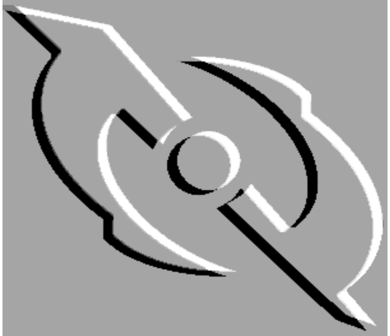

CASL
(Custom Audit Scripting Language)
for Linux Red Hat 5.x

Programming Guide

Version 2.0

COPYRIGHT

TRADEMARK ATTRIBUTIONS

ActiveHelp, Bomb Shelter, Building a World of Trust, CipherLink, Clean-Up, Cloaking, Compass 7, CNX, CyberCop, CyberMedia, Data Security Letter, Discover, Distributed Sniffer System, Dr Solomon's, Enterprise Secure Cast, First Aid, ForceField, Gauntlet, GMT, Hunter, ISDN Tel/Scope, LM 1, LANGuru, Leading Help Desk Technology, Magic Solutions, MagicSpy, MagicTree, Magic University, MagicWin, MagicWord, McAfee Associates, McAfee, MoneyMagic, More Power To You, Multimedia Cloaking, NetCrypto, NetRoom, NetScan, Net Shield, NetShield, NetStalker, Net Tools, NetOctopus, NetStalker, Network Associates, Network General, Network Uptime!, NetXRay, Nuts & Bolts, PC Medic, PCNotary, PGP, PGP (Pretty Good Privacy), PocketScope, PowerTelnet, Pretty Good Privacy, PrimeSupport, RecoverKey, RecoverKey-International, ReportMagic, RingFence, Router PM, Safe & Sound, SalesMagic, SecureCast, Service Level Manager, ServiceMagic, Site Meter, Sniffer, SniffMaster, SniffNet, Stalker, Statistical Information Retrieval (SIR), SupportMagic, T-POD, TeleSniffer, TIS, TMach, TMeg, Trusted Mach, Trusted Mail, Total Network Visibility, Total Virus Defense, Uninstaller, Virex, Virex-PC, Virus Forum, ViruScan, VirusScan, VShield, WebScan, WebShield, WebSniffer, WebStalker WebWall, and *ZAC 2000* are registered trademarks of Network Associates and/or its affiliates in the US and/or other countries. All other registered and unregistered trademarks in this document are the sole property of their respective owners.

LICENSE AGREEMENT

BEFORE YOU INSTALL, USE, OR MODIFY THIS SOFTWARE PRODUCT, CAREFULLY READ THE TERMS AND CONDITIONS IN THE FILE "LICENSE.TXT" ACCOMPANYING THIS DOCUMENT. IF THE FILE "LICENSE.TXT" IS MISSING, IT MAY BE OBTAINED FROM NETWORK ASSOCIATES. NETWORK ASSOCIATES IS PERMITTING THE USE, DISTRIBUTION, AND LIMITED MODIFICATION OF THIS SOFTWARE PRODUCT ON A NON-COMMERCIAL BASIS SUBJECT TO ALL OF THE CONDITIONS IN THE FILE "LICENSE.TXT." BY INSTALLING, USING, OR MODIFYING THE SOFTWARE PRODUCT, YOU AND ANY SUBSEQUENT USER ARE AGREEING TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS IN THE FILE "LICENSE.TXT." IF YOU DO NOT AGREE TO ALL OF THOSE TERMS AND CONDITIONS, DO NOT INSTALL, USE, OR MODIFY THIS SOFTWARE PRODUCT.

CONTACT INFORMATION

Network Associates, Inc.
Web site: **http://www.nai.com**
CASL discussion list: **casl@lists.nai.com**

Table of Contents

Preface

This preface includes important information about CASL (custom audit scripting language). We recommend that you read this preface thoroughly before using CASL.

CASL Overview

CASL is a high-level programming language designed to write programs (often called scripts) that simulate low-level attacks or information gathering checks on networks. To write programs that simulate an attack or information gathering check, you need to write code that constructs packets and then sends those packets to a host on a network just as an actual attack or information gathering check would. You can execute the programs you create in CASL to determine if a network is vulnerable to the attack or the information gathering check simulated by the programs. To write programs in CASL you must have the CASL interpreter installed on your system.

System Requirements

There are minimum system requirements that must be met to install and use the CASL interpreter. The minimum system requirements are as follows:

- Red Hat Linux operating system 5.x
- 133 MHz Pentium processor
- 32 MB of RAM
- 3 MB of free disk space

If your system does not meet the above-listed requirements, you need to upgrade the system accordingly before installing CASL.

How to Use this Programming Guide

This Programming Guide includes two chapters.

Chapter 1, “Installing CASL,” includes step-by-step instructions for installing and uninstalling the CASL interpreter.

Chapter 2, “CASL Reference,” provides a detailed explanation of the custom audit scripting language (CASL) which you can use to write your own scripts using a text editor and run them using the CASL interpreter. Chapter 2 includes a description of CASL program structure and syntax, as well as a programming reference section.

Installing CASL

1

Introduction

This chapter includes step-by-step instructions for installing (and uninstalling) the CASL (custom audit scripting language) interpreter. The CASL interpreter lets you write your own programs in a text editor that simulate attacks or information gathering checks. CASL directories and files are stored as a single compressed archive file. This chapter assumes that you already have the compressed archive file on your system.

There are minimum requirements that your system must meet before you can install and use CASL. They are as follows:

- Red Hat Linux operating system 5.x
- 133 MHz Pentium processor
- 32 MB of RAM
- 3 MB of free disk space

If your system does not meet the above-listed requirements, you will need to upgrade the system accordingly before installing the CASL interpreter.

Installing the CASL Interpreter

This section gives instructions for installing the CASL interpreter on your system. Your CASL distribution is stored as a single compressed archive file with the extension **.tgz** which you decompress and unpack using the Linux **tar** command. By default, the **tar** process will create the directory **casl-2.0** in your current working directory. This directory contains subdirectories and program files.

