JHU IDS Module 7 Lab: Zeek

Audrey Long

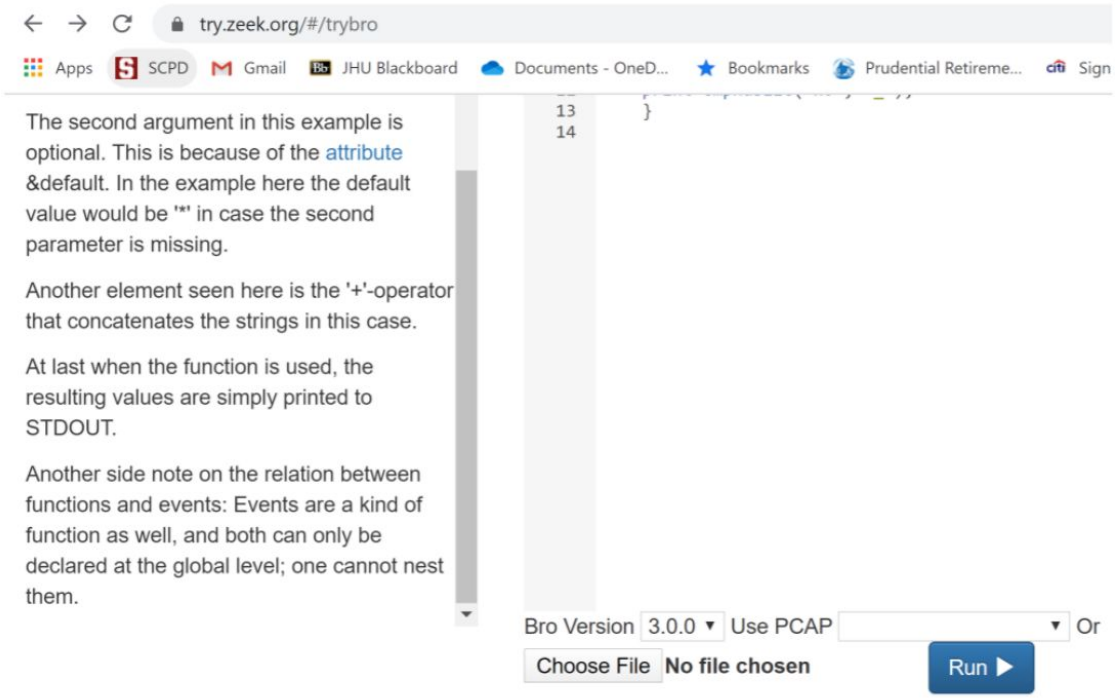
07/11/2020

Introduction
In this lab you'll be working with "Zeek" (the new name for "Bro") which has a programming language that enables you to extend Bro's functionality. Bro is the name of a popular, long-standing network analysis tool used for analyzing network traffic.

How to Brogram (Zeek)
There is a great Zeek tutorial here.

Complete each of the steps under the Basics section (Hello World through Exercise #2) and provide a screenshot of each completed section. If there are errors loading that page, the tutorials can also be found here and also provide a side-by-side view of the development IDE as well. Whichever you choose, please use **Bro 3.0.0 and exerceise_traffic.pcap** as your input. These settings can be found at the bottom of the IDE (shown below).

Throughout the tutorial you will find 2 Exercises. Please try to complete each one without consulting the solution. Provide a screenshot of the successful results (i.e., outputs) and a copy of your source code for each. In this week's Assignment we'll be applying your code to a Bro installation!



- Hello World

When you are ready you can just click on next below and start the next example.

