails-logins, 1166 bytes, 3 tape blocks  
x other_C2_patterns/access-open-to-same-path-diff-likes, 4622 bytes, 9 tape blocks  
x other_C2_patterns/bin-mail, 2534 bytes, 5 tape blocks
```

2
Make the C2.app1 application:

```
solaris 5.4 % ls
DL_list.h   audit_trails/  poldit.h
C2_Server.h  Expr.C  doc/
C2_app1.h   IF_Server.C  idoct.tar
C2_patterns/  Makefile  other_C2_patterns/  utilities.C
DL_list.h   DM.opt
```

```bash
make C2_app1
```

```
C2_app1.C, line 138: Warning [Anachronism]: Formal argument pattern file of type char' in call to C2_Server::space_files(char) is being passed const char'.
C2_app1.C, line 138: Note: Type 'CC -migration' for more on anachronisms.
C2_app1.C, line 144: Warning [Anachronism]: Formal argument file of type char' in call to C2_Server::fill_file(char) is being passed const char'.
```

2 Warning(s) detected.
```
make C2_app1
```

```
C2_app1.C, line 138: Note: Type 'CC -migration' for more on anachronisms.
C2_app1.C, line 144: Warning [Anachronism]: Formal argument file of type char' in call to C2_Server::fill_file(char) is being passed const char'.
```

2 Warning(s) detected.
```
```
Make sure that the `praudit` command is in your execution path.

```
solaris & which praudit
/usr/sbin/praudit
solaris 6% set path=/path:/usr/sbin

Run the C2_appl application to test the program. In this case we will compile, link and run a sample pattern and audit trail shipped with IDIOT. Parse the pattern:
```
solaris 57 3  /C2_appl
<ini> parse C2_patterns/creating-setid-scripts
Parsing file C2_patterns/creating-setid-scripts
-----------------------------------------------
Inside stmt reduced expr: printf("User id \d has successfully setuid'ed file \"\n", unified_cok->get_UID(), unified_cok->get_FULL_NAME());

Parsing transition chmod
-----------------------------------------------

```
c <cc_setid_pgmso cc_setid_pgm.s>c
"cc_setid_pgm.s", line 290: Warning (Anachronism): Format argument val of type char
in call to C2_cc_setid_pgmso Token:="assembl FULL_VARS(char*) is being passed "oops char"
"cc_setid_pgm.s", line 290: Note: Type "CC <migration>" for more on anachronisms.
1 warning(s) detected.

DONE compiling cc_setid_pgm.s
Inside create pattern.
instantiated new pattern instance for cc_setid_pgm.s

Link the resulting compiled pattern:
```
tini> dlink /home/gollum/IDIOT/cc_setid_pgm.so
Linking file /home/gollum/IDIOT/cc_setid_pgm.so
inside create pattern.

Run the pattern with the corresponding audit trail.
```
tini> run audit_trails/creating-setid-scripts.audit_trail
Showaudit: will execute the following command: tail -6 audit_trails/creating-setid-scripts.audit_trail | praudit -r
User id 833 has successfully setuid'ed file /home/gollum/ass.ass
User id 833 has successfully setuid'ed file /home/gollum/ass.ass
User id 833 has successfully setuid'ed file /home/gollum/ass.ass
User id 833 has successfully setuid'ed file /home/gollum/ass.ass
User id 833 has successfully setuid'ed file /home/gollum/ass.ass
User id 833 has successfully setuid'ed file /home/gollum/ass.ass
Showaudit: no of dropped events = 147
Showaudit: Could not follow the next audit file at ./showaudit.pl line 32.

NOTE: If you get an error message of the form Don’t handle events of type 0 yet! then
IDIOT was not able to open the audit trail. Verify that the praudit program is in your path and try again.
Chapter 3

IDIOT Patterns

There are currently three classes of pattern being shipped with IDIOT. They are:

1. Written and tested patterns. These are patterns that we have written and tested and can present audit trails known to trigger the patterns.
2. Written but not tested. These patterns were written but could not be tested either through lack of time or limitations in the underlying audit trail.
3. Theoretical patterns. These patterns are written to demonstrate a particular point, but cannot be handled by the IDIOT system currently.

Each of the following sections describes the different pattern types.

3.1 Written and Tested patterns

These patterns have been written and tested. They are shipped as working patterns to demonstrate IDIOT’s capabilities, and to be used in a real environment.

These patterns can be found in the c2_patterns directory. The audit trails to exercise these patterns are in a directory audit_trails under the main IDIOT distribution directory. Each pattern has an associated audit file named <pattern_name>_audit_file in the audit_trails directory.

3.1.1 Detecting when programs create setuid programs. PATTERN:creating-setid-scripts

Very few programs should create setuid files and system administrators often run programs that search the file system looking for unauthorized setuid files. This pattern will warn when a program creates a setuid file.

Vulnerabilities detected

Many vulnerabilities can sometimes be detected with this pattern.
State machine

Figure 3.1 illustrates the pattern.

![State machine diagram]

Figure 3.1: State machine

Pattern being used

```c
pattern cr_setid_pgm "Monitor creation of setid programs," priority 7
    state start, after_chmod;
    str FULL_NAME;
    int RUID;
    post_action(
        printf("User id %d has successfully setid'ed file %s", RUID, FULL_NAME);
    )

    exec chmod(CHMOD)
    <- start;
    -> after_chmod;
    |-
        if this[RUID] == 0 && RUID == this[RUID] && FULL_NAME == this[OBJ] &&
        this[OBJ_NEWMODS] > 511;
    )
    end chmod;
    end cr_setid_pgm;

Note: the final clause in this guard would be best written as:

(this[OBJ_NEWMODS] & 06000) && (this[OBJ_NEWMODS] & 0111)
```

Discussion

The pattern described in this document was tested under Solaris 2.4 using the BSM C2 audit trail generated by the auditd daemon.

An attacker, a user called gollum, ran the following exploit script.

```
#!/bin/sh

touch aaa.aaa
touch bbb.bbb
touch ccc.ccc
touch ddd.ddd

chmod 0750 aaa.aaa  # setgid
chmod 0755 bbb.bbb  # setuid
chmod ugo=xs ccc.ccc  # setuid and setgid
chmod 6607 ddd.ddd  # undefined (setuid and setgid

# Get rid of the files created
rm aaa.aaa bbb.bbb ccc.ccc ddd.ddd
```
The execution of IDIOT server with the audit trail generated for user gollum produces the following output:

```
User id 833 has successfully setid'ed file /.sword/home/gollum/aaa.aaa
User id 833 has successfully setid'ed file /.sword/home/gollum/bbb.bbb
User id 833 has successfully setid'ed file /.sword/home/gollum/ccc.ccc
User id 833 has successfully setid'ed file /.sword/home/gollum/ddd.ddd
```

3.1.2 Detecting the execution of attack programs. PATTERN: executing-progs

Sometimes users import attack scripts and run them with little or no modifications. We can detect the execution of well known attack scripts and programs by searching for well-known program names.

Vulnerabilities detected

CA-93:14 Internet Security Scanner will scan sites for potential security holes.

CA*** Crack tool for obtaining passwords.

State machine

Figure 3.2 illustrates the pattern.

```
EXECVE

Figure 3.2: State machine
```

Pattern being used

```c
extern str Basename[sex];
pattern exec_progs 'Watch for executions of cpwd/plain etc.' priority 7
  state exec. after_exec;
  str FULL_PROG, PROG;
  int RUID;
  printf("User id \d has successfully executed \$\n", RUID, FULL_PROG);
  return 1;

// The routine Basename is defined as a call to the system function
// basename. Given a pointer to a null-terminated character string that
// contains a path name, basename() returns a pointer to the last element
// of the path. Trailing '/n' characters are deleted.

trunc exec(execkeve)
  <- start;
  + after_exec:
    [ this[ERR] = 0 && RUID == this[RUID] && PROG == Basename|this[PROG]);
```
Note: this pattern can also detect the [CA-95:06] advisory on the SATAN tool. This can be done by adding \texttt{PROG = "satan"} to the last clause in the guard.

Discussion

How does this pattern work? It is composed of essentially one transition — when an EXECVE occurs, it checks to see if the executed program is one of the prohibited list. If it is, then the pattern has terminated, and the post action is taken. This prints out a friendly informational message.

We need to know two things while executing this pattern; the name of the program being exec'd and the user id of the user executing the program. When the execve system call is executed, the name of the program to execute is specified. This is recorded in the audit file. We can access this program name using the \texttt{this[PROG]} construct, which returns the program name from this audit record. We record the full program name in the variable \texttt{FULL_PROG} so it can be displayed in the post action. This variable is of type \texttt{str}, which is a string class from the Rogue Wave library.

The user id is returned using the \texttt{this[RUID]} construct. It extracts the uid field of the audit record. We assign it to a local variable so it can be displayed later in the post action.

To actually detect running a specific program, we use the pattern matching operator \texttt{=} . This works as in PERL. In this case, we are only interested in exactly the filename specified.

The pattern described in this document was tested under Solaris 2.4 using the BSM C2 audit trail generated by the auditd daemon. An attacker, a user called \texttt{gollum}, has the following files in his home directory:

```
gollum # is -l
-rwxr-xr-x 2 gollum gollum 10 Nov 26 16:13 crack'
rwxr-xr-x 1 gollum gollum 45 Nov 26 16:14 exploit'
rwxr-xr-x 2 gollum gollum 10 Nov 26 16:12 hidden_hand'
rwxr-xr-x 1 gollum gollum 5 Nov 26 16:14 hidden_sym -> crack'
```

The exploit script executed contains the commands:

```
#!/bin/sh
./crack
./hidden_sym
./hidden_hand
```

The execution of \texttt{IDIOT} on the audit trail generated for user \texttt{gollum} on produces the following output:

```
User id 833 has successfully executed ./moder/home/gollum/crack
User id 833 has successfully executed ./moder/home/gollum/crack
```

The program has detected two of the three executions of the program \texttt{crack}. In fact, the two instances detected are the first two lines of the \texttt{exploit} script. It is unfortunate that we cannot detect which execution produces which line of output. As shown next, the audit trail events generated for both are identical.
3.1.3 lpr can copy a file over any arbitrary file: PATTERN: lpr_copy_files

The line printer program can be made to overwrite arbitrary files on the system. lpr is a setuid root program that copies files to print into the spooler directory. The files are then printed from the spooler directory. If the -s option is used, the lpr program creates a link to the file in the spool directory. However, the temporary names created in the spool directory will wrap around after 1000 print jobs.

By forcing these names to wrap around, the lpr program can copy an arbitrary file over a link which points to a destination file. This will cause the destination file to be overwritten, even if the user did not have permission to access that file.

Vulnerabilities detected


State machine

Figure 3.3 illustrates the pattern.

![State machine diagram](image)

Figure 3.3: Detect lpr copying arbitrary files

Pattern being used

```c
/* lpr_copy_files */
/* Detects if lpr tries to overwrite a file outside of /usr/spool */
/* [81gm]-Advisory-3.UNIX.lpr.19-Aug-1991 */
/* Mark Cresbie February 1996 */

extern int inTrees(char *str); extern int true_print(char *str);
```
The lpr command is a setuid root program that places files in the spool directory on behalf of users. Typically it places a copy of the file in the spool directory, but if given the -s option, it will create a symbolic link to the file in the spool directory. If given the -q option, it will place the file in the directory, but not enqueue it for printing.

The files in the spool directory have a very predictable name. The name of a spool file starts with cf for a control file and df for its associated data file. For example, executing the command lpr -Poz -q -s .aliases (where oz is the name of our local printer), creates the two files in the /var/spool/lpq/oz directory:

```
-RW-R-------- 1 daemon daemon 152 Feb 19 17:54 cfA278yavin.cs.purdue.edu
-rw-rw------- 1 daemon daemon 23 Feb 19 17:54 dfA278yavin.cs.purdue.edu -> /home/microshib/aliases
```

The number after the cfA and dfA part of the file names will increment after every print command. Thus, after a thousand print commands, the file dfA278yavin.cs.purdue.edu will be reused.

