Perdix Language Reference Manual

Orr Bibring
ob2135@columbia.edu

Justin Prosco
jp2415@columbia.edu

Bing Wu
bw2236@columbia.edu

Angelika Zavou
az2172@columbia.edu

October 18, 2007

Contents

1 Introduction

This document is a reference manual for Perdix, a high-level query language for the analysis of Linux iptables firewall logs.

2 Anatomy of a Perdix Program

Every Perdix program must have the same basic structure. The program begins with a start statement and the body of the program is contained within braces. File and variable initialization is done at the top of the program body and are followed by conditional query statements and result print options. Finally, input and output files are closed.

```
start()
{
    variable declarations and initialization
    file open operations

    conditional statement { print options }
    ...
    conditional statement { print options }

    file close operations
}
```

3 Lexical Conventions

3.1 Tokens

Tokens area divided into the following classes:

- Constants
- Identifiers
- Keywords
- Operators

Any non-printable character such as a space, tab, new line, and form feed are considered whitespace. Whitespace is discarded by the compiler, but is available to the programmer to make a script more human readable. Whitespace is not used to dictate the scope of variables or program flow, but is used to separate tokens.

3.2 Constants

3.2.1 Integer Constants

An integer constant is a sequence of digits. All integers are positive base 10 numbers and do not need to be enclosed in double quotation marks when being defined.

3.2.2 String Literals

String literals are arbitrary length sequences of characters enclosed in double quotation marks. If a string literal contains double quotation marks, then they are written twice as shown below:

```
string test = "The ""double quotes"" are also part of the string"
```

3.3 Identifiers

Identifiers refer to user-defined variable names. An identifier must begin with a letter or underscore and can contain a series of letters, underscores, or numbers so long as the name of the variable does not match one of the reserved words. Identifiers may contain either uppercase, lowercase, or mixed case letters. All identifiers in Perdix are case sensitive.

3.4 Keywords

Keywords are reserved words within the Perdix language that cannot be used as identifier names.

3.4.1 Perdix Keywords

The following are keywords in the Perdix language:

cond	**int**	**file**	**string**
start	**open**	**print**	**close**

3.4.2 iptables Keywords

The following are keywords that are used to reference fields within an iptables log file:

- **date**: the timestamp of the packet
- **target**: the target rule chain such as ACCEPT, REJECT, etc.
- **mac**: the MAC address of the adapter
- **src**: the source IP address of the packet
- **srcport**: the source port of the packet
- **dst**: the destination IP address of the packet
- **dstport**: the destination port of the packet

- **in**: the interface the packet came in on
- **out**: the interface the packet was sent
- **protocol**: the protocol of the packet
- **tos**: the type of service
- **tcp_flags**: the flags set in the TCP header of the packet
- **ip_flags**: the flags set in the IP header
- **ttl**: the packet time to live
- **window**: the window size of the packet
- **state**: the connection state

3.5 Punctuation

There is a limited amount of punctuation defined in the Perdix grammar:

- Semicolon (;) : signifies the end of a Perdix statement
- Comma (,): separates the output arguments passed to print function
- Colon (:): used in functions to separate a file identifier argument from other arguments
- Double Quotation Marks (" "): indicate a string literal
- Braces ({ }) : define the body of a Perdix statement
- Parentheses (( )): used to contain the arguments passed to built-in functions. They can also be used to enforce precedence in conditional statements.
- Pound (#): signifies the beginning of a programmer comment

3.6 Comments

Comments are a single line beginning with the `#` character. Anything following a `#` character up to the newline character is assumed to be a comment and will be discarded by the compiler. Multiline comments are not allowed, as each line must begin with the `#` character. An example of this is shown below:

```
# This is a comment
# This is the second line of the comment
int f=5;  # Comment at the end of a line
```

4 Data Types

4.1 int

The int type is used to declare integer variables. Due to the nature of iptables log files, there is no need for negative values, as there should never be a negative integer value in an iptables log file. Constant values assigned to variable of this type do not need to be enclosed in double quotation marks.