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
Hello, World!
Goodbye, World!
```

Output Logs

capture_loss | conn | dhcp | dns | files | http | known_certs | known_hosts | known_services | notice | ntp | smtp | software | ssl | stats | weird | x509

ts	ts_delta	peer	gaps	acks	percent_lost
1258532133.914401	930.000003	zeek	0	0	0.0
1258533063.914399	929.999998	zeek	0	0	0.0
1258533977.316663	913.402264	zeek	0	0	0.0
1258534893.914434	916.597771	zeek	0	0	0.0
1258535805.364503	911.450069	zeek	0	45	0.0
1258536723.914407	918.549904	zeek	0	9	0.0
1258537653.914390	929.999983	zeek	0	0	0.0
1258538553.914414	900.000024	zeek	0	9	0.0
1258539453.914415	900.000001	zeek	0	0	0.0
1258540374.060134	920.145719	zeek	0	0	0.0
1258541283.914406	909.854272	zeek	0	0	0.0
1258542213.914408	930.000002	zeek	0	0	0.0
1258543143.914407	929.999999	zeek	0	18	0.0
1258544043.914430	900.000023	zeek	0	0	0.0
1258544973.914389	929.999959	zeek	0	39	0.0
1258545873.914404	900.000015	zeek	0	9	0.0
1258546803.914389	929.999985	zeek	0	0	0.0

- ## Loading Scripts

Example: basics: Loading Scripts Hide Text ✖

Previous Next

main.zeek + Add File

```
1 @load misc/dump-events
2
```

Loading Scripts

Like most programming languages, Zeek has the ability to load in script code from other files. There is a directive, `@load` which provides the capability.

The code here shows a simple script that does nothing but loading a script. The script misc/dump-events prints the events that Zeek generates out to standard output in a readable form. This is for debugging only and can be used to help understand events and their parameters. Note that it will show only events for which a handler is defined.

A small note needs to be made here because there are some default paths defined by Zeek automatically which make it easier to load many of the scripts that are included with Zeek. The default paths are as follows (based on the installed prefix directory):

- `<prefix>/share/bro`
- `<prefix>/share/bro/policy`
- `<prefix>/share/bro/site`

The most common use case of the load statement is in local.zeek. This file is part of Zeek's configuration files and adds further scripts that are not loaded by default. A reference of all scripts that can be loaded is found here. Everything you see there in base/ is loaded by default

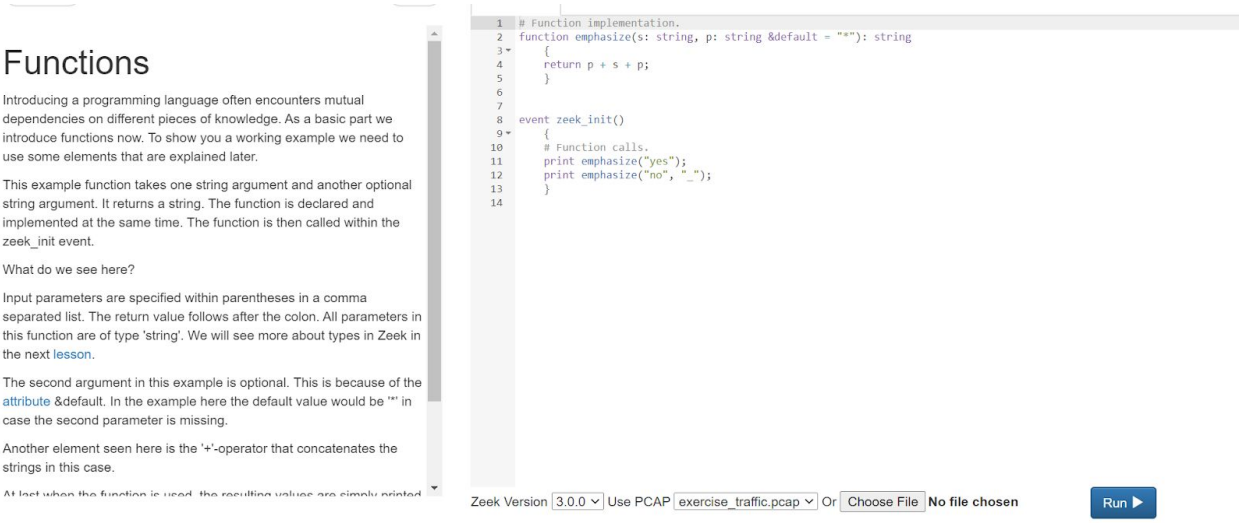
Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
1594494106.637979 zeek_init
1594494106.637979 zeek_script_loaded
                  [0] path: string       = /usr/local/zeek-3.1.3/share/zeek/base/init-bare.zeek
                  [1] level: count       = 0

1594494106.637979 zeek_script_loaded
                  [0] path: string       = /usr/local/zeek-3.1.3/share/zeek/base/bif/const.bif.zeek
                  [1] level: count       = 1
```

- Functions



Functions

Introducing a programming language often encounters mutual dependencies on different pieces of knowledge. As a basic part we introduce functions now. To show you a working example we need to use some elements that are explained later.