The essence of this attack is to create a link in the spool directory to a file you want to overwrite. Then, execute a thousand prints until the number in the spool directory filename wraps around, then print the file
you want to overwrite with. The lpr program will write over the existing link, and as it is setuid root, it can overwrite whatever that link pointed to.

The IDIOT pattern presented above detects this attack by seeing if the lpr program overwrites any files outside of the /var/spool directory tree. If so, it prints a warning. The audit trail in Solaris encodes the final destination filename when accessing a link, so this is available to the pattern.

A sample run is shown below:

```bash
$ dnb run ... udlt_dattflpr_toW_fl-le ....... "dit_dau
5h""a"dit, "ill
~a=ta
til" fOll"......"d, taU .0 ... "di.t_dau/lpr_toW_filt ....... "dit_dat ...
```

3.1.4 Print all executions of mknod. PATTERN: print-mknods

If a user creates a device, it is possible to access the disk or other system resources without proper authorization. For example, if a user created a device in a local directory with the same device numbers as the root disk, then that user could access and change any information on the disk.

Pattern State Machine

The state machine is very simple and is shown in Figure 3.4.

Figure 3.4: State machine for detecting mknods

Pattern code

```c
/*
 * print-mknods
 * Print all successful mknods
 * Mark Cecabio November 1995
 */

/* returns TRUE if the mode of the file corresponds to a block special
 * or char special file - is it a device?
 */
extern int iodevint;

pattern mknod 'Print all successful mknods' priority 7
  state start, after_mknod:
    int iuid, dev;
    /* path of the device the user created */
    str PATH;
    /* report creating a device */
```
post_section {
  printf("User id %d successfully created a device in \%a\n\", UID, PATH);
}

/* only one transition needed - if we see a MKNOD event then see if
  the user has successfully created a device
*/
trans makenode(MKNOID):
  => start;
  => after_mknod;
  l_ {<-
isdev() returns TRUE if the device mode used in the mknod()
  * indicates a device type (either char or block special)
  */
  chis[ERR] = 0 && isdev(this[DEV_MGER]) = 1 &&
  RUID = chis[RUID] && PATH = this[OBJ]1;
} end makenode;
end mknod;

Discussion

When a device is created, the mknod command is used. A parameter specifies the mode used to create the
device. If this mode indicates that this is a block special or character special file, then the message is printed.

A utility routine isdev() is used to check the mode of the device being created. If this is special file
(i.e. a device), isdev() returns TRUE. If this occurs, and the MKNOD succeeded, then the user ID and
path of the device are recorded and displayed.

Output from IDIOT

IDIOT produces the following output when presented with the attack audit trail:

tini: run audit_data/mknod.audit_data
Showaudit: will execute the following command: tsi d -D
audit_data/mknod.audit_data | praudit -r | User id 0 successfully created a device in
/home/prcablepry/IDIOT/audit_data/mkn.
Showaudit: No of dropped events = 79

3.1.5 Detecting when setuid programs write to setuid or executable programs. PATTERN:
setuid-writes-setuid

Very few setuid programs should write to other setuid or executable files, and many attack scenarios share
these characteristics. Rogue setuid programs will often create setuid shells or alter system owned executable
programs to alter their behavior. Hence, it is desirable to detect whenever such behavior and report it as
suspicious.

This pattern will warn when a setuid program creates or opens a setuid file (or executable program) for
write. It will also report when a setuid program opens a setuid or executable file for read only, using the
open system call, and specifies that the file should be created if it does not already exist.

Vulnerabilities detected

Many of the vulnerabilities listed below can sometimes be detected with this pattern. In particular, when
any of these setuid programs creates or opens a setuid or executable file.
Any user with access to lpr can alter system files and thus become root. We can only detect the vulnerability if lpr creates or opens for write a setuid or executable file.

sadc can be used to create files in normally unwritable directories. sadc normally runs egid sys, and therefore can be used to create files in group sys writeable directories. We can only detect the vulnerability if sadc creates or opens for write a setuid or executable file.

A race condition exists in binmail, which allows files to be created in arbitrary places on the filesystem. These files can be owned by arbitrary (usually system) users. We can only detect the vulnerability if binmail creates or opens for write a setuid or executable file.

The old race condition still exists in the patched binmail, which allows files to be created in arbitrary places on the filesystem. These files can be owned by arbitrary (usually system) users. We can only detect the vulnerability if binmail creates or opens for write a setuid or executable file.

A hole in binmail allows files to be created as root. We can only detect the vulnerability if binmail creates or opens for write a setuid or executable file.

CA-95:05 Sendmail Vulnerabilities February 22, 1995. This advisory supersedes all previous CERT advisories on sendmail. The CERT Coordination Center has received reports of several problems with sendmail, one of which is widely known. The problems occur in many versions of sendmail. We can only detect the vulnerability if sendmail creates or opens for write a setuid or executable file.

CA-95:08 Sendmail v.5 Vulnerability August 17, 1995. In sendmail version 5, there is a vulnerability that intruders can exploit to create files, append to existing files, or execute programs. We can only detect the vulnerability if sendmail creates or opens for write a setuid or executable file.

State machine

Figure 3.5 illustrates the pattern.

Pattern being used

```c
extern int issetid(int), ispid(int);

/*
 * Very few setuid programs should write to other setuid or executable
 * files. This pattern will warn when a setuid program creates or
 * opens a setuid file (or executable program) for write.
 */
pattern secuidwriteisetuid "setuid program writes to a setuid or executable file" priority 10
int PID;
int FILE;
sec PROGRAM;
state state, after, after, violation;
post_option {
    printf("setuid program %s created/opened for write the setuid or executable file %s.
", PROGRAM, FILE);
}
```
Figure 3.5: State machine
Discussion

The pattern described in this document was tested under Solaris 2.4 using the BSM C2 audit trail generated by the auditd daemon. The following setuid programs were created:

```c
/* Program: test1 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
    /* create a setuid executable file */
    create("test1.c1", S_IRWXU | S_IXGRP | S_IXGRP);
}

/* Program: test2 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
    /* Create a setuid program */
    open("test2.c1", O_CREAT, S_IRWXU | S_IXGRP | S_IXGRP);
    /* Open the setuid program created by program test1 */
    open("test1.c1", O_RDONLY);
}
```

An attacker, a user called gollum, ran these two programs in order.

The execution of the IDIOT server on the audit trail generated for user gollum on produces the following output:

```
setuid or executable file /home/kral/test1 created/opened for write the
setuid or executable file /root/home/gollum/test1.c1.
setuid program /home/kral/test1 created/opened for write the
setuid or executable file /root/home/gollum/test1.c1.
```

3.1.6 Writing to Executable Files. PATTERN: writing-to-executable-files

Writing to an executable file is often evidence of viral behaviour. A virus may attempt to affect a program by inserting code at the start of the file which will be executed before the program contained in the code. This is very common on MSDOS machines. However, a UNIX system may still be vulnerable to this type of attack if executable files are left in place with write permission for users or groups.

Vulnerabilities detected

Any activity in which an executable file is written to.
Pattern State Machine

The pattern is very simple. For each process that is EXECed, the pattern detects writes to a file by that process that has the execute permission bits set. If such a write is detected, the pattern has fired. Figure 3.6 shows this.

![State machine for pattern to detect viral behaviour](image)

**Figure 3.6: State machine for pattern to detect viral behaviour**

Pattern code

```c
/*
* writing-to-executable-files
* detect programs writing to executable files which may be an
* indication of viral behaviour
*/

extern int isexec(int);
extern int true_printstr(int);

pattern virus "Programs writing to executable files (viral behaviour)" priority 10

int PID, EUID; /* pattern local variables may be initialized. */

eae FILE, PROG;

state start, after_exec, violation;

/* post action simply reports on attack */
post_action {
    printf("Program %s running as EUID %d wrote to the executable file %s\n", PROG, EUID, FILE);
}

/* There can be >= 1 invariants */
```
/* this invariant kill a token if its corresponding parent exits */
new invariant first_inv /* negative invariant */
state start_inv, final;

trans exit(EXIT) <- start_inv;
  -> final;
  [ PID = this(PID); ]
end exit;
end first_inv;

/* pattern description follows */

/* EXECVE is the event type of the transition - executing a process */
trans exec(EXECVE)
  <- start;
  -> after_exec;
  [ ( chis(SID) = 0 && PID = this(PID) && PROG = this(PROG) &&
     EUID = chis(EUID) ); ]
end exec;

/* OPEN_RW - open a file for read or write */
trans mod0(OPEK_RW)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && PID = this(PID) && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

trans mod0(OPEK_WTC)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && PID = this(PID) && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

trans mod0(OPEK_RW)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

trans mod0(OPEK_NC)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && PID = this(PID) && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

trans mod0(OPEK_N)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

trans mod0(OPEK_WTC)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && PID = this(PID) && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

trans mod0(OPEK_WT)
  <- after_exec;
  -> violation;
  [ ( this(ERR) = 0 && PID = this(PID) && FILE = this(OBJ) &&
     isexec(this(OBJ_MODS) ); ]
end mod0;

end virus;
The pattern seems complex, but is essentially very simple. The pattern starts by detecting the execution of a new process using the EXECVE transition. From this after_exec state, it detects any open calls that could potentially write over a file which has its execute permission bits set. The isexec function is in utilities.c and returns TRUE if a file is executable. This can be computed by information about the opened file stored in the audit trail. The types of open call detected are:

- OPEN_W open for write.
- OPEN_WC open for write and create if not present.
- OPEN_WT open for write and truncate to zero size if present.
- OPEN_WTC open for write and create and truncate to zero size.
- OPEN_RW open for read or write.
- OPEN_RWC open for read or write and create if not present.
- OPEN_RWT open for read or write and truncate to zero length.
- OPEN_RWTC open for read or write and create if not present and truncate to zero size.

If any of these occur and the file is executable, we could have potential viral activity on the system. It could also indicate an attempt to install a trojan horse on the system. This would be a version of a program that had similar functionality but installed a malicious piece of code.

Discussion

The pattern will only detect a process being executed that overwrites executables. A series of shell commands to overwrite an executable won’t be detected, because each shell command results in a new process with a new PID. The PID attribute of the tokens won’t be unifiable.

Why does this pattern not detect WRITE events? In the Solaris BSM audit trail the write events are subsumed under the OPEN event. So the audit trail for a write looks as follows (output of praudit):

```
header,135.2,open(2) = read_write, Thu Feb 20 10:53:34 1995, - 846295300 msgid path, /home/mccarroll/my/210/calendar_data/Executable
attribute,100711, owner, mccarroll, mode, exec, 8198630,193085,0 subject,-2, mccarroll, owner, mccarroll, 20068,0,0 0 0 0 0
return success
```

The audit trail was gathered with the fw audit mask flag. This gathers data about file writes. The open call is recorded, but the write call is not. The above audit trail corresponds to the following code fragment from exploit_write.c:

```c
if (fopen("/executable", O_RDWR) < 0) {
    perror("open: ");
    exit(1);
}

printf("Error: writing to executable...
");
if (write(fd, "abcdef", 6) < 0) {
    perror("write: ");
    exit(1);
}
```
Thus, we are forced to detect writes to executable files by detecting the opening of the file, and not the actual write.

This pattern can also be written without the EXECVE transition. In effect it would then check every write on the system to see if it overwrites an executable file.