4.2 string

The string data type is used to define string literals. Variables of this type can be assigned a value of any arbitrary length character string as long as it is enclosed in double quotation marks.

4.3 file

The file data type is a special type of string that is used to represent file variables. The value of a file variable should be an operating system-specific path to a file, including the file name, enclosed in double quotation marks. Therefore, file data types are defined with string literal values.

5 Expressions

Expressions are combinations of values, variables and operators that are evaluated according to the particular rules of precedence and associativity defined in the Perdix language. Common locations for expressions are in the `cond()` statement as well as in the initialization of variables. There are four types of expressions found in Perdix, arithmetic expressions, conditional expressions, assignment expressions and logical expressions.

5.1 Arithmetic Expressions

Arithmetic expressions can be used in Perdix utilizing the addition and subtraction operators. These expressions can occur during an integer identifier initialization. A common use of arithmetic expressions is to assign a variable the modified value of another previously defined variable as shown below:

```
int var1 = 200;
int var2 = var1 + 30;
```

5.2 Conditional Expressions

Conditional expressions are relational expressions that involve the use of the >, <, >=, <=, ==, or != operators. The lvalue of all conditional expressions will be one of the iptables keywords, as a conditional statement implies putting a constraint on the query for a particular field within a log file.

Conditional expressions do not return a value. Rather, the lvalue of the statement is changed at runtime to the value of the field specified on the current input line. For instance, if the current line of the log file being read has a `src` value of "192.168.100.1" then a conditional statement including the line `src==` will be changed to "192.168.100.1" at run-time for this line.

If the boolean expression result is true, then the current line of the input log file will be included in the query results. If the statement evaluates to false at run time, the current line of the input log file is not included in the query results.

Examples of conditional expressions are:

```
dst == "192.168.100.1";
dstport >= 80;
```

5.3 Assignment Expressions

Perdix programmers have the ability to define variables at the top of every program. All variable declarations must immediately follow the open brace ("{") after the `start()` statement of a program. Variables must follow the naming conventions of identifiers. Additionally, variables may not be assigned to another variable if the second variable has not yet been initialized with a value.

Examples of assignment expressions are:

```
variable1 = 80;
variable1 = variable2;
```

5.4 Logical Expressions

Logical expressions are a sequence of one or more conditional expressions that are tied together by one of the logical operators. To include all conditional statements, the logical AND operator is used. To include one or more conditional statement, the logical OR operator is used. Finally, to negate all the conditions specified from a query, the NOT operator is used.

An example of a logical expression is:

```
(dst == "192.168.100.1" && dstport == 80) || !(protocol == "udp")
```

6 Operators

There are three classes of operators available in Perdix: arithmetic, relational, equality and logical. When expressions involving operators are evaluated at run time, a return value of true means the current line from the input file should be included in the output of the query. If the statements evaluate to false, then the the query does not return those lines that produced the false result.

6.1 Arithmetic Operators

Perdix contains addition and subtraction arithmetic operators:

- **Addition**: *lvalue + rvalue*
  The result of expressions using this operator will be the sum of the lvalue and rvalue.
- **Subtraction**: *lvalue - rvalue*
  The result of expressions using this operator will be the lvalue minus the rvalue.

6.2 Relational Operators

Perdix contains the following relational operators:

- **Greater Than**: *lvalue > rvalue*
  Evaluates to true when the lvalue is strictly greater than the rvalue.

- **Less Than**: *lvalue < rvalue*
  Evaluates to true when the lvalue is strictly less than the value of the rvalue.
- **Greater Than or Equal**: *lvalue >= rvalue*
  Evaluates to true when the lvalue is greater than or equal to the rvalue.
- **Less Than or Equal**: *lvalue <= rvalue*
  Evaluates to true when the lvalue is less than or equal to the rvalue.