This example function takes one string argument and another optional string argument. It returns a string. The function is declared and implemented at the same time. The function is then called within the zeek_init event.

What do we see here?

Input parameters are specified within parentheses in a comma separated list. The return value follows after the colon. All parameters in this function are of type 'string'. We will see more about types in Zeek in the next lesson.

The second argument in this example is optional. This is because of the attribute &default. In the example here the default value would be '*' in case the second parameter is missing.

Another element seen here is the '+'-operator that concatenates the strings in this case.

At last when the function is used, the resulting values are simply printed

```
# Function implementation.
function emphasize(s: string, p: string &default = "*"): string
    {
    return p + s + p;
    }


event zeek_init()
    {
    # Function calls.
    print emphasize("yes");
    print emphasize("no", "_");
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File **No file chosen** Run ▶

Output

```
*yes*
_no_
```

Output Logs

capture_loss | conn | dhcp | dns | files | http | known_certs | known_hosts | known_services | notice | ntp | smtp | software | ssl | stats | weird | x509

ts	ts_delta	peer	gaps	acks	percent_lost

- Variables

Previous Next

Variables

You can assign arbitrary data to a variable in order to store it for later use. A `local' variable differs from a` global' in that its scope is restricted to the body of a function and will be assigned its initial value each time the function body is executed.

The reference on declaring variables, constants, functions etc is found here. More on types and all things that can be declared will follow in later lessons of this tutorial.

Run the example for this exercise. Try to print 'z' in the second event. Does that work?

main.zeek + Add File

```
1  global x = "Hello";
2
3  event zeek_init()
4      {
5      print x;
6
7      const y = "Guten Tag";
8      # Changing value of 'y' is not allowed.
9      #y = "Nope";
10
11     local z = "What does that mean?";
12     print z;
13     }
14
15 event zeek_done()
16     {
17     x = "Bye";
18     print x;
19     }
20
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File **No file cho**

Output

```
Hello
What does that mean?
Bye
```

Output Logs

- Primitive Datatypes

Primitive Datatypes

Now that we have variables we can talk about which data types we can use and assign to variables. In this lesson we introduce the simpler types.

Zeek has a static type system (i.e., the type of data a variable holds is fixed) with type inference, e.g., `local x = 0` is equivalent to `local x: count = 0`. It also implicitly promotes/coerces types in certain situations.

The full reference on types in Zeek can be found here. For now, look through the simple types. Most of the types should be familiar from other programming languages, e.g., `bool`, `double`, `int`, `count`, `string`, `pattern` (a regular expression using flex's syntax). But Zeek as a network monitoring system introduces also a set of domain-specific types that are explained in the reference. Examples are `time`, `interval`, `port`, `addr`, and `subnet`.

These custom Zeek types and the more complex types will be discussed in detailed examples in later lessons.

Run the code in this example. Try to play with the given code example, e.g. change the given types. Does that work?

```
event zeek_init()
    {
    local x : string = "two";
    local y : int = 10000000000000000000000000000000000000000000000000;
    print "y is a large int:", y;
    print "x is a short string:", x;

    #pattern matching
    print /one|two|three/ == "two";  # T
    print /one|two|three/ == "ones"; # F (exact matching)
    print /one|two|three/ in "ones"; # T (embedded matching)
    print /[123].*/ == "2 two";  # T
    print /[123].*/ == "4 four"; # F
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File **No file chosen** Run ▶

Output

```
y is a large int:, -1
x is a short string:, two
T
F
T
T
F
```

- Operators

Operators

So far we have functions, variables, and we can even type them. We still can't connect two (or more) values to build a new one. So now we can talk about operators that are used to manipulate, inspect, or compare data.

Explore the operators below to play with the Zeek elements we have so far. In the next two steps, we introduce loops and if-statements so that we can solve more complex exercises.

Arithmetic Operators

Name	Syntax	Example Usage
Addition	`a + b`	`print 2 + 2; # 4`
Subtraction	`a - b`	`print 2 - 2; # 0`
Multiplication	`a * b`	`print 4 * 4; # 16`
Division	`a / b`	`print 15 / 3; # 5`
Modulo	`a % b`	`print 18 % 15; # 3`
Unary Plus	`+a`	`local a = +1; # Force use of a signed integer`
Unary Minus	`-a`	`local a = 5; print -a; # -5`
Increment	`++a`	`local a = 1; print ++a, a; # 2, 2`
Decrement	`--a`	`local a = 2; print --a, a; # 1, 1`