3.1.7 Writing to nonowned files. PATTERN: writing-to-nonowned-files

Users often create files in their home directories that are world writable. This is often because of a umask setting that does not disable the world write bits. Or, users may be sharing information via world writable files. Either way, this is a potential avenue for attack, as any information can be written into a non-owned file for later retrieval. This may allow an attacker to hide stolen information or plant a trojan horse in another user’s directory. In the worst case, an attacker could place a ++ into a world-writable .xhostes file owned by another user, leaving that user’s account open to abuse.

Pattern State Machine

The state machine for this pattern is presented in Figure 3.7.

![State machine for pattern to detect writing to nonowned files](image)

Figure 3.7: State machine for pattern to detect writing to nonowned files

Pattern code

The code for the pattern is as follows:
The pattern looks complex, but is essentially very simple. It detects any writes to a file that has an owner field (OBJ_OWNER) that differs from the current effective user ID (EUID). If so, then the pattern reports the write as being suspicious.

Discussion

This pattern is very simple. If the owner of a file differs from the effective user ID of the process writing to the file, then the post action reports this. Note, see the description for writing-to-executable-files for more details on how the audit trail represents write events.
3.2 Written but untested patterns

This section describes patterns that are written but untested. These patterns can be tested in the current IDIOT implementation, but they are being shipped "as is" without any testing.

These patterns can be found in the other.C2.patterns directory under the main IDIOT distribution.

3.2.1 PATTERN: lpd.delete_files

This pattern attempted to detect the CERT vulnerability outlined in CERT Advisory CA91:10.a. The lpd line printer daemon can be made to delete files outside of the usual /var/spool directory by exploiting a race condition. The race condition exists between the time the file is copied into the spool directory and the time the file is removed from the directory. By changing the file to a link, it was possible to remove any file on the system (as lpd ran as root).

This pattern cannot be tested on our machines, because our lpd daemon has the fix in place and we cannot replace it without seriously disrupting the local printing environment in COAST.

However, the core transition in the pattern detects a delete (an UNLINK event) occurring with a destination outside of a specified directory tree. It uses the inTree() external function (found in utilities.C to match the file being deleted with the path prefix /usr/spool. If no match is found, then we conclude that lpd is trying to delete a file outside of the print spool directory.

```c
trans delete (UNLINK)
  <- after_lpd_exec;
  -> after_delete;
  | | {
  |  |  chid[0x0] = chid[0x1] = vid as FILE = chid(0x1) as
  |  |  inTree('/usr/spool', FILE) = 0;
  | | };
  | end delete;
```

3.2.2 PATTERN: failed.su

This is a very simple pattern that reports all failed su attempts. Most system consoles also report similar information. The pattern simply looks for SU events in the audit trail that have an err attribute of 1, indicating that they have failed.

3.2.3 PATTERN: finger

This pattern attempts to detect the finger program spawning a program other than finger. This was motivated by the Internet worm attack where the stack for the finger daemon was overwritten and caused a new program to be spawned.

This pattern cannot be tested because it hard codes the inode value for the finger daemon fingerd and the finger program finger into the pattern. This will change from site to site, and from system to system. The pattern should be rewritten to allow a more portable specification of the location of the daemon program.
3.2.4 PATTERN: priv-pgm-in-userspace

This pattern attempts to detect a privileged program executing in user space. It does this using the `inUserSpace()` external function.

3.2.5 PATTERN: rcw

This pattern attempts to verify integrity using Clarke-Wilson triples. More information on this pattern and Clarke-Wilson integrity triples can be found in Section 4.8.9 of the taxonomy.ps file in the doc/IDIOT directory of the distribution.

3.2.6 PATTERN: shell-script-attack

Certain classes of attack make a shell execute and pass it strange command line arguments. A favourite attack is to run a shell with a command line argument of `-i`. This gives an interactive shell. If, somehow, a root shell can be spawned from a program with the `-i` option, an intruder has an interactive root shell on the system.

The key to this pattern is this transition:

```c
trans exec(EXECVE)
<- start;
-> after_exec;
| [this[PROG]] = 0 & RUID = this[RUID] & PROC = this[PROC] &
  (basename(this[PROC]) = "-i")
}
end exec;
```

The `basename()` external function gets the actual name of the program being executed, with the leading path component stripped off. If this name starts with a `-` symbol, the pattern indicates that something suspicious has occurred.

3.2.7 PATTERN: tftp

This pattern attempts to detect the `tftp` program accessing files outside of the `tftpboot/` directory. This could indicate an attacker using the TFTP protocol as an attack vector.

This pattern has too many hardcoded constants to be useful in its current form. It hardcodes the inode numbers for the tftp program into the pattern. This should be changed to a more natural way of specifying the name of the program.

3.2.8 PATTERN: bin-mail

This pattern attempts to detect the `/bin/mail` program writing to a file that has its setid bit on, or is executable. The pattern is very straightforward. The execution of the mail program is checked by the transition on the EXECVE event. This is coded rather awkwardly — it detects the execution of the mail program by looking at the `inode` of the argument to the `exec()` call. This is very system dependent, and should be fixed.
It then checks to see if that program opens any files for writing that have their execute bit set (using the isexec() utility function) or have their setid bit set (using the issid() function). In either case, the pattern triggers, indicating that the mail program probably has been subverted.

The invariant deletes any tokens associated with a process when that process exits. Thus, as a mail program runs it generates a token in the machine, and once that program exits, all tokens associated with it are deleted.

3.2.9 PATTERN: setid-pgms-cant-spawn-shell

A setuid program is not allowed to spawn a program with the effective user id unchanged. This would allow attackers to subvert (say) the mail program, and make it spawn a shell with an effective user id of root.

This pattern is conceptually simple — if an EXECVE occurs of a shell program and the EUID changes after the call, then trigger the pattern. However, because of a weakness in Sun BSM auditing, this information is not available. Instead, the pattern must monitor on a per-user basis — if a user executes a setid program that then spawns a shell with a user id different from the user's, it triggers the pattern. This is less flexible than the generic pattern, but is workable under Sun BSM auditing.

3.3 Theoretical patterns

This section describes theoretical patterns. These patterns cannot be tested in the current IDIOT implementation, and are written for instructive purposes only.

These patterns can be found in the other_C2_patterns directory under the main IDIOT distribution.

3.3.1 PATTERN: access-open-to-same-path-diff-inodes

The general problem that this pattern tries to detect is that of an attacker to exploit the small window of opportunity that exists between programs checking the validity of the files and the actual opening of the file for writing. If an attacker can slow down the system, by running the program with a very high nice value, for example, he can replace the file that has been validated with another of his choosing.

passwd -F Attack In SunOS

The passwd command can be directed to treat another file as the password file using the -F option. Before opening this file for writing, and depending on the version of Unix one is using, the passwd program will attempt to determine that the user can indeed read and write to the file by either opening it for read, calling the access system call or calling the stat system call. If after this check has been completed, and before the file is actually opened for writing, the user changes the path name to point to the real system password file, the passwd program will operate on this file assuming it is still working with the original file.
xterm Logging Attack

xterm needs to run as root because it needs to chown the tty it allocates to interact with the user. Logging is a security hole because of the race condition between access check permissions to the logging file and the logging itself. When given the -lf option, xterm creates a file by calling the creat system call and immediately changes the uid and gid of the file to match that of the user running the program and start logging.

The problem appears when a user gives the following sequence of commands:

```
xmod foo p & foo is the FIFO named pipe
xterm -lf foo & creat will open the FIFO and block because there is no process reading the other end of the pipe
mv foo junk
ln -s /etc/passwd foo
```

...will open the FIFO and block because there is a reader at the other side of the named pipe, the creat system call returns and the next statement executed is the chmod system call that will change the permission of the /etc/passwd file so that the user can add and delete entries.

Although more difficult to exploit, the patched version of xterm continues to have a similar problem. The code described above got replaced by:

```c
1 if (access(screen->logfile, F_OK) != 0) {
2 if (errno == ENOENT) {
3      creat_as(screen->uid, screen->gid, screen->logfile, 0644);
4    } else {
5        return;
6    }
7 }
8 if (access(screen->logfile, F_OK) != 0)
9     if (access(screen->logfile, W_OK) != 0) {
10       return;
11    }
12 if (access(screen->logfile, O_CREAT | O_WRONLY, 0644) != 0) {
13      return;
14    }
```

The creat system call was replaced by the creat_as routine. This routine forks a child process that changes its permissions to that of the user before it attempts to create the file, effectively eliminating the previous race condition. Unfortunately, a race condition, although much harder to exploit, still exists between the access call in line 8 and the open call in line 12. If a user can slow down this program and time things right, after the access call succeeds, the log file can be replaced by a link to the real /etc/passwd file. The xterm process will open the /etc/passwd for writing, and append to it the information that it is supposed to log.

Vulnerabilities detected

CA-93:17 xterm Logging Vulnerability.

8igm 5/11/1994 The passwd command takes a -F option in SunOS.

Pattern State Machine

The state machine is shown in Figure 3.8.
Figure 3.8: Attack state machine
Pattern code

/ * passwd rump advisory, system CA-95:17 advisory */
/ * detects the classic 'window-of-opportunity' exploitation. Used in */
/ * passwd program and system logging bug. */
/ * Mark Crable  Nov. 1995 */
/ * Ivan Krust  Nov. 1995 */
/ */
extern int inode[STR]; /* returns the inode for this particular file */

pattern executable 'stare-open-to-same-path-diff-inodes' priority 7
state start;
  nodup state after_accesses;
  nodup state after_create;
  state after_open;
  state after_exec;
  state violation;
int PID, INODE, UID, RUID;
six FILE, PROG;
post_action{
  printf('Violation!');
}

/* this invariant states that the patterns are matched as long as this */
/* process is running */
/* neg invariant inv /* negative invariant */
state start_inv, final_inv;

transition pre_exit(EXIT)
  < start;
  --> after_accesses;
  | [ this(UID) = 0 & & PID = this(PID) & & PROG = this(PROG) & & UID = this(RUID) & & RUID = this(RUID) & & RUID = RUID ];
end transition;

transition exec(EXIT)
  < start;
  --> after_create;
  | [ this[USR] = 0 & & PID = this(PID) & & PROG = this(PROG) & & UID = this(RUID) & & RUID = this(RUID) & & RUID = RUID ];
end transition;

/* this transition is taken when a program creates a file calling the create */
/* system call */
transition create(CREATE)
  < after_create;
  --> after_create;
  | [ this[USR] = 0 & & PID = this(PID) & & INODE = this(INODE, INODE) & & FILE = this(FILE) ];
end transition;

/* Programs can also create files by calling the OPEN_WC or OPEN_WC call and */
/* closing them. The next three transitions are designed to catch this. */
transition opencreate1(open_wc)
  < after_open;
  --> after_open;
  | [ this[USR] = 0 & & PID = this(PID) & & INODE = this(INODE) & & FILE = this(FILE) ];
end opencreate1;

27
trans opencreate2(OFDC, MC){
  <- after_exec;
  -> after_open;
  [...]
end opencreate2;

trans close(CLOSE)
  <- after_open;
  -> after_create;
  [...]
end close;

/*
 * After a file has been created, if the same file path was used as a
 * parameter to the chown or chmod routine but the mode is different
 * then we have a violation.
 */
trans chown(CHOWN)
  <- after_create;
  -> violation;
  [...]
end chown;

/*
 * This failing transition is taken after a successful exec if a file is tested
 * for existence with the stat or access call
 */
trans accessACCESS
  <- after_exec;
  -> after_access;
  [...]
end access;

trans stat(STAT)
  <- after_exec;
  -> after_access;
  [...]
end stat;

/*
 * If the same file that was opened for read is now opened for write
 * and the inode number has changed then we have a problem
 */
trans write1(OPGW, W)
  <- after_access;
  -> violation;
  [...]
end write1;

trans write2(OPGW, WC)
  <- after_access;
  -> violation;
  [...]
end write2;

trans write3(OPGW, W)
  <- after_access;
  -> violation;
  [...]
end write3;

trans write4(OPGW, W)
  <- after_access;
  -> violation;
  [...]
end write4;

end statepck;
Discussion

This pattern can detect the attack by seeing if the inode for the file has changed. How does this happen? When the file is first opened, it will have an inode given by the destination of the link. If an attacker moves the original file and creates in its place a link that points to another file, this inode will change. Thus, by detecting this change in inode value, the attack can be detected.