6.3 Equality Operators

The following equality operators are defined in Perdix:

- **Equal** : *lvalue == rvalue*
  Evaluates to true when the lvalue equals the rvalue.
- **Not Equal**: *lvalue != rvalue*
  Evaluates to true when the lvalue does not equal the rvalue.

6.4 Logical Operators

Logical operators are also defined in order to add or subtract results from the returned queries. These operators evaluate to a boolean value of 1 (true) if the constraints specified are matched, and 0 (false) if they are not. The following logical operators are defined:

- **Logical And** (`&&`) : *conditional_expression* `&&` *conditional_expression*
  Evaluates to true when both the lvalue and rvalue are true
- **Logical Or** (||): *conditional_expression* || *conditional_expression*
  Evaluates to true when either the lvalue and rvalue are true
- **Negation** (`!`): `!`*(conditional_expression)*
  This operator negates the rvalue of the statement it precedes. If the statement after the negation operator evaluated to true, then its negation will be false. Likewise, if the statement after the negation operator evaluated to false then the output will be true. It should be noted that the rvalue must be enclosed in parentheses for clarification.

6.5 Precedence

All statements in Perdix are evaluated from left to right and are left-associative. Parentheses may be used in logical expressions to force precedence.

The precedence of operators are organized in the table below from highest to lowest.

Precedence	Operators	Description	Associativity
1	( )	Parentheses	left to right
2	+ , -	Addition and Subtraction	left to right
3	=	Assignment	left to right
4	> , < , <= , <=, == , !=	Relational and Equality Operators	left to right
5	!	Logical Not	left to right
6	&& , ||	Logical And, Or	left to right

7 Statements

This section describes the statements that are defined in Perdix. Since the language does not allow for user-defined functions, the built-in functions available to programmers are described below.

7.1 Start

```
start() { perdix_body }
```

The `start()` statement signifies the start of a Perdix program. All Perdix programs begin with this statement as the first line (comments excluded). This statement should only appear once in any Perdix program. The `perdix_body` contains the body of the script.

7.2 Variable Declaration

Variable declarations are assignment statements that occur at the top of the `peridx_body`. Variable names must begin with a letter or underscore and can contain a series of letters, underscores, or numbers so long as the name of the variable is not one of the Perdix reserved words.

Variables must be given a data type and also initialized during declaration and cannot be referenced until doing so. Initialization must occur in the same line where the variable is initially declared. The syntax for this is the same as the syntax for an assignment statement, but with an added data type declaration:

```
<data type> <identifier> = <string literal, number, identifier, arithmetic expression>;
```

Variables may not be assigned to another variable if the second variable has not yet been defined with a value. In addition, only one variable can be declared per line. Some examples of variable declaration are:

```
int variable1 = 20;
string variable2 = "192.168.100.100";
int variable3 = variable1;
```

7.3 Assignment Statement

```
<identifier> = <string literal, number, identifier, arithmetic expression>;
```

Variables are assigned values by using the assignment operator. A variable may be defined as either a number, string literal, another previously defined variable or using an arithmetic expression.

7.4 Open

```
open(file: file_perm);
```

The `open()` statement opens an existing file. This statement takes two arguments, both of which are required. The first argument is a file identifier, specifying the path of the file to be opened. The file variable should be defined before using it within an open statement. The second argument is a file permission mode. The arguments must be separated by a colon.

7.4.1 File Permission Modes

There are two file permission modes in Perdix, "read" and "write." These modes use the values `r` and `w` respectively. While they are not keywords, the built-in functions that use these modes specifically check whether or not they have been supplied as an argument to the function call. Only a single permission mode can be supplied to a function at a time.

7.5 Condition

```
cond(file: conditional_statements) { print(file: iptables_keywords); }
```

All query statements in Perdix are defined with the `cond()` statement. This statement takes a file to read from as the first argument and then a query statement as the second argument. The file argument is required, but the `conditional_statement` argument is optional. Directly following a `cond()` statement is a set of curly braces in which a `print()` statement is enclosed. The print statement specifies how the matches from the query statement in the file should be displayed.