> **NOTE:** You must have permission to write to the directory that you wish to install CASL to.

To install the CASL interpreter, do the following:

1. Download the CASL distribution file (**.tgz** extension) to your computer. The distribution file has the filename **casl20**.
2. To decompress and unpack the distribution file, type the following command:

   ```
   # tar zxvf [filename.tgz]
   ```

   where **[filename.tgz]** is replaced by the filename (including the **.tgz** extension) of the CASL distribution file.

The contents of the file will be extracted and a directory named **casl-2.0** with subdirectories and program files will be created in your current working directory.

3. Set the environmental variable CASL_DIR to the directory where CASL is installed.

If you wish to uninstall CASL, delete the directory **casl-2.0**.

Summary

This chapter included step-by-step instructions for installing the CASL interpreter. It also included instructions for uninstalling the CASL interpreter in case you need to remove it from your system. At this point, you are ready to use CASL. You can go to Chapter 2, the reference manual for CASL. Chapter 2 includes a description of CASL program structure and syntax, as well as a programming guide.

CASL Reference

2

Introduction

This chapter is a guide to CASL (custom audit scripting language). CASL is a high-level programming language. CASL lets you write programs in a text editor that simulate attacks or information gathering checks, making CASL ideal for evaluating network security. To write programs in CASL you must have the CASL interpreter installed on your system. (See Chapter 1 for instructions on installing the CASL interpreter.)

In this chapter, you will find information on the following topics:

- an explanation of CASL
- an introduction to the main elements of CASL programs, including an example CASL program
- a reference section containing detailed descriptions of the elements you can use in CASL programs
- a summary of the CASL built-in functions you can use in CASL programs

CASL is for expert use only. CASL requires high-level programming experience and an understanding of TCP/IP protocol.

About CASL

CASL is a high-level programming language designed to write programs (often called scripts) that simulate low-level attacks or information gathering checks on networks. To write programs that simulate an attack or information gathering check, you need to write code that constructs packets and then sends those packets to a host on a network just as an actual attack or information gathering check would. You can execute the programs you create in CASL to determine if a network is vulnerable to the attack or the information gathering check simulated by the programs.

Writing programs to simulate low-level attacks on networks is difficult, if not impossible, in most high-level programming languages. As an example, consider the Tear Drop attack. Tear Drop sends two IP packet fragments to a host. The two IP packet fragments overlap each other, which cause crashes on Windows NT and Linux operating systems. Sending overlapping IP packet fragments is difficult in C and impossible in COBOL. In CASL sending overlapping IP packet fragments is easy, making CASL ideal for simulating attacks like Tear Drop.

Writing programs that are not operating system dependent is impossible in most high-level programming languages. For instance, consider the information gathering check TCP Stealth Port Scan. TCP Stealth Port Scan detects if a connection can be made to a port on a host. (TCP Stealth Port Scan does not open the connection.) In C, you need to write separate programs for different operating systems. For example, if you want to execute TCP Stealth Port Scan on the Windows NT and Linux operating systems, you write two programs—one for Windows NT and the other for Linux. In CASL, you can write one program for TCP Stealth Port Scan and execute it on many operating systems.

The next section, “Programming With CASL,” is designed to familiarize you with the main elements of CASL programs. It also includes an example CASL program for TCP Stealth Port Scan.

Programming With CASL

This section is divided into two parts. The first part, “Structuring CASL Programs,” introduces you to the main elements of CASL programs. The second part, “Understanding an Example CASL Program,” includes an example CASL program—TCP Stealth Port Scan. This part guides you through the elements you use to create the TCP Stealth Port Scan program.

Structuring CASL Programs

You write CASL programs in a text editor. The main elements you use to write CASL programs (or, scripts) include:

- statements
- variables
- comments
- packets

A CASL program consists of statements. A **statement** is defined as an action, for example calculating the value of `2+2` or reading a UDP packet. A statement operates on variables. A **variable** can be:

- an ASCII character, which is represented in single quotes (e.g. ’`c`’)
- a number, which is represented as either: 1) a positive or negative integer without quotes; or 2) an integer in hexidecimal format with `0X` preceding the integer
- a string, which is represented as either: 1) a sequence of characters in double quotes (e.g. `"hello,world!"`); or 2) control sequences represented in backslash quoted codes (e.g. new line is ’`\n`’)
- a buffer, which holds a collection of data, generally input packets
- a list, which holds a collection of data, generally output packets

A CASL program supports comments that are ignored by the interpreter. A **comment** can be either a single line or multiple lines. A single line comment beings with `"//"`. A multiple line comment begins with `"/*"` and ends with `"*/"`.

In a CASL program, you create **packets**, which are units of protocol data, from scratch. Or, you create packets using predefined packet templates included in CASL. Defining a packet in CASL consists of selecting the desired protocol structure and then setting data elements in the packet.

The subsequent section includes an example CASL program, TCP Stealth Port Scan, which illustrates the main elements of a CASL program.

Understanding an Example CASL Program

This section guides you through an example CASL program for TCP Stealth Port Scan. TCP Stealth Port Scan is an information gathering check. TCP Stealth Port Scan requests a connection to a port on a host by sending a TCP SYN packet to the host. The TCP Stealth Port Scan program then waits for a response to the TCP SYN packet. The TCP response can be:

- an acknowledgment, indicating a service is listening and willing to accept a connection for the port,
- a reset, indicating a service is not offered for the port, or
- nothing, indicating something, for example a firewall, is filtering out the connection attempt

Note that the TCP Stealth Port Scan does not open a connection to a port, even when a service is available on the port.