Unfortunately, this pattern cannot be used as is with the Basic Security Module C2 logging provided by Sun with Solaris 2.4. For this pattern to work, the audit trail would need to provide, at least, the following information:

1. execve: ERROR, EUID, RUID, PID, and program name
2. creat: ERROR, PID, initial file name, final file name, and inode number
3. open: ERROR, PID, initial file name, final file name, and inode number
4. close: ERROR, PID, inode, initial file name, and final file name
5. chown: ERROR, PID, inode, initial file name, and final file name
6. chmod: ERROR, PID, inode, initial file name, and final file name
7. access: ERROR, PID, inode, initial file name, and final file name
8. stat: ERROR, PID, inode, initial file name, and final file name

The only items that require explanation are the file names. If we have a symbolic link sym1 that points to a file fill then a call to chmod("sym1", mode) would require an audit trail record that would indicate that the chmod system call was executed on the initial file name sym1 and the final file name fill. The inode should be the one corresponding to the final name.

The audit trail generated by the Basic Security Module provided with Solaris 2.4 violates the file names requirement. To see why this is important, consider the following fragment of a setuid C program called xtermbug:

```
if( create("xtermbug.log", 0) == -1 ) { 
  /* The permissions of the created file were set to 0x00000. Change them to something more reasonable */
  if(chmod("xtermbug.log", S_IXUSR|S_IXGRP|S_IXOTH) == -1 ) {
    fprintf(stderr, 'Could not change the permission of file xtermbug.log\n');
  }
  else {
    fprintf(stderr, 'Could not create log file xtermbug.log\n');
  }
}
```

and an attack script that will exploit the vulnerability described in this document is called xtermbug.exploit:

```
#!/bin/sh
chmod xtermbug.log p
xtermbug &
sleep 1
mv xtermbug.log junk
ln -s /homes/kruai/break_me xtermbug.log
cat junk
cat /homes/kruai/break_me
```

29
This attack can not be detected by the pattern described in this document if you are using the Basic Security Module logging in Solaris 2.4. The audit trail generated for this session, reformatted for clarity and brevity, with the attacker called *gollum*, and the victim called *krsul*, is:

```
1 execve(2) = path./home/gollum/xtermbug.exploit - RUID gollum - RUID gollum
2 execve(2) = path./usr/bin/shmod - RUID gollum - RUID gollum
3 shmod(2) = argument.2.0x1b6.mode - argument.1.0x0,6w
4 path./modror/home/gollum/xtermbug.log - RUID gollum - RUID gollum
5 chown(2) = argument.2.0x341.new file uid - argument.1.0x341.new file gid
6 path./modror/home/gollum/xtermbug.log - RUID gollum - RUID gollum
7 execve(2) = path./home/krsul/bin/xtermbug - RUID krsul - RUID gollum
8 execve(2) = path./usr/bin/keep - RUID gollum - RUID gollum
9 execve(2) = path./usr/local/bin/mv - RUID gollum - RUID gollum
10 rename(2) = path./modror/home/gollum/xtermbug.log
11 path./modror/home/gollum/junk - RUID gollum - RUID gollum
12 execve(2) = path./usr/bin/ln - RUID gollum - RUID gollum
13 symlink(2) = text./homes/krsul/break_me - path./modror/home/gollum/xtermbug.log
14 RUID gollum - RUID gollum
15 execve(2) = path./usr/bin/cat - RUID gollum - RUID gollum
16 creat(2) = path./modror/home/gollum/xtermbug.log - argument.3.0x2敶 cessp: open
17 RUID krsul - RUID gollum
18 chmod(2) = argument.2.0x1f11.new file mode - path./homes/krsul/break_me
19 RUID krsul - RUID gollum
20 execve(2) = path./usr/bin/cat - RUID gollum - RUID gollum
```

Notice that in line 20, the audit trail indicates that the `chmod(2)` system call was made to the file `/homes/krsul/break_me`. While this is ultimately true, it does little to help detect the exploitation of the vulnerability. The pattern design specifically looks for a `chmod` or `chown` to the same file name as the `creat` but with different inode numbers. The audit trail should mention that the original call was made to file `xtermbug.log`.

### 3.3.2 Pattern: 3-failed-logins

This pattern attempts to detect failed login attempts. Specifically, it looks for more than 3 failed login attempts in a 3 minute time period. This could indicate that a password attack is occurring against a user account.

This pattern cannot be tested because IDIOT currently does not support a CLK event. The idea behind a CLK event is to allow timing information to be incorporated into the pattern. Consider the code for the pattern:

```java
/* CLK has not been implemented yet */

neg invariant inv

state start_inv, final_inv;

trans clk(CLK)
<- start;
-> final;
| [ this[TIME] - time > 100 ]
end clk;
end inv;

/* pattern description follows */

trans fail1(LOGIN)
<- start;
-> after_fall1;
| [ this[EN] = 1 && ruid = this[RUID] && time = this[TIME] ];
end fail1;

trans fail2(LOGIN)
<- after_fall1;
-> after_fall2;
| [ this[EN] = 1 && ruid = this[RUID] ];
end fail2;

trans fail3(LOGIN)
```
The first failed login attempt sets the time attribute of the token. This is taken from the timestamp information in the audit trail. The third failed login attempt can use this to make a transition if less than 180 seconds (3 minutes) has passed. The CLK event occurs at predictable time intervals (say every second). It is used in the invariant to delete any tokens that have been in the state machine for longer than 3 minutes.

The CLK event should be simple to add to IDIOT and would provide great flexibility in writing patterns to detect failed login, network access and other object creation attempts.

3.3.3 PATTERN: dir_browser

This pattern detects a user browsing through directories. It does this by counting the number of change-directory events in the audit trail. Normally a user executes a few chdir commands in a session, but an abnormally high number of these commands could indicate someone browsing where they shouldn’t be.

This pattern cannot be tested because it uses the CLK event as described in the previous section. However, it illustrates a useful way of detecting if someone is browsing through sensitive directories. It uses the inode of a shell process to identify which process to monitor — most users browse using their shell. It then counts the number of CHDIR (change directory) events in the audit trail. If this count exceeds a threshold, it triggers the pattern.

3.3.4 PATTERN: le

This pattern tests to see if an ethernet device was put into promiscuous mode. In the words of the comments to the pattern:

Establishing a hardlink to /dev/le and opening the hardlink is not possible because hard links cannot be established across devices. Don’t know how to test for promiscuity yet.

This pattern may not be possible to implement at all, but is included as an example.

3.3.5 PATTERN: port_walk

This pattern attempts to detect a port walk attempt. It uses accept events in the audit trail to detect an intruder connecting to ports across the machine in an attempt to see what services are running. It cannot be tested or implemented as IDIOT does not support the accept event yet. If it were extended to handle such events, then a wide variety of network attack patterns could be detected. See the section on limitations of auditing for more information.

3.3.6 PATTERN: dont-follow-sym-links

This pattern attempts to detect a setuid root program following a symbolic link. This a potential vulnerability as an attacker could replace the link with a link to another file, and subvert the operation of the setuid root
program.

This pattern contains a subtle flaw. Consider the following code snippet:

```c
trans open5(open3, A)
<- start;
-> after_open;
_. ( this[ERR] = 0 & this[UID] = 0 & islink(this[ONAME]) &
    FILE = this[OBJ] & PID = this[PID] );
end open5;
```

This code detects if a file was opened for reading by a process running with root privileges. It uses the `islink()` call to test whether the file was a link. This is a mistake — the state of the file system may have changed between when the audit data was generated and the pattern is being run. This is explained in the section on External declarations in the technical documentation.

This pattern is an example of how not to use external functions to carry out computations. External functions should never query system state that may change between when the audit trail was generated and when the pattern is run.

### 3.3.7 PATTERN: timing-attack

This pattern also uses the CLK event, but to detect a race condition attack where a file is unlinked and relinked to a new destination before being accessed. This is a classic "time-to-check-to-time-to-use" attack.

The first transition records the time that shell script starts to execute a program pointed to by a link. The next transition is taken when the same file is unlinked within 1 second. The final transition is taken if that same file is relinked to a different location within 1 millisecond. This triggers the pattern — an attacker is attempting to relink a setuid link to point at a file he/she wishes to gain access to.

The invariant in this pattern deletes any tokens that have been in the state machine for longer than 5 seconds. As the CLK event is not implemented, this invariant won’t compile under the current version of IDIOT.
Chapter 4

Users Guide

4.1 The C2_appl Application

Testing a pattern is a straightforward procedure. It involves compiling the pattern, linking the pattern into a program that can process the pattern and running the pattern with an audit trail. In IDIOT the three steps are performed using the same program: C2_appl.

The patterns can be stored in arbitrary directories. The example shown here assumes that the pattern is stored in a directory called $IDIOT_HOME/C2_patterns. Note that the descriptions of these patterns are normally not located in this directory.

To compile a pattern run the C2_appl application and type the command “parse <pattern name>”. The following figure is an example of compiling a pattern from the C2_appl application:

```
$ cd $IDIOT_HOME
$ ./C2_appl

$ parse C2_patterns/actuid-writes-actuid

Parsing invariant transition exit

<Some states deleted>

Parsing transition mod1

...
After compilation of the pattern you should be able to see the C++ and relocatable files produced in the $IDiot_HOME directory. The files generated will have the name that was indicated in the first line of the pattern description. For example, in the following pattern the name given to the pattern is setuidwritessetuid and hence the resultant files will be setuidwritessetuid. C and setuidwritessetuid.so.

```c

After you have compiled the pattern you must link it to the C2_appl program by issuing the "dlink <pattern name>" command. Note that the pattern needs to be compiled only once and can be linked many times in many different runs. In the following example the pattern compiled earlier in this section is linked in a different session.

Running the pattern on an audit trail involves giving the "run <audit trail file>" command. The following figure shows an example of running a pattern on a C2 audit trail.
4.2 Writing IDIOT Patterns

This section will take a step-by-step approach to writing a pattern. Along the way, each element of a pattern will be described. This will enable you to read an IDIOT pattern and understand what it is doing, and to write your own patterns.

As an example, we will use the other_C2_patterns/lpr_copy_files pattern. We will explain each part of the pattern as it is developed.

4.2.1 Basic structure of a pattern

A pattern consists of four key components:

1. A *name* for the pattern — *lpr_copy_files* in our case. The name is found just after the *pattern* keyword.
2. A post action which is executed when the pattern is matched.

3. An invariant which specifies when tokens are to be deleted.

4. A set of transitions between states, with associated guard expressions.

Of these, the post action and invariant are optional. A pattern without a post action doesn't make much sense as it won't be able to report to the outside world. A pattern need not have an invariant if it does not need to delete tokens.

The general structure for a pattern is as follows:

```plaintext
<external declarations>

pattern <pattern name> */optional comment*/ priority <priority value>
<token color declarations>
[post action ( <post action code> )]
[invariant <invariant name> /* optional invariant */
<invariant state declarations>
  trans <transition name><event>
  - <transition from state name>;-><transition to state name>
  }{ <guard expression>;
  end <transition name>
end <invariant name>;
/* actual pattern transitions start here */

trans <transition 1 name><event>
  - <transition from state name>
  -><transition to state name>
  }{ <guard expression>;
  end <transition 1 name>

trans <transition 2 name><event>
  - <transition from state name>
  -><transition to state name>
  }{ <guard expression>;
  end <transition 2 name>

end <pattern name>;
```

There can be more than one invariant. Each invariant is given a unique name. An invariant is specified like a real pattern — it can have multiple states.