7.6 Print

```
print(file: iptables_keywords);
```

The print statement defines how the output of a `cond()` statement should be displayed. The print statement takes two arguments, a file to write the output to and a comma-separated list of keywords, which choose the fields that are displayed on the output. Both arguments are optional, and if none are specified, the default behavior is to print all fields to standard output. Again, the colon must also be included even if both arguments are left out.

7.7 Close

```
close(file);
```

The `close()` statement closes a file. This statement takes a single, required argument, the file identifier to close.

8 Scope

All Perdix programs are contained within a single file and contained within a single `start()` statement block (as defined by the left and right braces). All variables defined are within the same scope. Variables defined at the top of a Perdix program can be used in any statement that accepts variables as arguments.

9 Appendix: Perdix Grammar

The following code is the Perdix grammar that has been designed for use in an ANTLR v2 environment.

```
/*-----------------------------------------*/
/*               PERDIX LEXER              */
/*-----------------------------------------*/

class PerdixLexer extends Lexer;
options {
    k = 2;
    testLiterals = false;
    exportVocab = Perdix;
    charVocabulary = '\3'..'\377';}

/* Punctuation */
LPAREN : '(';
RPAREN : ')';
LBRACE : '{';
RBRACE : '}';
DOT: '.';
COMMA : ',';
DQUOTE : '"';
SEMI : ';';
COLON : ':';


/* Operators */
NOT: '!';
AND: "&&";
OR: "||";

EQ : "==";
NEQ : "!=";
GT : '>';
GEQ : ">=";
LT : '<';
LEQ : "<=";

ASSIGN : '=';
PLUS :'+';
MINUS: '-';

protected LETTER: ('a'..'z' | 'A'..'Z');
protected DIGIT: '0'..'9';

/* Identifiers */
ID options {testLiterals = true;}
: (LETTER |'_') (LETTER | DIGIT | '_')*;

/* Number */
NUMBER : (DIGIT)+;
```

```
/*String*/
LITERAL: '"'! ('"'! '"' | ~('"' | '\n'))* '"'! ;

/*Whitespace*/
WS: (' ' | '\t')+ {$setType(Token.SKIP);};

NEWLINE : ('\n' | ('\r' '\n') => '\r' '\n' | '\r') {
$setType(Token.SKIP); newline();};

/*single-line comment*/
COMMENT: '#' (~('\n'|'\r'))*
{$setType(Token.SKIP);};


/*-----------------------------------------*/
/*              PERDIX PARSER              */
/*-----------------------------------------*/

class PerdixParser extends Parser;

options {k = 3;
buildAST = true;
exportVocab = Perdix;
}

tokens{
/* Keywords for the fields included in each line in iptables log file */
    DATE="date";
    TARGET="target";
    MAC="mac";
    SRC="src";
    SRCPORT="srcport";
    DST="dst";
    DSTPORT="dstport";
    IN="in";
    OUT="out";
    PROTOCOL="protocol";
    TOS="tos";
    TCP_FLAGS="tcp_flags";
    IP_FLAGS="ip_flags";
    TTL="ttl";
    WINDOW="window";
    STATE="state"
}


/* Structure of Perdix programs */
program: main EOF! ;

/* Start function */
main: "start"! LPAREN! RPAREN! body;

body: LBRACE! stmt RBRACE!;
```

```
stmt: (var_decl)+ (open_func)+ (condition_func)+ (close_func)+ ;

/* ---- STATEMENTS ---- */
/* Open Statement */
open_func: "open"! LPAREN! ID COLON! file_perm RPAREN! SEMI! ;

/* Close Statement */
close_func: "close"! LPAREN! ID RPAREN! SEMI!;
file_perm: "r" | "w";

/* Conditional Statement */
condition_func: "cond" LPAREN! ID COLON (selection_body)? RPAREN! LBRACE! print_func RBRACE!;
selection_body: select_condition;

/* Print Statement */
print_func: "print"! LPAREN! print_body RPAREN! SEMI!;
print_body: file_id (arg_stmt)? (COMMA arg_stmt)*;

file_id: COLON | (ID COLON) ;

arg_stmt: iptables_stropt | iptables_intopt ;

/* Variable Declaration */
var_decl: intdecl | stringdecl ;

/* Integer */
intdecl: "int"! ID int_tail SEMI!;
int_tail: ASSIGN! int_initial ;
int_initial: (NUMBER|ID) ((PLUS|MINUS) (NUMBER|ID))*;

/* String */
stringdecl: type ID string_tail SEMI!;
string_tail: ASSIGN! LITERAL;

type: "string" | "file" ;

/* ---- EXPRESSIONS ---- */
/* Logical Expressions*/
select_condition: logic_expr ((AND|OR) logic_expr)*;

logic_expr: (NOT)? (
(LPAREN! select_condition RPAREN!) => LPAREN select_condition RPAREN
| relat_expr);

/* Conditional Expressions*/
relat_expr: (iptables_stropt general_op (ID|LITERAL))
  | (iptables_intopt oper arithm_expr);

/* Arithmetic Expressions */
arithm_expr: arith_factor ((PLUS | MINUS) arith_factor)*;

arith_factor : ID
   | NUMBER
```

```
   | (LPAREN! logic_expr RPAREN!);

str_factor : ID
   | LITERAL;

iptables_stropt: DATE | TARGET | MAC | SRC | DST | IN | OUT |
                 PROTOCOL | TOS | TCP_FLAGS | IP_FLAGS | TTL | WINDOW | STATE;
iptables_intopt: SRCPORT | DSTPORT;


/* ---- OPERATORS ---- */
//logical operators
cond_op: NOT | AND | OR;

/* Operators that apply only to integers */
int_op: GT | GEQ | LT | LEQ ;

/* Operators that apply to all data types */
general_op: EQ | NEQ ;

/* All operators except logical */
oper:  int_op
     | general_op;
```