This is the TCP Stealth Port Scan program created in CASL.

```
#include "tcpip.casl"
#include "packets.casl"
for(i = 1; i < 1023; i = i + 1) {
    OurSYN = copy SYN;
    OurSYN.tcp_source = 10;
    OurSYN.tcp_destination = i;
    OurIP = copy TCPIP;
    OurIP.ip_source = 127.0.0.1;
    OurIP.ip_destination = 127.0.0.2;
    OurPacket = [ OurIP, OurSYN ];
    ip_output(OurPacket);
    OurFilter = [ "src host ", 127.0.0.2, " and tcp src port ", i ];
    ReadPacket = ip_input(2000, OurFilter);
    if(!ReadPacket)
            continue;
    if(size(ReadPacket) < size(IP) + size(TCP))
            continue;
    ReadIP=extract ip from ReadPacket;
    ReadTCP=extract tcp from ReadPacket;
    if(ReadTCP.tcp_ack != 1
                || ReadTCP.tcp_syn != 1
                || ReadTCP.tcp_rst == 1)
            continue;
    print("Port", i, "Alive");
}
```

NOTE: The key words in the TCP Stealth Port Scan program above are described in detail in the section "CASL Reference" later in this chapter.

The sections below lead you through the steps you perform to create the TCP Stealth Port Scan program in CASL.

Step One: Defining TCP/IP Packets

To set up a TCP Stealth Port Scan program, you need to create TCP/IP packets. TCP/IP header defaults for TCP/IP packets are included in CASL. You enter the following statement to access TCP/IP header defaults:

```
#include "tcpip.casl"
#include "packets.casl"
```

Step Two: Creating a TCP SYN Packet

Next, you need to create a TCP SYN packet, which is the packet that requests a connection to a port on the destination host. You create a TCP SYN packet using a predefined TCP packet header template, changing predefined parameters in the template as appropriate. You enter the following statement to create a TCP SYN packet using the template:

```
OurSYN = copy SYN;
OurSYN.tcp_source = 10;
OurSYN.tcp_destination = 2049;
```

The above statement assigns a source port of `10` (an arbitrary number) and a destination port of `2049` (the TCP NFS port) to the TCP packet header for example purposes only. You can change the source port and the destination port numbers as you wish.

Step Three: Specifying a Destination Host for the TCP SYN Packet

Now, you add an IP header to the TCP SYN packet header. In the IP header, you specify the destination host for the TCP SYN packet. You enter the following statement to add an IP header to the TCP SYN packet header:

```
IP= copy TCPIP;
OurIP.ip_source = 127.0.0.1;
OurIP.ip_destination = 127.0.0.2;
```

The above statement defines the source host as `127.0.0.1` and the destination host as `127.0.0.1`. The source host and destination host IP addresses are provided for example only. If you write the TCP Stealth Port Scan in CASL, make sure that you enter IP addresses appropriate for desired source and destination hosts.

Step Four: Combining TCP SYN and IP Headers

Next, you combine the TCP SYN and IP headers. There are two ways to combine TCP SYN and IP headers. You can combine them using either: 1) a list variable or; 2) list operators.

You enter the following statement to combine TCP SYN and IP headers using a list variable:

```
PacketList = [ OurIP, OurSYN ];
```

The above statement creates a list called PacketList, with one operator for each component in the list. The opening bracket starts the list and the closing bracket ends the list. Individual values in the list are separated by a comma.

You enter the following statement to combine TCP SYN and IP headers using list operators:

```
PacketList = PacketList push OurSYN;
PacketList = PacketList push OurIP;
```

The above statement creates a list called PacketList, with a separate operator for each component in the list. TCP and IP headers are added to the list separately. (The last element added (or, pushed) onto the list is the first element written to the list.)

Step Five: Outputting the TCP SYN Packet

Next, you instruct the program to output the TCP SYN packet onto a network by entering the following statement:

```
ip_output(PacketList);
```

Step Six: Defining Port Connections

Most standard network services listen to reserved ports. Therefore, you want to instruct TCP Stealth Port Scan to get information for reserved port nos. 1 through 1023. You get information about reserved ports by looping through the ports. You enter the following statement to loop through reserved ports:

```
for (i = 1; i < 1023; i = i + 1) {
    //
}
```

The `for` statement above is defined using three parameters, with `i` as the counter:

- The first parameter, `i=1`, tells the interpreter where to start counting.
- The second parameter, `i < 1023`, tells the interpreter how long to count.
- The third parameter, `i = i + 1`, tells the interpreter how far to move forward for each step.

Step Seven: Sending Connection Requests to Ports

You enter the following statement to send connection requests to reserved ports.

```
For (i = 1; i < 1023; i = i + 1) {
    OurSYN = copy SYN;
    OurSYN.tcp_source = 10;
    OurSYN.tcp_destination = i;
    OurIP = copy TCPIP;
    OurIP.tcp_source = 127.0.0.1;
    OurIP.tcp_destination = 127.0.0.2;
    OurPacket = [ OurIP, OurSYN ];
    ip_output(OurPacket);
}
```

Step Eight: Reading TCP Responses

You use `ip_input()` routines to determine if a port on a destination host answered the program's connection requests. `ip_input()` routines specify the time (in milliseconds) for attempting a connection. `ip_input()` routines also specify the packets types to be read using a `tcp_dump` filter.

You enter the following statement to read a response to a packet:

```
OurFilter = [ "src host ", 127.0.0.2, " and tcp src port ", i ];
```

where `i` is equal to 103

```
ReadPacket = ip_input(2000, OurFilter);
```

If `ip_input()` does not read a packet successfully, it returns a value of zero. Each time `ip_input()` is used, you must check if it reads a packet successfully by comparing the returned value to `0`. You enter the following statement to compare values:

```
if(!ReadPacket)
continue;
```

In the above statement, `continue` tells the interpreter to move forward in the loop. When the program reads a packet, it returns a complete IP packet.

Step Nine: Determining TCP Response Types

Next, you need to determine if the complete IP packet is a TCP SYN+ACK or a TCP RST packet. If the IP packet is a TCP SYN+ACK packet, a service was listening and willing to accept a connection for the port. If the packet is a TCP RST packet, a service is not offered for the port. You can determine if the IP packet is a TCP SYN+ACK or a TCP RST packet by looking at its packet size and packet header, as described below.