The external declarations refer to functions that will be used to perform computations in the pattern. They are covered in detail in Section 5.2.

The pattern name follows the pattern keyword. This must agree with the name at the end of the pattern following the end keyword. This will be the name of the pattern C++ code generated when this pattern is parsed. So in our case, lpr_copy_files will generate lpr_copy_files.C which will be compiled to lpr_copy_files.so. Note, the pattern name and the name of the file containing the pattern do not have to agree. The pattern name is what is used.

### 4.2.2 State declarations

The state declarations section is where the states of the pattern are listed. This is similar to declaring a variable — the state keyword specifies that the variable names following it are actually states in the IDIOT pattern. For example,
specifies three states — the start state, and two other named states. These names are used in the transition section to specify what transitions connect what states. Note, the start state does not have to be explicitly denoted as such — when the transitions are parsed, the start state is deduced.

States can be declared nodup. This means that tokens will not be duplicated when they move from that state. Typically, a token which satisfies a guard will be duplicated into the state after the guard, leaving a token behind. A nodup state means that the token moves into the new state, and no token remains in the source state. The start state cannot be nodup’d. To declare a state nodup do:

```plaintext
nodup state this_is_a_nodup_state;
```

### 4.2.3 Token color declarations

Tokens have bindings associated with them that can carry values along as the token moves through the machine. For example, in our pattern, we might want to record the user id of the person who executed the `lpr` command and the process id of the command itself. We declare two variables to store these values: `RUID` and `PID`:

```plaintext
int RUID, PID;
```

These variables can now be used in the guard expressions. Once assigned to, they will store a value that is bound to that token as it moves through the state machine — consider them as “local variables” to a token. Each token will have a unique pair of these variables as every execution of the `lpr` command will have a unique PID assigned to it.

How are these variables set and used? We will discuss this when we describe transitions.

### 4.2.4 The post action

The post action code is executed when the pattern is matched. It is usually used to inform the system operator that a potential intrusion has occurred, or it could take active steps to halt the intrusion (such as shutting down the system). The post action code is normal C code, and it can reference any of the local variables declared in the pattern.

### 4.2.5 The invariant

Every time a pattern is evaluated, the invariant is also evaluated. This controls the deletion of tokens from the machine. Why is this necessary? Consider our example pattern again. Every time the `lpr` program is executed, a token will move from the start state to the `after_lpr_exec` state. But this invocation of `lpr` may be completely innocent. Thus that token will remain in the `after_lpr_exec` state. Over time, tokens will accumulate in the machine increasing processing time and causing memory overflow. Thus we need a mechanism to delete tokens that are no longer necessary.

For every token that is placed in the start state of the main pattern, a token is placed in the start state of the invariant state machine. Events drive the tokens through both the main pattern and the invariant pattern. If a token reaches the final state in the invariant state machine, the corresponding token, and all its
descendents, are deleted from the main pattern state machine. To do this, IDIOT maintains a link between tokens in the main pattern machine and those in the invariant machine. This deletion happens automatically.

An invariant is specified using the same syntax as a normal pattern transition. In our example, there are only two states to the invariant machine, with one transition between them. The following code shows the invariant:

```python
neg invariant first_inv /* negative invariant */
state start_inv, final;

trans exit(EXIT)
  <- start_inv;
  -> final;
  \{ PID = thid(PID); \}
end exit;
end first_inv;
```

The transition is named exit and it leads from state start_inv to state final. These states are declared prior to the transition specification. This transition is taken when an EXIT event occurs in the audit trail — when a process has exited. The PID attribute is available in this event, and it gives the process id of the exiting process. Our invariant guard states that any token whose PID color matches the PID attribute from this event will be deleted. In simple terms, all tokens created by this process will have the same PID value. When the same process exits, we will delete all these tokens using this invariant. The guard expression in the \( \_ \) section will be explained in Section 4.2.7.

This is a very common invariant. A lot of patterns will delete tokens associated with a process once that process exits. Another common invariant is deleting tokens that no longer can form part of an intrusion attempt. For example, in the other_C2_patterns/3-failed-logins pattern, the invariant deletes any tokens that have been in the system for longer than 3 minutes. This pattern looks for 3 failed login attempts within the space of 3 minutes, so keeping tokens around for any longer is pointless.

```python
neg invariant inv
state start_inv, final_inv;

trans clk(CLK)
  <- start_inv;
  -> final_inv;
  \{ thid(TIME) - time > 100; \}
end clk;
end inv;
```

The CLK event has not been implemented yet.

### 4.2.6 Unification - binding values to tokens

Before we can discuss transitions and guards, the concept of unification must be discussed. IDIOT uses unification to test guard expressions. An expression is unifiable if the value on the left hand side and that on the right of the = sign can be combined. The reader is referred to standard textbooks on this topic: [Set89] for example. Simply put, an expression of the form \( x = y \) is unifiable if \( x \) and \( y \) both have the same value. If one of \( x \) or \( y \) does not have a value bound to it, it assumes the value of the other variable. This allows assignment to occur, without requiring unique assignment and equality operators.

### 4.2.7 Pattern transitions

After the invariant comes a list of the transitions. There is no special order to each transition, as IDIOT computes the placement of the states based on the transition specifications. Each transition is contained within a trans ... end block.
A transition is associated with a particular audit event. In our example pattern, the two transitions are associated with EXECVE and CREAT respectively. A transition will only be taken if its corresponding event occurs, and its guard is satisfied.

A guard is a boolean expression that must be satisfied before a token can transit a guard. The boolean expressions in IDIOT are expressed in a C-like syntax. They are evaluated left-to-right with short circuit evaluation. This means that if a component of the guard causes the whole guard to evaluate to false, evaluation halts. Most guards are specified in conjunctive normal form — a conjunction of clauses. Conjunction is specified using the AND operator, which is && in C (and IDIOT).

The guard for the transition exec.lpr looks as follows:

```c
- (this[err] = 0  &&  pid = this[PID]  &&  prog = this[PROG]  &&
    ruid = this[RUID]  &&
    intexec("./lpr", this[PROG]) = 1  &&  this[RUID] = 0;)
```

This guard is composed of six clauses each separated by an && operator. The full guard is only true if each of the clauses are individually true. If any clause evaluates to false, the value of the conjunction of these clauses is false, so evaluation halts and the guard evaluates to false.

The first clause checks the ERR attribute of the event. This is a very common test, and can be seen in just about every guard. If ERR is 0 the event occurred successfully, if it is 1 the event failed for some reason. So if we had this[ERR] = 1 in the clause, we would be testing for failed EXECVE events. In the clause, this refers to the attributes of the current event.

The second clause extracts the PID attribute from the audit record, and binds it to the current token. This now becomes a local variable available to all other guards in other transitions. Effectively, we have tagged this token with the process id of the process that caused the token to enter the state machine.

The third clause does the same but for the name of the program which is stored in the PROG attribute of the EXECVE event. This is the full pathname of the executable associated with this event. Again, we bind this to a token color (i.e. a local variable to the token) called PROG.

Similarly, the fourth clause extracts the real user id of the process that caused the audit event and binds it to the token color RUID. This will be used later in the post action to identify which user attempted to copy over a file. This is a common occurrence in patterns — some identification information is extracted from the events and bound to a token color. This information is usually process id, user id or file name information. This can be used to pinpoint who attempted an intrusion, and what they were using or trying to access.

These bindings are actually used in the second transition. The guard for the second transition looks as follows:

```c
- {
    this[uid] = 0  &&  this[PID] = pid  &&  file = this[CRED]  &&
    intexec("/var/spool/lpr", this[CRED]) = 0;
}
```

The == operator is not used in IDIOT. Instead, the = operator is used to unify the values on the left and right of the operator.

Notice how the order of the terms around the = sign is reversed. We are now unifying in the opposite direction. Each clause will evaluate to true if the value of the specified attribute from this audit event matches the corresponding color for the token. IDIOT will match the PID attribute for this event with the PID color for every token in the after_lpr_exec state. This color was bound in the first transition.
This enables us to tie audit records together that belong to the same process. If another process called the create() system call, there would be a CREAT event in the audit trail — but the PID field would not match the token's PID field. Only the process that first introduced the token into the state machine will match the PID field.

We then bind the file name in the OBJ attribute of the event to the token color FILE. This is used in the post action to identify which file is the target of the attack.
Chapter 5

Technical Details

5.1 How IDIOT works

As with most sizable projects, IDIOT consists of a fairly complex file structure. Determining how the components of the program integrate and work together can be a challenging task. This section is presented as an overview of the structure of IDIOT, discussing its major components and how they cooperate to form an intrusion detection system.

IDIOT consists primarily of four components: the audit trail, showaudit.pl, the C2.Server, and the pattern descriptions. Of these four, both the audit trail and showaudit.pl are machine dependent, while the C2.Server and patterns are portable. A fifth component, C2.appl, provides an interactive user interface to IDIOT; this is also portable.

5.1.1 Audit Trail

While technically quite separate from IDIOT, the audit trail is obviously an extremely important part of this intrusion detection system. Without an audit trail, there would be no record of activities against which patterns could be matched, and this system would be rather useless. Currently IDIOT has only been used with Solaris 2.4 and the Sun BSM audit trail. However, the system design should work equally well with virtually any other OS and audit trail. Difficulties arise from the fact that, while most operating systems provide some manner of audit trail, they each have a different format. This leads to portability problems. IDIOT deals with this problem by working with a canonical form of the audit trail, rather than the raw audit trail itself.

5.1.2 showaudit.pl

showaudit.pl is IDIOT's solution to handling different audit trail formats. The current showaudit.pl is simply a PERL script that converts a Sun BSM audit trail into the canonical format necessary for IDIOT. If IDIOT is moved onto a new platform, only showaudit.pl need be rewritten to accommodate the new audit trail format. Similarly, if IDIOT were to be expanded on an existing system to examine more detailed
information from the audit trail, showaudit.pl would need to be extended to include the new information in the canonical audit trail.

showaudit.pl accepts either raw binary input files or ASCII files generated using the praudit command. It can be run from the C2_app interpreter (discussed later) to parse audit files, or it can be run manually to view an audit trail in canonical form. By utilizing command-line options, the audit trail can be printed either with or without symbolic names for events.

As the inner workings of showaudit.pl are discussed in Section 5.3, we will not go into any more detail here.

5.1.3 C2_Server

The C2_Server is the core of IDIOT. It is actually a C++ class, an instance of which is instantiated to perform the intrusion detection. An object of type C2_Server has several methods associated with it; the most interesting of those are listed here:

```c
int run_praudit(char *audit_file) — primary method, described below
C2_Pattern *parse_file(char *patternfile) — parse an IDIOT pattern file
C2_Pattern *dllink_file(char *file) — dynamically link a compiled pattern description into the server object
```

As shown above, run_praudit() simply takes the name of the audit file as input. run_praudit() calls showaudit.pl to transform the audit trail information into canonical form. Then it goes into a loop, reading one audit event at a time from the transformed audit file. If run against a static audit trail, this loop continues until it reaches end-of-file. Otherwise, run_praudit only exits upon encountering some kind of failure.

For each event, run_praudit() steps through the list of patterns which are requesting events. For each of these patterns, it executes the method PatProc(), passing the current event as a parameter. PatProc() is a method for each pattern, and thus knows the current state of matching for that pattern. It takes the current event and performs the operations that should result from the occurrence of that event. These include operations such as token unification and transition firing.