10 Appendix: Example Programs

This appendix provides example programs in the Perdix language. This section should be used as a reference for available functionality.

10.1 Example 1

This prints /var/log/syslog to the screen.

```
start()
{
    file in = "/var/log/syslog";

    open(input:r);

    cond(input:) { print(:); }

    close(input);
}
```

10.2 Example 2

This prints all entries from /var/log/syslog that have source ip = 192.168.100.1 to the screen.

```
start()
{
    string ip1 = "192.168.100.1";
    file input = "/var/log/syslog";

    open(input:r);

    cond(input: src==ip1) { print(:); }

    close(input);
}
```

10.3 Example 3

This code prints all entries in /var/log/syslog that have source ip = 192.168.100.1 to an output file /tmp/output.

```
start()
{
    string ip1 = "192.168.100.1";
    file input = "/var/log/syslog";
    file output = "/tmp/output";

    open(input:r);
```

```
    open(output:w);

    cond(input: src==ip1) { print(output:); }

    close(input);
    close(output);
}
```

10.4 Example 4

This code prints destination addresses of entries that have source ip = 192.168.100.1 in the file /var/log/syslog to the screen.

```
start()
{
    string ip1 = "192.168.100.1";
file input = "/var/log/syslog";

    open(input:r);

    cond(input: src==ip1) { print(:dst); }

    close(input);
}
```

10.5 Example 5

This code prints only source addresses and source ports to the file /tmp/output, where the destination address is 192.168.100.1 on port 80.

```
start()
{
    string ip1 = "192.168.100.1";
    file input = "/var/log/syslog";
    file output = "/tmp/output";

    open(input:r);
    open(output:w);

    cond(input: dst==ip1 && dstport==80) { print(output:src,srcport); }

    close(input);
    close(output);
}
```