First, you check the size of the IP packet. The IP packet must be large enough to contain a TCP and IP header. You enter the following statement to check the IP packet size:

```
if(size(ReadPacket) < size(IP) + size(TCP))
continue;
```

The above statement tells the interpreter to move forward in the loop if the IP packet is smaller in size than the sum of the sizes of the TCP and IP headers. If the IP packet is large enough, the packet header can be extracted from the IP packet. You enter the following statement to extract the packet header:

```
ReadIP = extract ip from ReadPacket;
ReadTCP = extract tcp from ReadPacket;
```

Each header in the above statement is extracted using the `extract` operator. Once the packet headers are extracted, you look at the individual fields of the TCP header to verify that they are set properly. The SYN and ACK fields should be set; the RST field should not be set. Note that if the aforementioned fields are not set properly, the connections to the port will be opened.

Enter the following statement to view TCP header fields:

```
if(ReadTCP.tcp_ack != 1 || ReadTCP.tcp_syn != 1 || ReadTCP.tcp_rst == 1)
continue;
```

where `||` is a logical or and != is not equal. The statement reads: If the ACK flag is not set, or the SYN flag is not set, or the RST flag is set restart the loop for the next port. If the programs proceeds in the loop after this statement, the packet is a TCP SYN + ACK packet. This packet type indicates that a service was listening and willing to accept a connection for the port.

Step Ten: Verifying an Open Port Connection

The `print` function notifies you if there is a port open for connection. You enter the following statement to see if a port is open for connection:

```
print("Port", i, "Alive");
```

If `i` is `1022`, Port 1022 Alive is printed.

Step Eleven: Evaluating the Completed Program

The program for TCP Stealth Port Scan is now complete.

```
#include "tcpip.casl"
#include "packets.casl"
for(i = 1; i < 1023; i = i + 1) {
    OurSYN = copy SYN;
    OurSYN.tcp_source = 10;
    OurSYN.tcp_destination = i;
    OurIP = copy TCPIP;
    OurIP.ip_source = 127.0.0.1;
    OurIP.ip_destination = 127.0.0.2;
    OurPacket = [ OurIP, OurSYN ];
    ip_output(OurPacket);
    OurFilter = [ "src host ", 127.0.0.2, " and tcp src port ", i ];
    ReadPacket = ip_input(2000, OurFilter);
    if(!ReadPacket)
            continue;
    if(size(ReadPacket) < size(IP) + size(TCP))
            continue;
    ReadIP=extract ip from ReadPacket
    ReadTCP=extract tcp from ReadPacket
    if(ReadTCP.tcp_ack != 1
                    || ReadTCP.tcp_syn != 1
                    || ReadTCP.tcp_rst == 1)
            continue;
    print("Port", i, "Alive");
}
```

You can write the above program in a text editor making changes where appropriate (for example changing IP addresses) and then execute the program.

NOTE: Before testing CASL programs on critical networks, we recommend that you test them on non-critical networks. CASL programs are most often attacks, which means they can disrupt and disable networks.

The next section, "CASL Reference," includes detailed descriptions of all the elements you can use in CASL programs.

CASL Reference

This section includes a description of each element you can use in a CASL program, or script. It is divided into four main sections:

- program structure
- lists
- packet headers
- subroutines

You can skip straight to the section that describes the element you are interested in.

Program Structure

This section includes definitions of elements related to CASL program structure. This section is divided into four main parts:

- statements
- variables
- syntax
- control statements

Statements

CASL programs consist of statements. Statements consist of control constructs and expressions. Control constructs are statements which define the flow of a program, for example loops (`while` and `for`) and conditionals (`if`). Expressions are sentences which evaluate to a value. You can execute statements in global scope, which eliminates the need for creating a program with routines. You do not need to use an entry point `main()` function in CASL.

Variables

Statements operate on variables. Variables are dynamically typed, therefore they do not have a declared type and do not need to be declared prior to use. You can assign variables (described below) to expressions. There are five variable types—character, integer, string, buffer, and list.

Characters

Characters are ASCII characters. Characters are represented in single quotes (e.g. `’c’`).

Integers (Numbers)

Integers (i.e. numbers) are represented as either: 1) positive or negative intergers without quotes; or 2) integers in hexidecimal format when `0X` precedes the integer. Note that floating point and decimal point numbers are not allowed in CASL.

Strings

Strings are any number of characters enclosed in double quotes, for instance `"hello world!"` CASL treats strings as built-in types, not as arrays. (Perl and C treat strings as arrays.)

You can define string literals, which may include adjacent string literals. **String literals** are constant strings in a CASL source file, for example "hello world!" **Adjacent string literals** are concentrated into a single string. For example, "foo" "bar" is equivalent to the string "foobar". String literals can contain escape codes representing non-ASCII characters. Escape codes include "\n" (newline), "\r" (carriage return), "\t" (tab), and "\xNN" (the character represented by the ASCII hex code NN).

Buffers

Buffers are complex types, which can contain many pieces of information. Buffers express pieces of information as bytes. Buffers generally hold packet structures and input packets.

Lists

Like buffers, lists are complex types which can contain many pieces of information. Lists are discrete series of variables. Lists generally hold output packets.

Syntax

The subsequent sections describe the syntax used to express elements.

Statements

CASL code consists of statements. Statements are terminated with a semicolon. They are case sensitive and whitespace insensitive. Thus, you can indent and space CASL programs as you wish.

You can use single statements or a collection of statements in CASL programs. Single statements stand on their own. A collection of statements can be grouped together. (When enclosed in curly braces, a collection of statements is treated as a single statement.)

Comments are remarks in CASL source code that are ignored by the interpreter. A comment can be either a single line or multiple lines. A single line comment beings with "//". A multiple line comment begins with "/*" and ends with "*/".

Variables

Variables are the basic elements of CASL programs. You can use characters, integers, strings, buffers, and/or lists as variables. Variables are assigned names. When you assign a name to a variable, the name must: 1) start with a letter; and 2) consist of zero or more trailing letters, numbers, or the underscore "_" character. Examples of valid variable names include the following: foo, bar_baz, i, and z1. Examples of invalid variables include 1a and a@b.

