Nathan Steinmann
nsteinma@gmail.com

PERIL Language Reference Manual

1. Introduction

PERIL (Packet Extraction, Reporting, and Identification Language) is a domain-specific language for analyzing a series of IPv4 packets contained in a PCAP-formatted capture file. PERIL programs consist of a single source file organized into the following sections: an optional sequence of global variable declarators, one or more packet tests, one or more conformance rules, and one or more function definitions. See Appendix B for a sample program.

1.1. Program Behavior

In a manner analogous to awk's implicit iteration over lines in a text file, PERIL implicitly iterates over each packet in a capture file. For each packet, all packet tests are evaluated and their results saved. Next, the first conformance rule is evaluated. If true, then execution is transferred to the function whose name matches the conformance rule. Alternatively, if the rule evaluates as false, then subsequent rules are evaluated until one is found that evaluates as true. If all the rules are false, then PERIL iterates to the next packet and repeats the process.

If control is passed to a function due to successful evaluation of a conformance rule, execution continues to the end of that function, at which point an event will be written to standard output. Execution continues with PERIL iterating to the next packet and re-evaluating the packet tests.

2. Lexical Conventions

The PERIL language utilizes several types of tokens: comments, identifiers, keywords, constants, operators, and separators. Separators include tab, space, carriage return, and newline, and are ignored by PERIL except as needed to separate other token types.

2.1. Comments

The characters // denote the beginning of a single line comment, which terminates at the next encountered newline character. Similarly, the characters /* indicate the beginning of a multiline comment, which terminates upon the characters */. Comments do not nest, nor do they occur within string literals.

2.2. Identifiers

Identifiers are a sequence of letters and digits and must begin with a letter or a hash symbol (#). Identifiers beginning with a hash are events, and are used to store metadata from packets. Identifiers are case sensitive and may be of any length.

2.3. Keywords

Keywords are identifiers with special significance to PERIL and may not be used as regular identifiers. The PERIL keywords are: **int**, **string**, **byte**, **bytes**, **EOP**, **SPACE**, and **when**.

2.4. Constants

There are two types of constants: integers and string literals.

2.4.1. Integers

Integer constants may be specified as either decimal or hexadecimal values. An integer constant is hexadecimal if it begins with the characters 0x and is followed by a sequence of digits, including the hexadecimal digits a or A through f or F. If the integer constant does not begin with the characters 0x, then it is interpreted as decimal and must consist of a sequence of decimal digits.

2.4.2. String Literals

String literals consist of a sequence of characters enclosed in double quotes. Double quotes may be included as part of a string literal by using the escape sequence \”.

3. Meaning of Identifiers

Variables, packet tests, conformance rules, functions, and variables are all named using identifiers. Variables have associated properties indicating their scope and the type of information referred to by the variable.

3.1. Scope

All identifiers have global scope with the exception of variables, which may have either global or local scope. Global variables are declared at the beginning of the program, outside of a function definition. Local variables are declared inside a function definition, and are visible only within that function. The value of a local variable does not persist between calls to a function.

3.2. Type

Variables may be of type integer or string. Integer variables store 32 bits of information, and are always interpreted as an unsigned value. Strings store a sequence of characters and may be any length.

4. Expressions

The subsections listed below are in order of the precedence of the operators. Operators defined within the same subsection have equal precedence.

4.1. Primary Expressions

Primary expressions are identifiers, constants, parenthesized expressions, subscripted identifiers, or function calls, and are associated left to right.

4.1.1. identifier

Identifiers (described in section 2.2) are primary expressions provided that they have been properly declared as described below.

4.1.2. constant

Constants, as defined in section 2.4, are primary expressions and may be either numeric integers or string literals.

4.1.3. (expression)

Parenthesized expressions are primary expressions with higher precedence than the equivalent expression without parenthesis. The type and value of the parenthesized expression is equivalent to the unadorned expression.

4.1.4. primaryExpression `[ expression secondaryIndex`$_{opt}$`]`

Primary expressions followed by an expression and an optional secondary index in square brackets is a primary expression referred to as a subscripted expression. In PERIL, the primary expression refers to a collection of data elements and the indexes (i.e. the expression and optional secondary index) identify a particular data element or range of data elements within the collection.