5.1.4 Patterns

Finally, of course, the patterns play a major role in the workings of IDIOT. Patterns are written in a language that describes a known method of attack as a set of states and transitions between them. As indicated above, transitions are based on audit trail records, or events.

To be used by IDIOT, patterns must translated into C++, compiled into shared objects, and linked into the C2_Server object. This can all be done at startup time, or patterns may be added to the server dynamically.

Figure 5.1 depicts the basic component structure of IDIOT.
Figure 5.1: Structure of IDIOT
5.1.5 C2.appl

C2.appl provides an interactive interface into IDIOT. It allows the user to compile patterns, dynamically link pattern objects and instantiate server objects, as well as several other related activities. Listed below are some of the major commands available:

- **parse** — parse a pattern file
- **dlink** — dynamically link a compiled pattern into the server
- **server** — perform operations on the server (set infile to binary or ascii, exercise patterns, change debug level, or print the server queue)
- **debug** — debug a pattern description
- **run** — initiates run.praudit, described above
- **time** — display time accumulated in C2.appl

5.2 Referring to external functions

External utility functions are used to compute results that cannot be expressed in IDIOT pattern form. For example, the pattern lpd.delete.files checks to see if a file is being deleted outside of a specific directory tree. A function called inTree will check to see if a given filename is below a specified directory point. It returns TRUE if this is so.

How can this be incorporated into an IDIOT pattern? The following example shows how:

```c
extern int inTree(char *dir, char *file);
pattern lpd_delete_files 'lpd deletes files not under spool dir' priority 7
state start, after_lpd_exec, after_delete;
set PROG, FILE;
......
    if (strcmp(dir, "") == 0 && strcmp(dir, FILE) == 0 && FILE == name("delete", "")
        inTree("/usr/spool", FILE) = 0;
        ...
```

The function inTree is declared external to the pattern. The declaration states that it takes two string arguments, and returns an integer. When the function is used, we pass two string arguments: /usr/spool and the token attribute FILE.

The extern keyword behaves similarly to C, but there are slight differences. The external declaration is parsed by IDIOT to see if it matches an internal type. If we examine the actual declaration for inTree, we see it takes two strings as arguments, which is declared in utilities.C as:

```c
int inTree(char *dir, char *file);
```

However, this will cause an error in IDIOT. Instead, the built in string type is used: str. IDIOT will parse this and emit C++ code as follows:
extern int inTree(Str , Str );

This uses the inbuilt string class which is then translated into a RWString.

The allowable parameter types for external declarations are as follows:

1. int Integer type — equivalent to C int.
2. bool Boolean type — equivalent to C int.
3. str String type — equivalent to C char *.

NOTE it is important that any external C function be used to compute data that will not change over time! For example, if an external function was written which computed the number of users on the system, an IDIOT pattern which was parsing an audit trail and called that function would probably get a different result every time it was run.

This problem occurs because IDIOT patterns get all their information about the system from the audit trail. If an external function computes a value that changes over time (such as number of processes, average system load, disk utilization) then the values being returned by the external function are not what they would have been at the time the audit trail was generated. Instead, they will reflect the system state when the pattern is run, which may be completely different from the state when the audit file was generated.

For example, see the pattern other_C2_patterns/dont-follow-sym-links. This pattern uses a function islink() to see whether a specified file is a symbolic link. However, the state of the file system may have changed between the time when the audit data was generated and the time the pattern is run.

In all the patterns provided with IDIOT, the utility functions perform computations that cannot be easily done in a pattern specification. For example, a pattern can check to see if a file name exists in a specified path component (this is a string match — it does not check the state of the file system). Or it can see if a file permission mask has the execute bit set (a simple bit-wise computation).

5.3 Audit trail canonicalization

A problem faced by any intrusion detection system is portability — every vendor and system has its own unique audit trail format. Indeed, this format often changes across OS families from the same vendor (SunOS vs. Solaris, for example). A well designed intrusion detection system should be able to accommodate multiple audit trail formats with little modification.

5.3.1 The role of showaudit.pl

IDIOT achieves this platform independence by splitting the intrusion detection engine into two parts: a front end tool that reads a system-dependent audit trail and generates a platform independent intermediate form and a back end that performs the pattern matching. The showaudit.pl script is a PERL [WS92] script that converts a Sun BSM audit trail [Sun] into a canonical audit format that IDIOT can handle. The back end pattern matching engine will take this canonical form and run the patterns against it. If the IDIOT
system is ported to a new system with a different audit trail format, only the showaudit.pl script has to be rewritten. Similarly, if IDIOT is expanded on an existing system to monitor more detailed information from the audit trail, the showaudit.pl script has to be extended to include this new information in the canonical audit trail.

The showaudit.pl script can accept either raw binary input files or ASCII files generated using the praudit command. Typically, showaudit is run from the C2_appl interpreter to parse audit files when running IDIOT patterns. However, it can also be run manually to view an audit file in canonical form. The script will run the praudit script to preprocess the audit trail. If the -r option is given, the audit trail is printed in raw format — no symbolic names for events are printed. The praudit command runs faster in raw mode. By default, the script prints symbolic names for events.

As an example, the output when the script is run with the audit trail for writing-to-executable-files is:

```
execve The Feb 20 18:53:14 1996 mcrobie mcrobie mcrobie mcrobie -2 0 20005 failure: No such file or directory -1 = ARARRAY(0xd554) 0
execve The Feb 20 18:53:14 1996 mcrobie mcrobie mcrobie mcrobie -2 0 20005 failure: No such file or directory -1 = ARARRAY(0xd554) 0
execve The Feb 20 18:53:14 1996 mcrobie mcrobie mcrobie mcrobie -2 0 20005 success 0 = ARARRAY(0xd554) 191000
open - read mcrobie mcrobie mcrobie mcrobie mcrobie -2 0 20005 success 4 - ARARRAY(0xd554) 192005
exec The Feb 20 18:53:14 1996 root root root root -2 0 10116 success 0
```

As can be seen, the script prints a symbolic name for events — for example, execve is the execve( ) system call used to overlay a process with an executable image. Parameters to the system call are displayed after each event name. Identification information for the user calling the command is also printed. In this case, the user id, effective user id, group id and effective group id is printed. This is described in more detail in the Sun BSM audit trail documentation. In contrast, the raw output from showaudit.pl -r looks as follows (for the same audit trail):

```
23 024650414 727 727 727 727 -2 0 20005 2 -1 -
  /struct/kernel/macbsd/macbsd/bsd/audit/exploit 0
23 024650414 727 727 727 727 -2 0 20005 2 -1 -
  /struct/kernel/macbsd/macbsd/bsd/audit/exploit 0
23 024650414 727 727 727 727 -2 0 20005 0 2 -
  /home/microbie/microbie/audit_data/exploit 191000
60 024650414 727 727 727 727 -2 0 20005 0 4 -
/home/microbie/microbie/audit_data/exploit 191000
1 024650421 727 727 727 727 -2 0 20005 0 0 -
2 024650421 0 0 0 0 -2 0 10116 0 0 -
```

The same information is encoded in the raw format as the ASCII format, but the symbolic names are replaced by their corresponding integer codes from the audit trail. For example, the last line shows a user and group id line of 0 0 0 0 which corresponds to the root user and group id.

### 5.3.2 Operation of showaudit.pl

The script takes three command line arguments: -i controls what type of audit file format showaudit expects. -i ascii processes ASCII audit trails and -i binary processes binary audit trails. -f makes it follow the audit trail as it is generated.
The script starts by opening a pipe to a command to parse the raw, kernel generated audit file. This is usually a `tail -f` of the output of `praudit` run on the command-line supplied audit filename. The audit file is then parsed record by record by the `showaudit` script and each record is printed in canonical form to stdout. When `showaudit` is used from within C2_app1, this output is redirected into the C2_Server to drive the IDIOT patterns.

An audit trail is composed of a series of records each of which is a sequence of tokens. Figure 5.2 shows a typical audit record layout. These tokens contain information such as the time the audit record was generated, the event it represents and user IDs associated with the audit event. Before the actual event token is decoded, the surrounding header and informational tokens must be decoded. `showaudit` processes each type of token as described below, and then it generates output for the actual audit event in the audit record.

![Figure 5.2: Layout of a typical audit record](image)

The script must handle binary and ASCII input formats so the `$opt_r` flag is checked while parsing audit record tokens. If the flag is true, the numeric value associated with an event is tested, of not, an ASCII string can be tested. For example, the file token gives the pathname to the next audit file. It's numeric event number is 17, and its ASCII string representation is "file". The code to determine whether the current record is a file record looks as follows:

```bash
if ($opt_r && $[0] == 17) || ($opt_r && $[0] eq 'file')) then
    print 'file

    print 'No of dropped events = $dropped_events', '

    exit 0;
}
```

`showaudit` handles the following types of audit tokens:

**File token** — a file token indicates that the current audit trail has ended and it points to where the next audit file can be found. `showaudit` will parse this token and extract the file name for the new audit file and make three attempts to open the new audit file. If it cannot open it after 3 attempts, it gives up and exits. Furthermore, a file token must mark the **start** of a new audit file — if `showaudit` does not find a file token, it complains that the audit file is invalid and gives up.
**Header token** — a header token marks the beginning of an audit record. A corresponding trailer token marks the end of the audit record. The header token encodes the length of the audit record, the event type the record encodes and a timestamp for when the audit record was generated. `showaudit` extracts the type of the audit event and the timestamp from the header token. It then starts to parse the remaining tokens in the audit record.

**Path token** — a path token contains access path information for an object (typically a file). The pathname is encoded as a length field and an arbitrary length character string. `showaudit` stores the pathname component. There can be multiple path tokens if a link is being followed.

**Attribute token** — an attribute token contains information from the file’s i-node entry. This includes the user and group id of the file’s owner, the file system the file resides on, the inode for the file and the device id for the file system device. `showaudit` extracts this information.

**Argument token** — an argument token contains system call argument information. `showaudit` extracts the argument values. Note, there can be multiple argument tokens — one for each argument to a system call.

**Subject token** — a subject token describes a subject (i.e. a process). The fields of interest to `showaudit` are the real and effective user and group id’s, the process id of the process that generated the audit event, and an audit session id.

**Socket token** — this records information about an Internet socket. `showaudit` extracts the type, local port and address and remote port and address information from the token.

**Return token** — this stores the return value of a system call and the returned error code for the system call (the `errno` value in UNIX). `showaudit` extracts both these fields.

**Text token** — this contains an arbitrary text string which is extracted.

Once the header and informational tokens have been processed, `showaudit` has enough information to start generating a canonical form for the audit record. It calls the `print_base_c2_record` routine to print out the audit record in canonical form. This routine does not handle every audit event — a variable dropped events records how many events were ignored from the audit trail. For each audit event type, it calls `print базе_c2_record` to print the basic information about the audit record. This routine prints the user and group id’s, audit session id, process id and the return value and error number from the system call.

Then the `print_c2_record` prints information about certain audit events. This uses the information stored while parsing the tokens. For example, the code to print a `chmod()` event looks as follows:

```c
C2_10:
    print_base_c2_record();
    chmod has 3 args, path, inode & the newmode on the file, an type Ox...
    $arg1('path') = ['<*>'] if : exists $arg1('path');
    print ' ', $arg1('path')[10];
    if (exists $arg1('obj'))
        if (print * '
', $arg1('obj')[1][inode]);
    else
        if (print * 0';
    print ' ', $arg1('args')[1][0];
    print '
': return;
```
The C2_10 is the branch of the case statement that handles this audit record type. The path and inode information was stored earlier in the associative array `arg` when the tokens were being parsed. These fields are now accessed and printed out, separated by spaces.