```
event zeek_init()
    {
    print "Try to figure out what happens if ";
    print "you use numbers as strings and compare them with 'ints'";
    print "get to know Zeek's types and operators so you don't ";
    print "need to look them up all them time in later exercises.";
    }

event zeek_done()
    {
    print "Well done!";
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File **No file chosen** Run ▶

Output

```
Try to figure out what happens if
you use numbers as strings and compare them with 'ints'
get to know Zeek's types and operators so you don't
need to look them up all them time in later exercises.
Well done!
```

Output Logs

capture_logs conn dhcp dns files http known_certs known_hosts known_services notice ntp smtp software ssl stats weird x509

- If

Previous Next

If Statement

If statements conditionally execute another statement or block of statements. Play with the given example.

main.zeek + Add File

```
event zeek_init()
    {
    local x = "3";

    for ( c in "12345" )
        {
        if ( c == x )
            {
            print "Found it.";
            # A preview of functions: fmt() does substitutions, outputs result.
            print fmt("And by 'it', I mean %s.", x);
            }
        else
            {
            # A quick way to print multiple things on one line.
            print "I'm looking for", x, "not", c;
            }
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
I'm looking for, 3, not, 1
I'm looking for, 3, not, 2
Found it.
And by 'it', I mean 3.
I'm looking for, 3, not, 4
I'm looking for, 3, not, 5
```

- For Loops

Apps Mail - along36@jhu... Hacking: The Art of... Johns Hopkins Insti... Welcome, AUDREY... Cybersecurity | Joh... fc02.deviantart.net/... Open Data Structures OneDrive for Busin...

Previous Next

For Loops

Zeek provides a "foreach" style loop. In the given example we simply iterate through the string "abc" and print the current character.

Note: Iterating over any collection other than a vector won't provide any guarantee of the order Zeek iterates over the collection. If the order is important the collection should be a vector.

main.zeek + Add File

```
event zeek_init()
    {
    for ( character in "abc" )
        {
        print character;
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
a
b
c
```

Output Logs

capture_loss conn dhcp dns files http known_certs known_hosts known_services notice ntp smtp software ssl stats weird x509

- Loops: While

Previous Next

Loops: While

A "while" loop iterates over a body statement as long as a given condition remains true.

A break statement can be used at any time to immediately terminate the "while" loop, and a next statement can be used to skip to the next loop iteration.

main.zeek + Add File

```
event zeek_init()
    {
    local i = 0;

    while ( i < 5 )
    print ++i;

    while ( i % 2 != 0 )
    {
    local finish_up = F;

    if ( finish_up == F )
        print "nope";
        ++i;
        next;

    if ( finish_up )
        break;
    }
    print i;
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
1
2
3
4
5
nope
6
```

Output Logs

- Exercise

```
event zeek_init()
    {
        local excluded_letter = "e";
        local final_string = "";
        for (i in "find meeeeeee!")
        {
            if (i != excluded_letter)
            {
                final_string = final_string + i;
            }
        }
        print final_string;
    }
```

Output

```
find m!
```

```
event zeek_done()
    {
    local max_val = 100;
    local i = 1;
        while (i  <= max_val)
        {
            if ((i % 3 == 0) && (i % 5 == 0))
            {
                print "FizzBuzz";
            }
            else if ( i % 3 == 0 )
            {
                print "Fizz";
            }
            else if (i % 5 == 0)
            {
                print "Buzz";
            }
            else
            {
                print i;
            }
            i  = i + 1;

        }
    }
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
31
32
Fizz
34
Buzz
Fizz
```

```
Fizz
Buzz
56
Fizz
58
59
FizzBuzz
61
62
Fizz
64
Buzz
Fizz
67
68
Fizz
Buzz
71
Fizz
73
74
FizzBuzz
76
77
Fizz
79
Buzz
Fizz
82
83
Fizz
Buzz
86
Fizz
88
```

Exercise 1: Solution

Output

```
tsting
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
```

- The switch statement

Previous Next

The Switch Statement

Sometimes a switch statement is a more convenient way to organize code. For example, consider a switch instead of large chains of "else if" blocks if there's a large chain of OR'd conditions.