Variable Assignments

Variable names are not valid until they are assigned to by an assignment operator, =. An assignment takes the value of the expression to the right of the = and assigns it to the variable on the left. The variable assigned to does not need to exist beforehand. For instance, i = c assigns the value of the variable c to i. In this example, c must exist beforehand; i does not need to exist beforehand.

Increment and Decrement Operators

Increment operators add a value of one to a variable. Decrement operators subtract a value of one to a variable. Both increment and decrement operators can be used with either preincrement or postincrement options. **Preincrement** adds the value one to a variable and then returns it for further expression evaluation. **Postincrement** subtracts the value one to a variable, however, it returns the original variable for further expression evaluation.

Expressions for increment operators with preincrement and postincrement options are ++x and x++, respectively. Expressions for decrement operators with the preincrement and postincrement options are --x and x--, respectively.

Math

CASL supports both standard mathematical operations and binary operations. **Standard mathematical operations** include addition, subtraction, multiplication, and division, which are represented by +, -,*, /, and % (modulo division), respectively. For example, if you want to increment a variable i by one, you use the statement i = i + 1. **Binary operations** allow integers to be masked against one another to extract bit patterns. Supported binary operations include: AND (&), OR (|),XOR (^), NOT (~), and left/right shifts (<< and >>).

Comparison Operators

Comparison operators test the value of an expression. Comparison operators include:

- x > y, which reads x is greater than y
- x < y, which reads x is less than y
- x >= y, which reads x is greater than or equal to y
- x <= y, which reads x is less than or equal to y
- x == y, which reads x is exactly equal to y
- x != y, which reads x is not equal to y

Expressions

Expressions enclosed in parenthesis () are treated and evaluated as single expressions. You can use parenthesis to clarify complicated expressions, which may be confusing to the CASL interpreter. You can also use parenthesis to compare the value of an assignment, for example:

```
if((i = 1) == 1)
print(i);
```

You can invert expressions for comparison with the ! operator. Expressions preceded by a ! evaluate false if the expression value is nonzero. For instance, if i is NOT 1 you enter the following:

```
if(! (i == 1))
print(i);
```

Negation with ! is most useful when comparing something to zero. !z evaluates true if z is zero. You can combine these rules to see if a packet is read from ip_input() by writing:

```
if(!(packet = ip_input(2000, filter))
    print("didn't get a packet");
```

You do not need to compare an expression's value to > 0 to see if the expression is nonzero, for example if(i > 0). If the expression evaluates nonzero, it evaluates true. If the expressions is zero, it evaluates false. Consider the following statement:

```
if(i)
    print(i);
else
    print("i is zero");
```

The above statement prints the value of i if i is not zero.

Control Statements

Control statements affect the flow of a program. Control statements are:

- loops, which cause a piece of code to be executed zero or more times, or
- conditionals, which cause a piece of code to be executed only if the condition is satisfied

Control statements operate on other statements and are terminated with a semicolon.

Loops

There are two loops types in CASL–while and for. while and for are described in the subsequent sections.

While

while statements represent loops that are not implicitly terminated. while loops execute their bodies until their conditional arguments are satisfied. while loops are written as follows:

```
while (conditional) statements
```

In the above statement, conditional is an expression and statements is either a statement or a group of statements enclosed in curly braces. The following is an example statement for a while loop:

```
while(i > 0)
    i = i - 1;
```

For

for statements represent loops that generally have implicit termination. for statements consist of three parts: an initializer, a conditional, and an iterator.

- The **initializer** is intended to set up a counter or some other place holder variable for the loop.
- The **conditional** works the same way a while conditional works; it is intended to terminate the loop when the condition evaluates false.
- The **iterator** is intended to move the loop forward, typically advancing or decrementing a counter.

The following is an example statement for a for loop:

```
for(i = 0; i < 10; i = i + 1)
    print(i);
```

The above statement executes print(i) ten times, starting with i equal to zero (outputting 0) and executing the last statement with i equal to 9. The statement terminates when i evaluates to 10. for(;;) is a legal statement representing an infinite loop. Note that each part of a for statement is separated by a semicolon.

Loop Control

Control can be affected by either the loop terminator or the loop continue statements in the body of a loop. Loops can be immediately terminated by executing the break statement. Loops can be continued to the next iteration with the continue statement as follows:

```
for(i = 0; 1; i = i + 1) {
    if(i != 4)
              continue;
    if(i == 4)
              break;
}
```

The above statement sets up an infinite loop. When the counter is a value besides 4, the loop moves forward. However if the counter reaches a value 4, the loop terminates. (Note continue in the above statement is redundant: It is meant for illustration purposes only.)

Loop control statements are only valid within loops. If you are not in a loop, you cannot execute a `break` or `continue`. `if` conditionals are not loops and remember the control statement affects the closest loop.

Consider the following statement:

```
for(;;)
    while(1)
              if(c == 1)
                        break;
```

In the above statement, `continue` affects `while`, not `for`. `continue` is valid in this statement because it is executed while at least one loop is in effect.

Now, consider the statement:

```
if(1)
    break;
```

The above statement is not valid because a loop is not present.

Conditionals

In CASL, conditional statements are `if`. When the conditional argument evaluates true, `if` executes its body of statements. Consider the following statement:

```
if(i == 1) {
    print(i);
    print("done");
}
```

When `i` is equal to 1, the above statement executes code in the body of the conditional.

Code can also be executed when a loop evaluates false using an `else` extension. The body of `else` is executed when `if` is false. For instance:

```
if(0)
    print("foo");
else
    print("bar");
```

The above statement prints the string `"bar"`. (The `0` conditional always evaluates false.)

`if/else` statements can be chained indefinitely using `else if`. For instance:

```
if(i == 1)
    print("foo");
else if(i == 2)
    print("bar");
else if(i < 4)
```

```
    print("baz");
else
    print("quux");
```

The above statement prints "foo" if i is 1, "bar" if i is 2, "baz" if i is 3, and "quux" if i is any other value.

Subroutine Calls

Subroutine calls divert control to code in the named subroutine. Subroutine calls pass arguments to subroutines, affecting execution of subroutines. Subroutines return values, which you can obtain by assigning subroutine call expressions to variables.