5.4 Adding new audit events to IDIOT

IDIOT does not handle all audit events in the Sun BSM trail. It is possible to enhance IDIOT to extract information about audit events and provide extra information to patterns. Before attempting to modify IDIOT to add new events you should read all the documentation shipped with IDIOT and (for Solaris machines) the Audit Record Description section of the BSM answerbook.

There are three components of the IDIOT system that must be modified to handle a new audit record. 1) The `showaudit.pl` script must parse the audit trail and extract the relevant information out of the audit record tokens; 2) The C2_Server module must know how to interpret the data and generate the correct class; 3) The C2_Server::parse method defined in `pat.y` must add the event names to the parsing symbol table.

To add a new event (or add attributes to an existing event) edit the `showaudit.pl` file and remove the event you are going to add from the dropped events list (if it's there) and add the event to the appropriate group. Events are grouped together according to the information they print. All events print the base record. Groups of events print additional information. The event number can be extracted from the `/etc/security/audit_event` file. For example, the `link` system call is event number 5. The entry in the `audit_event` file for `link` is:

```
5:AUD_LIM:link(2):fe
```

Then edit the `C2_events.h` file and add the code for your event (or the attribute you added). As in `showaudit.pl`, events are grouped into classes of similar events. A GENERIC event in this file is only a class definition that encompasses many events. For example, the class "C2Event_GENERIC_EXEC" represents all execute events that have the same format. The individual instances are defined by the lines:

```c
typedef C2Event_GENERIC_EXEC C2Event_EXEC;
typedef C2Event_GENERIC_EXEC C2Event_EXECV2;
```

Similarly, the class "C2Event_GENERIC" represents all the audit events that share the same format but are not a logical group (similar to execs). From the declaration:

```c
typedef C2Event_GENERIC C2Event_CLOSE;
typedef C2Event_GENERIC C2Event_ACCESS;
typedef C2Event_GENERIC C2Event_CHDIR;
typedef C2Event_GENERIC C2Event_FORK;
typedef C2Event_GENERIC C2Event_LSTAT;
typedef C2Event_GENERIC C2Event_STAT;
typedef C2Event_GENERIC C2Event_UNLINK;
typedef C2Event_GENERIC C2Event_Vfork;
```

we gather that the events CLOSE, ACCESS, CHDIR, FORK, LSTAT, STAT, UNLINK, and Vfork have been grouped together by the `showaudit.pl`.

Finally, edit the `pat.y` file and search for the `C2_Server::parse` method. It should resemble:
Add another line to this routine to push the name of the event you are defining.

5.4.1 Events Supported in Shipped Version

The following tables have a list of the events that can be used in IDIOT patterns with the system as shipped.

<table>
<thead>
<tr>
<th>Events: EXEC, EXECVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes:</td>
</tr>
<tr>
<td>TIME (time), RUID (uid), EUID (uid), RGID (uid), EGID (uid), AUID (uid), SID (uid), PID (int), ERR (int), RETVAL (int), PROG (const char *), PROG_INODE (int), OBJ_MODS (int)</td>
</tr>
<tr>
<td>Description:</td>
</tr>
<tr>
<td>TIME: Time the event took place</td>
</tr>
<tr>
<td>RUID: Real user ID</td>
</tr>
<tr>
<td>EUID: Effective user ID</td>
</tr>
<tr>
<td>RGID: Real group ID</td>
</tr>
<tr>
<td>EGID: Effective group ID</td>
</tr>
<tr>
<td>AUID: User audit ID</td>
</tr>
<tr>
<td>SID: Session ID</td>
</tr>
<tr>
<td>PID: Process ID</td>
</tr>
<tr>
<td>ERR: Return status of system call</td>
</tr>
<tr>
<td>RETVAL: Process return value</td>
</tr>
<tr>
<td>PROG: Name of program executed</td>
</tr>
<tr>
<td>PROG_INODE: Program inode number</td>
</tr>
<tr>
<td>OBJ_MODS: Permissions of program executed</td>
</tr>
</tbody>
</table>
### Events: LINK, SYMLINK

**Attributes:**
- TIME (time_t), RUID (uid_t), EUID (uid_t), RGID (uid_t), EGID (uid_t), AUID (uid_t), SID (uid_t), PID (int),
- ERR (int), RETVAL (int), OLDPATH (const char *), NEWPATH (const char *)

**Description:**
- TIME: Time the event took place
- RUID: Real user ID
- EUID: Effective user ID
- RGID: Real group ID
- EGID: Effective group ID
- AUID: User audit ID
- SID: Session ID
- PID: Process ID
- ERR: Return status of system call
- RETVAL: Process return value
- OLDPATH: Pathname of the object linking to
- NEWPATH: Pathname of new object

### Events: MKNOD

**Attributes:**
- TIME (time_t), RUID (uid_t), EUID (uid_t), RGID (uid_t), EGID (uid_t), AUID (uid_t), SID (uid_t), PID (int),
- ERR (int), RETVAL (int), OBJ (const char *), DEV_MODE (int)

**Description:**
- TIME: Time the event took place
- RUID: Real user ID
- EUID: Effective user ID
- RGID: Real group ID
- EGID: Effective group ID
- AUID: User audit ID
- SID: Session ID
- PID: Process ID
- ERR: Return status of system call
- RETVAL: Process return value
- OBJ: Name of special file
- DEV_MODE: Permissions of the special file

### Events: LOGIN, SU, EXIT

**Attributes:**
- TIME (time_t), RUID (uid_t), EUID (uid_t), RGID (uid_t), EGID (uid_t), AUID (uid_t), SID (uid_t), PID (int),
- ERR (int), RETVAL (int)

**Description:**
- TIME: Time the event took place
- RUID: Real user ID
- EUID: Effective user ID
- RGID: Real group ID
- EGID: Effective group ID
- AUID: User audit ID
- SID: Session ID
- PID: Process ID
- ERR: Return status of system call
- RETVAL: Process return value

**Attributes:**
- TIME (time.t)
- RUID (uid.t)
- EUID (uid.t)
- RGID (uid.t)
- EGID (uid.t)
- AUID (uid.t)
- SID (uid.t)
- PID (int)
- ERR (int)
- RETVAL (int)
- OBJ (const char *)
- OBJ_INODE (int)
- OBJ_MODS (int)
- OBJ_OWNER (int)

**Description:**
- **TIME:** Time the event took place
- **RUID:** Real user ID
- **EUID:** Effective user ID
- **RGID:** Real group ID
- **EGID:** Effective group ID
- **AUID:** User audit ID
- **SID:** Session ID
- **PID:** Process ID
- **ERR:** Return status of system call
- **RETM:** Return status of system call
- **OBJ:** Name of file being opened
- **OBJ_INODE:** Inode number of file opened
- **OBJ_MODS:** Permissions on file opened
- **OBJ_OWNER:** User ID of file owner

---

### Events: ACCESS, CHDIR, CLOSE, FORK, LSTAT, STAT, UNLINK, Vfork

**Attributes:**
- TIME (time.t)
- RUID (uid.t)
- EUID (uid.t)
- RGID (uid.t)
- EGID (uid.t)
- AUID (uid.t)
- SID (uid.t)
- PID (int)
- ERR (int)
- RETVAL (int)
- OBJ (const char *)
- OBJ_INODE (int)

**Description:**
- **TIME:** Time the event took place
- **RUID:** Real user ID
- **EUID:** Effective user ID
- **RGID:** Real group ID
- **EGID:** Effective group ID
- **AUID:** User audit ID
- **SID:** Session ID
- **PID:** Process ID
- **ERR:** Return status of system call
- **RETM:** Return status of system call
- **OBJ:** Name of file being opened
- **OBJ_INODE:** Inode number of file opened
- **OBJ_MODS:** Permissions on file opened
- **OBJ_OWNER:** User ID of file owner
- **OBJ:** Name of object
- **OBJ_INODE:** Inode of object

---

52
| Events: CREAT | Attributes: TIME (time.t), RUID (uid.t), EUID (uid.t), RGID (uid.t), EGID (uid.t), AUID (uid.t), SID (uid.t), PID (int), ERR (int), RETVAL (int), OBJ (const char *), OBJINODE (int), OBJMODS (int) |
| Description: TIME: Time the event took place |
| RUID: Real user ID |
| EUID: Effective user ID |
| RGID: Real group ID |
| EGID: Effective group ID |
| AUID: User audit ID |
| SID: Session ID |
| PID: Process ID |
| ERR: Return status of system call |
| RETVAL: Process return value |
| OBJ: Name of file being opened |
| OBJINODE: Inode number of file opened |
| OBJMODS: Permissions on file opened |
| OBJOWNER: User ID of file owner |
| OBJ: Name of file created |
| OBJINODE: Inode of file created |
| OBJMODS: Permissions of file created |

| Events: CHMOD | Attributes: TIME (time.t), RUID (uid.t), EUID (uid.t), RGID (uid.t), EGID (uid.t), AUID (uid.t), SID (uid.t), PID (int), ERR (int), RETVAL (int), OBJ (const char *), OBJINODE (int), NEWMODS (int) |
| Description: TIME: Time the event took place |
| RUID: Real user ID |
| EUID: Effective user ID |
| RGID: Real group ID |
| EGID: Effective group ID |
| AUID: User audit ID |
| SID: Session ID |
| PID: Process ID |
| ERR: Return status of system call |
| RETVAL: Process return value |
| OBJ: Name of file being opened |
| OBJINODE: Inode number of file opened |
| OBJMODS: Permissions on file opened |
| OBJOWNER: User ID of file owner |
| OBJ: Name of file for which the permissions are being changed |
| OBJINODE: Inode of file being changed |
| NEWMODS: New permissions for file |
### 5.5 A sample IDIOT program

*jig. C* was written as an example of how to run IDIOT non-interactively. It instantiates a C2Server object, sets the desired debug level, loads the specified patterns, and attempts to match all patterns against the specified audit trail.

Following is the usage summary for *jig. C*:

**Usage:** *jig [options] <auditfile> <patternfile>*

**Options:**
- `-d n` -- Set debug level to `n` (defaults to `0` if option not given)
- `-l` -- Link in precompiled patterns (patterns parsed by default)
- Pattern files on command line should be `-so` versions

As can be seen, the current defaults are to generate no debugging information and to parse all patterns. For most cases, it would likely be preferable to have *jig. C* link precompiled patterns by default. This would be a minor change within the code.

It should be noted that *jig. C* was written as an example program and assumes some degree of experience on the part of the user. No error-checking is performed on the command-line arguments; they are assumed to be accurate and complete.

### 5.6 Debugging

The original debugging options for IDIOT were quite simple: debugging was either on or off, and the only debugging information that could be automatically generated came from the server. Pattern debugging
information had to be manually inserted into the pattern itself. We decided that more flexibility would be helpful. There are now three levels of debugging:

Level 1 — Only generate server debugging information

Level 2 — Only generate pattern debugging information

Level 3 — Generate both pattern and server information

We also decided that the ability to separate the pattern debugging information from the server debugging information would be quite useful. A PERL script, view_debug.pl, was developed for this purpose; it is also described below.

5.6.1 Debugging the server

Debugging within the C2_Server is fairly basic. If the debug level is appropriately set, the server generates information about each audit event it processes. This information includes the type of event, time of occurrence, RUID, EUID, PID, return value, and various event-dependent information, such as program name. This information is now only generated when the debug level is set at 1 or 3. Also, each line of debug output has been prepended with a %S to distinguish it from pattern debugging information.

5.6.2 Debugging patterns

This section describes how to debug patterns written in the IDIOT pattern language. As the language is especially tailored to specifying pattern transitions, it is difficult to use regular tools such as gdb to debug patterns. To aid debugging, we have provided some rudimentary output routines. The main utility routine we have provided is true_print(), which takes a string as a parameter, prints the string to STDOUT, and returns TRUE.