4.1.5. identifier ( ) ;

An identifier followed by an open and close parenthesis and a semicolon is a primary expression denoting a function call.

4.2. Postfix Expression

A postfix expression is a primary expression followed by square brackets containing an expression or two expressions separated by a comma.

4.3. Multiplicative operators

The binary multiplicative operators *, /, and % group left to right. Both operands of a multiplicative operator must be integer.

*4.3.1. expression * expression*

The * operator denotes multiplication.

4.3.2. expression / expression

The / operator denotes integer division. Dividing by zero is a run-time error and will cause a PERIL program to terminate.

4.3.3. expression % expression

The % operator returns the remainder of the first operand divided by the second.

4.4. Additive operators

The binary operators + and – group left to right. Both operators require integer operands.

4.4.1. *expression + expression*

The + operator returns the sum of the operands.

4.4.2. *expression – expression*

The – operator returns the difference of the second operand subtracted from the first.

4.5. Shift operators

The binary shift operators << and >> group left to right. Both operands must be integer.

4.5.1. expression << expression
4.5.2. expression >> expression

The left shift operation is denoted by <<, and indicates shifting the binary representation of the first operand left by the number of bits specified by the second operand modulo 32. All vacated bits are zero-filled. The right shift operator, >>, is defined similarly, differing only in the direction that bits are shifted.

4.6. Relational operators

The binary relational operators perform logical comparisons between the operands. The operands must be the same type, but that type may be either integer or string.

4.6.1. ***expression < expression***
4.6.2. ***expression > expression***
4.6.3. ***expression <= expression***
4.6.4. ***expression >= expression***

The operators < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) return 1 if the specified relation is true or 0 if it is false.

4.7. Equality operators

Similar to the relational operators, the binary equality operators perform logical comparisons between the operands. The operands must be the same type, but that type may be either integer or string.

4.7.1. expression == expression
4.7.2. expression != expression

The operators == (equal to) and != (not equal to) return 1 when the specified condition is true or 0 if it is false.

4.8. expression & expression

The binary & operator groups left-to-right and returns the bitwise logical `and` of the operands. Both operands must be of type integer.

4.9. expression ^ expression

The binary ^ operator groups left-to-right and returns the exclusive bitwise `or` of the operands. Both operands must be of type integer.

4.10. expression | expression

The binary | operator groups left-to-right and returns the bitwise inclusive `or` of the operands. Both operands must be of type integer.

4.11. expression && expression

The binary && operator returns 1 if both operands are non-zero, 0 otherwise. Both operands must be of type integer.

4.12. *expression || expression*

The binary || operator returns 1 if either operand is non-zero, 0 otherwise. Both operands must be of type integer.

4.13. expression = expression

The binary = operator stores the second operand into the memory location referred to by the first operand. The type of both operands must be identical.

5. Declarations

Declarations are used to specify names and data types of variables. Declarations have the form:

```
declaration:
        typeSpecifier declaratorList ;
```

5.1. Type specifiers

The type specifiers are

```
typeSpecifier:
        int
        string
```

and are described further in section 2.4.1 and 2.4.2, respectively.

5.2. Declarators

As described previously, declarations consist of a type specifier followed by a declarator list, which is a sequence of comma-separated declarators.

```
declaratorList:
     declarator
     declarator , declarator-list
```

The type given in a declaration applies to all declarators in the declarator list. Declarators have the form:

```
declarator:
     identifier initializer_opt ;
```

5.3. Initializers

Declarators may optionally initialize a variable by including an initializer. If present, the initializer follows the identifier and consists of an assignment operator followed by an expression.

```
initializer:
        = expression
```

6. Statements

A statement is defined as an expression followed by a semicolon. Related to a statement is a compound statement, which is an open brace, zero or more local variable declarations, zero or more statements, and a closing brace. Compound statements are used only in the body of a function definition.