The syntax for a subroutine call is function(argument0, argument1, argumentN), where function is the name of the function (e.g., `ip_input`) and argumentX is the argument at position X. For example if `foo` is a function that takes as an argument a value and has as a return value of the value plus one, the following statement prints a value of two:

```
{
    i = 1;
    i = foo(i);
    print(i);
}
```

Lists

This section describes elements relating to lists. Lists represent collections of data, composed of individual variables. Lists can grow or shrink dynamically. You can use lists to represent complicated strings and packets. You can also use lists as data structures for CASL programs.

List Creation

There are two ways to create a list. You can create a list using a list comparison operator. Or, you can create a list by creating a new list and then using a list operator to assign an element to the list.

As mentioned above, you can create a list using the list composition operators [and]. The square brackets enclose a comma separated list of element. The following statement creates a new list:

```
[ foo, bar, baz, 1 ]
```

The above statement creates a list containing the variables `foo`, `bar`, `baz`, and `1`.

You can also create a new list using a list operator to assign an element to the list. More specifically, you assign the name of the list to an expression with a list operator operating on the name and then insert a new element. Consider the following statement:

```
list = list push foo;
```

The above statement creates a new list called `list` which contains only the element `foo`.

Recursion

Lists can contain any variable, including other lists. Lists can nest indefinitely. Routines that act on lists expand elements from lists in the order it encounters them. For example:

```
 [ "foo ", "bar ", [ "baz ", "quux " ], "zarkle" ];
```

The above statement defines a string list that evaluates to the following:

```
"foo bar baz quux zarkle"
```

When stepping through a list with list operators, an element of a list that is itself a list is returned as the entire list. It will not be returned as the first element of the list. The same string list above is processed with the following statement:

```
{
    list = [ "foo ", "bar ", [ "baz ", "quux " ], "zarkle" ];
    x = pop list;
    y = pop list;
    z = pop list;
    print(z);
}
```

The above statement prints the string "baz quux" because the value of z is equal to the third element of the list list.

List Operators

There are four list operators. They are as follows:

- head, which takes an element from the head of the list
- tail, which takes an element from the tail of the list
- prepend, which adds an element to the head of the list
- append, which adds an element to the tail of the list

Head and tail operate on a list, evaluating to the element removed from the list. The following is an example head statement:

```
{
    list = [ foo, bar, baz ];
    x = head list;
    print(x);
}
```

The above statement prints the value of foo, the first item (the head) of the list.

NOTE: You can use the head statement format to create a tail statement. To create a tail statement, you simply replace head with tail in the head statement format.

prepend and append operate on a list and an element to add to that list. If the list referred to doesn't already exist, it is created. An example of a prepend statement is:

```
{
    list = [ foo, bar ];
    list = list prepend baz;
    print(list); // list is now [foo, bar, baz]
}
```

The above statement prints the values of foo, bar, and baz.

NOTE: You can use the format of the prepend statement to create an append statement. To create an append statement, you simply replace prepend with append in the prepend statement format.

The commonly used computer stack terms, push and pop, are aliases for prepend and head, respectively.

List Control

You can use the foreach statement to step through each element in a list. A foreach statement has two parts:1) a binding name; and 2) a list to operate on. The binding name is set to refer to each element in the list. The following is an example of a foreach statement:

```
{
    list = [ foo, bar, baz ];
    foreach element [ list ] {
              print(element);
    }
}
```

The above statement prints the values of foo, bar, and baz, in order. The looping control statements continue and break function as they normally do.

NOTE: List expansion within foreach is recursive. A list containing other lists is expanded to all enlisted data elements.

Packet Headers

This section describes elements related to packet headers. You can create a packet that consists of a series of protocol headers, each with a fixed format. You can define fixed format protocol headers with the protocol structure construct. The format lays out bit-by-bit the order and the contents of a protocol structure.

Definition

Protocol structures are defined by `define` statements. A `define` statement creates a new structure with a specified name. The `define` statement consists of a curly-brace enclosed definition. The definition is composed of field specifiers which dictate the name, length, and order of the protocol fields. A basic protocol structure definition is as follows:

```
define foo {
    // contents here
}
```

The above statement creates a new structure named `foo`. However, `foo` is meaningless since it does not define fields. Consider the statement below, where `ip` defines fields:

```
define ip {
    ip_version: 4 bits;
    ip_headerlen: 4 bits;
    ip_tos: 8 bits;
    ip_length: 16 bits;
    ip_id: 16 bits;
    ip_df: 1 bit;
    ip_mf: 1 bit;
    ip_offset: 14 bits;
    ip_ttl: 8 bits;
    ip_protocol: 8 bits;
    ip_cksum: 16 bits;
    ip_source: 32 bits;
    ip_destination: 32 bits;
}
```

The above statement defines an IPv4 header. Each specifier enclosed in the curly braces denotes a field of the structure. Each field consists of a name, a colon, and a size. The name in a field can be any valid variable name. The size in a field can be specified in terms of any number of bits, bytes, words, and dwords. Words are16 bit quantities; dwords are 32 bit quantities. Protocol structure definitions can mix any combination of sizes specified in bytes, bits, word, or dwords.

Instantiation

A new instance of a protocol structure is created by assigning its name to a variable with the `new` operator. This creates a buffer large enough to hold the structure, with all fields in the structure set to `0`. When you assign a buffer to another variable, the buffer is copied. For example, consider the following statement:

```
{
    x = new ip;
    y = x;
    z = y;
}
```

In the above statement, `x`, `y`, and `z` are all independent copies of `ip` structures.

Field Reference

Individual fields of a structure are referenced with the field reference operator. For instance, if `x` is an `ip` structure `x.ip_ttl` refers to the `ip_ttl` field of `x`.

Any number can be assigned to a protocol structure field. Numbers are packed in Internet byte order into the field. Numbers will use as many bits as the field is large. It is an unchecked error to try to fit a value in a field that is too large for the value. For instance if `foo` is a field that is 1 bit wide, `x.foo = 4` results in undefined behavior.