As mentioned above, debugging information for patterns originally had to be generated manually. We decided that having some debugging information generated automatically could be useful, if it were governed by the debug level. To achieve this goal, we modified pat.y to insert the following code into every pattern as it is parsed and translated to C++:

```c
int dbg;
extern int true_print(Scr);
...
 dbg = $->debug; // executed within pattern constructor...

 if (dbgl > 1)
     true_print("<pattern name> -- <transition name> transition fired");
```

We decided to only insert calls to true_print() at the completion of each transition to allow the user to maintain a fine grain of control over debugging output.

If more debugging information is desired, there are a couple of options. Note that IDIOT uses short-circuit evaluation of guards, so if any one of the conjunctive clauses is false, then evaluation of the whole clause stops and the transition is not taken. Thus, if a true_print() call is placed within a clause, all the preceding clauses must be true before the print will occur. true_print() itself returns TRUE, which is the identity element for logical-and and has no effect on the evaluation.
For example, if we were interested only in the occurrence of an OPEN.W event (from the writing-to-executable pattern), whether or not it was successful, we could do:

```plaintext
tran modS(OPEN_N)
  <- after_exec;
  -> violation:
  ] (true_print('Matched OPEN.W event...') && event occurred */
    this[ERR] = 0 && /* if this operation succeeded */
    PID = this(PID) && /* and this PID matches that of the exec */
    FILE = this(FILE) && /* remember this filename */
    exec(this[DEL_WARNINGS]); /* if this file is executable */
  end mod;
```

Note that this call to `true_print()` would always be executed, regardless of the debug level. If the statement should be conditional upon the debug level, it could be inserted as follows:

```plaintext
tran modS(OPEN_N)
  <- after_exec;
  -> violation:
  ] ( this_print('Matched OPEN.W event...') && event occurred */
    this[ERR] = 0 && /* if this operation succeeded */
    PID = this(PID) && /* and this PID matches that of the exec */
    FILE = this(FILE) && /* remember this filename */
    exec(this[DEL_WARNINGS]); /* if this file is executable */
  end mod;
```

As with an ordinary call to `true_print()`, this would leave the value of the guard unchanged. In addition, the debugging information would only be generated if the debug level was set appropriately (> 1).

One item to note is that each string passed to `true_print()` is prepended with `%P before being printed, distinguishing it from the server debugging information.

### 5.7 Interactive debugging

The `C2_app1` program can be used to run IDIOT interactively. This can sometimes be useful for debugging purposes. The debug level is initialized to 0. To change the debug level, issue the following command:

```
server debug <debug level>
```

The specified debug level will cause the appropriate debugging information to be printed to STDOUT.

We mentioned above that each pattern’s debug variable is initialized only once, within the pattern constructor. Thus, the debug variable retains the value of the server debug level at the time the pattern was instantiated. This allows the user to dictate which patterns generate debugging information.

For example, if you only wished to debug the executing-particular-pgms pattern while matching several, you would do the following:

```plaintext
> ./C2_app1  //start C2_app1, debug_level set to 0
this> d link <pattern1.so>
this> d link <pattern2.so>
//Link in N patterns without debug info
...
this> d link <patternN.so>
this> server debug 2
this> d link ex_prg_pgms.so
this> run <audit trail>
```

All of the included patterns would be utilized, but only the executing-particular-pgms pattern would generate debugging information.
5.8 Viewing debug information

As an aid in viewing and understanding debugging information, we developed view_debug.pl. This PERL script separates pattern debugging information from server debugging information. Furthermore, it allows both sets of output to be viewed simultaneously with the debugging statements synchronized.

To run it simply type ./view_debug.pl, followed by the command line you wish to have passed to jig. view_debug.pl will execute ./jig with the specified command line and separate the debugging output into the files pat.debug.out (pattern debug information) and C2.debug.out (server debug information). (NOTE: view_debug.pl currently assumes it is in the same directory as jig.) The output is separated according to the presence of the %P at the beginning of pattern debugging output and the %S preceding server debugging output. For each line of output the following occurs:

- if preceded by %P, the line is numbered and written to pat.debug.out, minus the %P, and a blank line (numbered identically) is written to C2.debug.out
- if preceded by %S, the line is numbered and written to C2.debug.out, minus the %S, and a blank line (numbered identically) is written to pat.debug.out
- all other lines are simply printed to STDOUT, without affecting the numbering

After view_debug.pl has finished, pat.debug.out and C2.debug.out may be viewed side by side, allowing the user to see exactly what sequence of events is causing each transition to fire. The line numbers allow easy synchronization while viewing the files.

NOTE: While view_debug.pl will not currently work with the C2.appl interface, this could be changed with a fairly simple alteration of the PERL code.
Chapter 6

Limitations of C2 Audit Trails

6.1 Auditing socket calls on System V (Solaris)

In SunOS, the abstraction of sockets is built into the kernel. Calls such as socket, connect and accept must cross the system call boundary, and will generate audit records.

However, in System V, the communication abstraction is streams. A stream is a data flow path between two endpoint entities. Streams are a generic abstraction for data flow — they are used to implement terminal drivers, sockets and FIFOs in System V versions of UNIX. In particular, the abstraction of sockets is provided by a library built on top of the kernel. Socket calls in this library execute a series of getmsg, putmsg and ioctl calls to interface with System V streams. No audit data is generated that specifically mentions that a socket call has occurred. The audit data only shows the System V stream interface calls.

This makes it difficult to write patterns to detect such simple network based attacks as port-flooding and port-walking. A port-flood is where an abnormally high number of connections are received on a single port. The idea is to use up kernel socket resources and denying network access to any other users. A port-walk is a series of connections to ports which attempts to find services running on ports which may be exploited.

Both of these attacks could be detected from an audit trail which contained records for connect and accept events. To simulate this, a wrapper library for the socket library must be written. This wrapper library will generate an audit record to be placed in the audit file, and will then call the original socket library call.

An example below shows how this is done:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <fcntl.h>

/* the _connect routine is the system call */
extern int _connect(int a, struct sockaddr *name, int namelen);

/* write our own connect routine that will generate an audit record */
int connect(int a, struct sockaddr *name, int namelen) {
    struct socket sock;
    int token;
    token = su_open(); /* allocate an audit token to write info to */

    /* generate a socket token that records the information */
```
6.2 Auditing operations on symbolic links on System V (Solaris)

Many of the patterns devised at the COAST lab needed information about system calls that were operating on symbolic links. The pattern that detects the exploitation of the xterm bug, for example, can detect exploitations by seeing if the inode for the log file has changed between the creation and accessing of the file.

Unfortunately, this pattern cannot be used as is with the Basic Security Module logging provided by Sun with Solaris 2.4. For this pattern to work, the audit trail would need to provide, at least, the following information:

1. execve: ERROR, EUID, RUID, PID, and program name
2. creat: ERROR, PID, initial file name, final file name, and inode number
3. open: ERROR, PID, initial file name, final filename, and inode number
4. close: ERROR, PID, inode, initial file name, and final file name
5. chown: ERROR, PID, inode, initial file name, and final file name
6. chmod: ERROR, PID, inode, initial file name, and final file name
7. access: ERROR, PID, inode, initial file name, and final file name
8. stat: ERROR, PID, inode, initial file name, and final file name

The only items that require explanation are the file names. If we have a symbolic link sym1 that points to a file fill then a call to chmod("sym1", mode) would require an audit trail record that would indicate that the chmod system call was executed on the initial file name sym1 and the final file name fill. The inode should be the one corresponding to the final name.

The audit trail generated by the Basic Security Module provided with Solaris 2.4 violates the file names requirement. To see why this is important, consider the following fragment of a setuid C program called xtermbug:

```c
/* Create a log file called xtermbug.log */
if (create("xtermbug.log", 0666) == -1) {
    /* The permissions of the created file were set to 0666. Change them to something more reasonable */
    if (chmod("xtermbug.log", S_IRWXU | S_IRWXG | S_IRWXO) == -1) {
        printf(stderr, "Could not change the permission of file xtermbug.log\n");
    } else {
        printf(stderr, "Could not create log file xtermbug.log\n");
    }
```

The audit trail for symbolic links on System V (Solaris) requires additional information to accurately detect exploitations.
and an attack script that will exploit the vulnerability described in this document is called `xtermbug.exploit`:

```
#!/bin/sh
mknot /c~r=l>u
...
U=buq
r.
sloop 1
IV
C~rm
.....
jucJ

This attack can not be detected by the pattern described in this document if using the Basic Security Module logging in Solaris 2.4. The audit trail generated for this session, reformatted for clarity and brevity, with the attacker called `krsul`, and the victim called `xtermbug`, is:

```
1. execve(2) - path./home/krsul/xtermbug.exploit - EUID gollum - RUID gollum
2. execve(2) - path./usr/sbin/ashmod - EUID gollum - RUID gollum
3. chmod(2) - argument.2 ./xtermbug.exploit - argument.3 .xtermbug.log
4. path./usr/bin/filestat - EUID gollum - auto gollum

m
5. chown(2) - argument.2 .xtermbug.exploit - argument.3 .xtermbug.log
6. path./usr/bin/filestat - EUID gollum - RUID gollum

m
7. execve(2) - path./home/xtermbug.exploit - EUID xtermbug - RUID xtermbug
8. execve(2) - path./usr/sbin/ashmod - EUID xtermbug - RUID xtermbug
9. execve(2) - path./usr/local/bin/mknot - EUID gollum - RUID gollum
10. rename(2) - path./usr/bin/xtermbug.exploit - RUID gollum
11. path./usr/bin/filestat - EUID gollum - RUID gollum
12. execve(2) - path./usr/bin/mknot - EUID gollum - RUID gollum
13. eyeglass(2) - path./home/xtermbug.exploit - path./usr/bin/filestat
14. chmod(2) - argument.2./xtermbug.exploit - argument.3 .xtermbug.exploit
15. chown(2) - argument.2./xtermbug.exploit - argument.3 .xtermbug.exploit
16. execve(2) - path./usr/bin/filestat - EUID xtermbug - RUID xtermbug
17. execve(2) - path./usr/bin/filestat - EUID xtermbug - RUID xtermbug
18. chown(2) - argument.2./xtermbug.exploit - argument.3 .xtermbug.exploit
19. chmod(2) - argument.2./xtermbug.exploit - argument.3 .xtermbug.exploit
20. execve(2) - path./usr/bin/filestat - EUID xtermbug - RUID xtermbug
```

Notice that in line 20, the audit trail indicates that the `chlnod(2)` system call was made to the file `/homes/krsul/break_me`. While this is ultimately true, it does little to help detect the exploitation of the vulnerability. The pattern design specifically looks for a `chlnod` or `chown` to the same file name as the `creat`, but with different inode numbers. The audit trail should mention that the original call was made to file `xtermbug.log`.

### 6.3 Write events are not audited

Some patterns need to detect WRITE events. However, in the Solaris BSM audit trail the write events are subsumed under the OPEN event. So the audit trail for a write looks as follows (output of `praudit`):

```
header,135,2,open(2) - read.write ,.Tue Feb 21 18:53:34 1995, - #68005000 exec
path./home/acrobat/myDIR/audit_data/executable
attribute,109811,acrobat,acrobat,20005,0 0 0 0 0 0 0 0
subject,-2,acrobat,acrobat,acrobat,acrobat,20005,0 0 0 0 0 0 0 0
return,success,4
```

The audit trail was gathered with the `fw` audit mask flag. This gathers data about file writes. The open call is recorded, but the write call is not. The above audit trail corresponds to the following code fragment:
Thus, we are forced to detect writes to executable files by detecting the opening of the file, and not the actual write.
Chapter 7

Source Code
Bibliography