6.1. Packet Tests

At least one packet test must follow the optional global variable declarators. Packet tests consist of an identifier, a colon, and an expression. The expression will typically refer to packet data through a packet reference and compare it with a constant or computed value. The result of a packet test will be interpreted by one or more conformance rules as a boolean value where zero is considered to be false and any non-zero value is true.

6.1.1. Packet References

Packet data is represented as a sequence of bytes and is referred to using an array-like syntax via the **byte**, **bytes**, or **string** keywords. **byte** is accessed using a single subscript, so that **byte**[x] returns the byte at offset x in the current packet. **bytes** can return up to four contiguous bytes, and is referenced as **bytes**[x, y], where x is the starting offset in the current packet and 1 <= y <= 4 is the number of bytes to return. **string** uses similar notation as **bytes**, but returns the data as a string and therefore has no length restriction on the size of data that can be returned.

6.2. Conformance Rules

Packet tests are followed by at least one conformance rule. Conformance rules are syntactically similar to packet tests, differing only by the replacement of the colon with the keyword **when**. Conformance rules are used evaluate the results of packet tests through boolean algebra.

7. Functions

One of more function definitions follow the conformance rules. Functions may have the same name as a conformance rule, a unique identifier, or may have the special identifier `init`. If a conformance rule evaluates as true, then the function with the matching name is called. Function names without matching conformance rules are utility functions, and may be called by other functions. If defined, the `init` function will be called before any other functions are executed.

7.1. Definitions

Following the conformance rules are one or more function definitions. A definition consists of an identifier suffixed by an open and close parenthesis followed by a compound statement.

Appendix A: Syntax Summary

1. Expressions

```
expression:
     primaryExpression
     expression binop expression

primaryExpression:
     identifier
     constant
     ( expression )
     primaryExpression [ expression secondaryIndexopt]
     functionCall

secondaryIndex:
     , expression

functionCall:
     identifier ( ) ;
```

The primaryExpresion operators

() []

have highest priority and associate left-to-right. The binary operators all associate left-to-right, and are listed in decreasing precedence as indicated:

```
binop:
     *    /    %
     +    -
     >>   <<
     <    >    <=   >=
     ==   !=
     &
     ^
     |
     &&
     ||
     =
```

2. Declarations

```
declaration:
     typeSpecifier declaratorList ;
```

typeSpecifier:
  int
  string

declaratorList:
  declarator
  declarator , declarator-list

declarator:
  identifier initializer$_{opt}$ *;*

initializer:
  = *expression*

3. Statements

statement:
  expression ;
  packetTest ;
  conformanceRule ;

statementList:
  statement
  statement statementList

packetTest:
  identifier : *expression ;*

conformanceRule:
  identifier **when** *expression;*

4. Functions

functionDefinition:
  identifier **( )** *functionBody*

functionBody:
  { *declaratorList statementList* **}**

Appendix B: Sample Program

```
/***************************
* Global variable declarators
***************************/

int ipLength = (byte[0] & 0xf) * 4;
int space;

/**************
* Packet tests
**************/

//Next protocol TCP?
tcp : byte[9] == 6;

// Client to server POP3?
dstport110 : bytes[ipLength, 2] == 110;

// Does the packet contain the user command?
user : string[0, 3] == "user";
USER : string[0, 3] == "USER";

// Does the packet contain the password command?
pass : string[0, 3] == "pass";
PASS : string[0, 3] == "PASS";

/******************
* Conformance rules
******************/

/*
Is this a TCP packet going to the POP3 well-known port that
contains a user command, either in all uppercase or
lowercase?
*/
pop3_user when tcp && dstport110 && (user || USER);

/*
Similar to previous conformance rule, except looks for the
pass command.
/*
pop3_pass when tcp && dstport110 && (pass || PASS);

// POP3 client-to-server catch-all conformance rule
pop3 when tcp && dstport;
```

```
/**********************
* Function definitions
**********************/

init()
{
     space = SPACE;
     #command = string[0, space-1];
}

pop3_user()
{
     #username = string[space+1, EOP];
}

pop3_pass()
{
     #password = string[space+1, EOP];
}
```