Special Fields

Every buffer variable has four special fields which reference arbitrary locations within the buffer. The fields are bits, bytes, words, and dwords. The fields are specified with ranges corresponding to how many of units are referenced.

The syntax of a direct memory reference to a structure follows these examples:

- `z.bits[x .. y]`, which reads bits x through y of the buffer z
- `z.bytes[x .. ]`, which reads bytes x through the end of buffer z
- `z.word[x]`, which reads word x of buffer z

The above-listed statements evaluate to integer numbers. The statements can be assigned to, for example:

```
z.bit[10] = 1;
```

The above statement sets the eleventh bit (counting from `0`) of the buffer `z` to `1`.

Buffer Size

Buffers represent an arbitrary amount of data. You obtain buffer size using the `size` function. `size` evaluates to the size, in bytes, of its argument. Consider the following statement:

```
{
    x = new ip;
    print(size(x));
}
```

The above statement prints 20, which is the size (in bytes) of an IP header.

Variable Size Buffer

A variable size buffer is a structure that is defined without any fields. A variable size buffer can only be accessed using special fields. A variable size buffer automatically expands to fit new data.

Buffer Scale

You can define a default scale in a variable size buffer. A default scale is defined in the definition using `scale`. `scale` can be represented in bits, bytes, words, or dwords. When `scale` is defined, you can access the associated special field in the buffer by specifying the range. You do not need to include the field reference.

Structure Extraction

A buffer can contain several structures. You can obtain a structure from the buffer by extracting data with the `extract` operator. `Extract` is specified as follows:

```
foo = extract bar from baz;
```

The above statement extracts a `bar` structure from the buffer `baz`, leaving the remaining `bytes` in `baz`. To leave remaining `bytes`, write the following:

```
foo = extract z bytes from baz;
```

The above statement extracts `z bytes` from `baz`, leaving the remaining `bytes`.

Subroutines

This section describes elements related to subroutines.

Declaration

Subroutines are defined with the proc keyword. A subroutine takes a fixed number of arguments and returns a value. Subroutines can be defined anywhere. They do not require prototypes. To declare a new structure, you use the proc keyword as follows:

```
proc foo(arg1, arg2, argN) {
    // statements
}
```

In the above statement, foo names the new function, argX specifies the name of the argument at place X, and the body of the function appears in curly braces. Within the body of the function, the variables named argX are replaced by the value of the arguments passed at place X. For instance, to declare a function called foo that takes an argument named x and adds 1 to it you write the following:

```
proc foo(x) {
    x = x + 1;
    print(x);
}
```

Argument Passing

An argument specified in a function's declaration is called a formal argument. The name of the argument is available to all the statements executed in the body of this function. An argument passed to a function in a subroutine call is called a calling argument. Its value is made available through the name of the corresponding formal argument.

Argument passing in CASL is by value. (There is one exception, which is described below.) Thus, the formal argument is bound to the VALUE of the calling argument not the actual calling argument. Consider the following statement:

```
proc foo(x) {
    x = x + 1;
    print(x);

}
```

In the above statement foo, the addition of 1 to the argument x is never seen by the caller of foo—it affects only the variable x within the function foo.

The only exception to this argument is structure and list passing. References to lists and structures are passed. Changes to lists and structures affect variables on the caller side and variables in the body of the subroutine. Thus, it is easy to write routines that set fields within structure headers or to change the order of packet lists.

Variable Argument Lists

CASL supports creating procedures that take a variable number of argument using the list type. A variable argument function is defined as an argument that takes more calling arguments than formal arguments. The final formal argument becomes a list of all the extra calling arguments. Consider the following statement:

```
proc foo(x) {
    ...
}
foo(i, j, k);
```

The above statement defines a function called `foo`. `foo` can take a variable number of arguments. The function call to `foo()` specifies three arguments; the definition specifies one argument. Therefore, `x` becomes a list containing `i`, `j`, and `k`.

Return Values

Subroutines end when either: 1) a curly brace is reached; or 2) a control reaches a return statement. A return statement ends the execution of a subroutine and causes the subroutine call to evaluate to the value specified as return argument. For instance, to make `foo` return the value it calculated change use the following statement:

```
proc foo(x) {
    x = x + 1;
    return(x);
}
```

In the above statement, a call to `foo` will evaluate to the argument passed to `foo`, plus `1`.

Any variable can be returned through the return statement. Multiple values are returned from a function using list variable returns.

Scope

Scope is the space within which a variable is valid. When a program is executes within a subroutine, any variable it defines is accessible only within execution of the subroutine. The caller of the subroutine cannot access variables defined in the subroutine.

Code that is not executing within a subroutine is in global scope. Variables defined in global scope are accessible anywhere—even within subroutines. The following statement illustrates this concept:

```
i = 1;   // global
foo(i);

proc foo(x) {
    x = x + 1;          // local, "x" can only be accessed within "foo"
    y = i;              // "y" is local and can only be accessed within
                        // "foo," but "i" is global and can be accessed
                        // anywhere.
    return(x);
}
```

CASL Built-in Functions

The CASL interpreter includes built-in functions. Built-in functions are subroutines that cannot be easily programmed in CASL. Therefore, the CASL interpreter includes them as built-in functions. Built-in functions are divided into three categories: network I/O, file I/O, and misc (miscellaneous).

Network I/O Built-in Functions

Network I/O functions include subroutines that can be used to read packets from the network or to write packets to the network. Network I/O functions are described in subsequent sections.

The IP Output Function

IP output writes a complete IP packet (including the IP header) to the network. IP output in CASL is accomplished via the ip_output() routine. ip_output() takes as an argument a list of data elements that are expected to comprise an IP packet. A single buffer variable can also be passed to ip_output() for writing.

Sending a well formed IP packet involves some tricky issues, for instance checksum and length calculation. The IP and transport headers require knowledge of the length of the entire packet, the lengths of the individual headers, and the calculation of a checksum over some of the headers and the data.

You can write CASL code to compute checksums and lengths. However, this code can potentially be cumbersome and error-prone. Rather than requiring the implementation of CASL-scripted checksum and length calculation, the CASL interpreter provides a few shortcuts to solve these issues transparently. For the basic IP protocols (e.g. IP, TCP, UDP, and ICMP), the CASL interpreter automatically calculates checksum fields, packet lengths, and header lengths. The appropriate values are filled in before the packet is written to the wire. The computed values do not affect the passed in data; computed values only affect the packet written to the wire. In order to allow for arbitrary packets (possibly with intentionally bad header values) to be sent, CASL does not touch header fields it thinks have explicitly been filled in. For the basic IP protocols, this means that CASL does not fill in values for fields that already have nonzero values.

The IP Fixup Function

It is sometimes important to fill in the variable header fields of an IP datagram without outputting it to the network. This is a common requirement of IP fragmentation code. CASL supports this with the ip_fixup() procedure. Ip_fixup() takes the same arguments as ip_output(). However, instead of outputting the packet to the network, it returns a new packet. The new packet is a copy of the input with the appropriate header fields filled in.

The IP Input Function

IP input reads a complete packet (starting with the IP header) from the wire. Packet input in CASL is done using the ip_input() routine. Ip_input takes as arguments a timeout value, specified in milliseconds, and a tcpdump filter. The timeout specifies how long to wait for a packet before giving up and the filter defines which packets to read. If the millisecond timer runs out before a packet is read, ip_input returns the integer value 0.

If a packet is read successfully within the allotted time, it is returned minus the link-layer (Ethernet) header as a buffer. The size of the buffer can be queried with size() to determine the length of the inputted packet.

The IP Filters Function

CASL allows the explicit setting of global filters that affect all reads by using the ip_filter() routine. ip_filter takes as an argument a tcpdump filter, through which all packets read by CASL must successfully pass before being returned via ip_input.

On some computer architectures (notably 4.4BSD) ip_filter() also sets kernel packet filters. Enabling a kernel packet filter prevents the CASL interpreter from reading packets you specified not be read. This can be a major performance benefit, as it prevents the CASL interpreter from needing to explicitly filter out spurious packets.

The IP Range Function

Ranges of IP addresses can be quickly parsed into a list of IP address using the ip_range routing. The argument is a string describing a range of address and the return value is a list of integers.

File I/O Built-in Functions

The file I/O functions are subroutines which can be used to read and write to files. The file I/O functions are described in the table below.

Table 2-1. File I/O built-in functions.

Function	Description
open()	Takes a filename as an argument, and returns a descriptor number that can be used to manipulate that file. If the file does not exist, it will be created; if it does, it will be appended to. If the file cannot be opened, "0" is returned.
close()	Takes a descriptor number as an argument, and closes the associated file, flushing any pending output and preventing further manipulation of the file.
read()	Takes as arguments a descriptor number and a count of bytes to read. It reads at most the specified number of bytes from the file, and returns a buffer containing those bytes. The number of bytes actually read by the file can be queried with the "size()"command; if no data was read, "0" will be returned.
write()	Takes as arguments a descriptor and a data element (which can be a list or a buffer, or any of the basic types) to write to the file matching that descriptor. The number of bytes written to the file is returned.
fgets()	Takes as arguments a descriptor and a number representing the maximum number of characters to read from a file. It then reads at most that many characters, stopping when a line terminator (the new line character) is found. It returns the data read, or "0" if nothing was read.
rewind()	Repositions the offset into the descriptor given as an argument, so that it points to the beginning of the file. This allows the same data to be read from the same file descriptor twice.
fastforward()	Repositions the offset into the descriptor given as an argument, so that it points to the end of the file. This allows recovery from rewind(), for further writing.
remove()	Deletes the specified file from the system, returning "1" if successful.

seek()	Repositions the offset into the descriptor give as an argument, so that it points the offset referenced by the second argument. A third argument can be given to specify what the new offset is relative to. The possible values are as follows. SEEK_SET to set the offset from the beginning of the file. SEEK_CUR to set the offset relative to the current offset. SEEK_END to set the offset value relative to the end of the file. Note if the third argument is not given, the default is SEEK_SET.

MISC (Miscellaneous) Built-in Functions

The misc (miscellaneous) built-in functions are described in the table below.

Table 2-2. Misc built-in functions.

Function	Description
print()	Takes a list of data elements to write to standard output. It writes each of these elements, separated by a space, to standard output followed by a new line.
checksum()	Takes a list of data elements to perform an Internet checksum on. It returns an integer representing the checksum of these elements.
timer_start()	Starts a stopwatch timer in the CASL interpreter. It returns a descriptor number, which can be used to retrieve the amount of time that has elapsed since the timer started.
timer_stop()	Takes a descriptor number as an argument, stops the stopwatch timer associated with the descriptor, and returns the number of milliseconds that have elapsed since the timer was started.
tobuf()	Takes a list as an argument and returns a buffer containing the ordered contents of that list.
atoi()	Takes a string as an argument and returns the integer represented by that string.
wait()	Takes an integer as an argument, representing the number of seconds for the interpreter to wait before continuing.

getip()	Takes a string as an argument and returns a number representing the IP address contained in that string.
putip()	Takes a binary IP address as an argument and returns a string representing that IP address.
getenv()	Retrieves the specified environment variable (represented as a string), returning it's value as a string (or null if the variable is not set).
setenv()	Changes the value of the environment variable specified as it's first argument (a string) to the value represented by it's second argument.
strep()	Returns an ASCII string representation of an arbitrary variable, useful for obtaining strings representing integers.
exit()	Exits the CASL interpreter, taking an optimal argument of the exit code.
size()	Returns the size in bytes of a buffer argument, or the number of entries in a list argument.
rand()	Returns a pseudo random number. If an optional argument is given, the random number generated is seeded with that number.
gettimeofday()	Returns the time in milliseconds since midnight.

Summary

This chapter covered CASL. Specifically, this chapter:

- explained the benefits of writing programs in CASL
- introduced the main elements of a CASL program
- provided a reference section, which contains detailed descriptions of elements that can be used in CASL programs
- included a summary of CASL built-in functions that can be used in CASL programs

You can use the information provided in this chapter as reference material when writing your own CASL programs.