The syntax is similar to other common languages, "switch - variable - label". In Zeek it is possible to collect two or more label values to execute the same block of code. Also you can declare a default case if the input value does not match any of the cases. You must finish each case block with either "break" statement (to continue after the switch), or an explicit "fallthrough" to proceed into the subsequent case.

Now click "next" to solve an exercise using a switch.

main.zeek + Add File

```
event zeek_init()
    {
    local x = 4;

    switch ( x )
        {
        case 0:
            # This block only executes if x is 0.
            print "case 0";
            break;
        case 1, 2, 3:
            # This block executes if any of the case labels match.
            print "case 1, 2, 3";
            break;
        case 4:
            print "case 4 and ...";
            # Block ending in the "fallthrough" also execute subsequent case.
            fallthrough;
        case 5:
            # This block may execute if x is 4 or 5.
            print "case 5";
            break;
        default:
            # This block executed if no other case matches.
            print "default case";
            break;
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
case 4 and ...
case 5
```

Output Logs

capture_loss conn dhcp dns files http known_certs known_hosts known_services notice ntp smtp software ssl stats weird x509

ts	ts_delta	peer	gaps	acks	percent_lost

- Switch Exercise

Example: basics/switches: Switch Exercise Hide Text ✖

Previous Next

Switch Exercise

Write a program that relies on a switch statement to count the number of vowels (a, e, i, o, u) in an arbitrary string of your choosing.

main.zeek + Add File

```
    {
    switch (letter)
        {
        case "a":
            a_counter = a_counter + 1;
            break;

        case "e":
            e_counter = e_counter + 1;
            break;

        case "i":
            i_counter = i_counter + 1;
            break;
        case "o":
            o_counter = o_counter + 1;
            break;

        case "u":
            u_counter = u_counter + 1;
            break;

        default :
            break;
        }
    }

    print "a counter value: ", a_counter;
    print "e counter value: ", e_counter;
    print "i counter value: ", i_counter;
    print "o counter value: ", o_counter;
    print "u counter value: ", u_counter;
}
```

Zeek Version 3.1.3 Use PCAP Or Choose File No file chosen Run ▶

Output

```
write a switch to count vowels!
a counter value: , 4
e counter value: , 3
i counter value: , 3
o counter value: , 3
u counter value: , 2
```

- Switch Exercise: Solution

Example: basics/switches: Switch Exercise: Solution Hide Text ✖

Previous Next

Switch Exercise: Solution

Write a program that relies on a switch statement to count the number of vowels (a, e, i, o, u) in an arbitrary string of your choosing.

Here is an example that works. We loop through the string that is given in the variable `input`. Inside the for loop every character is sent to the switch to test if it is a vowel. If so, the variable `result` is incremented.

main.zeek + Add File

```
event zeek_init()
    {
    local result = 0;
    local input = "The Zeek Network Security Monitor";
    for ( c in input )
        {
        switch ( c )
            {
            case "a", "e", "i", "o", "u":
                ++result;
                break;
            }
        }
    print result;
    }
```

Zeek Version 3.1.3 Use PCAP Or Choose File **No file chosen** Run ▶

Output

```
11
```

Output Logs

- Event

Example: basics: Event Hide Text ✖

Previous Next

event, all handler bodies for that event are executed in order of `&priority`.

In the Zeek documentation, there is a detailed chapter about Zeek's event engine, how Zeek and the scripts interact, and what role the `event` plays in a Zeek script. Please read. A reference for predefined events not related to protocol or file analysis is here.

This example shows how to define and trigger a custom event.

- We first see an event declaration of "myevent" that takes the string "s".
- The the event handler implementation follows. The `&priority` attribute is optional and may be used to influence the order in which event handler bodies execute. If omitted, &priority is implicitly 0. In the example the priority is `-10` and thus very low. When this handler is called it will increment `n` from `0` to `1`.
- The next handler for the same event sets the priority to 10. This handler will print the string "myevent" and the current values of the variables `s` and `n`.
- Next we see the already familiar `zeek_init` event that is executed once when Zeek starts. It schedules the event twice. The first execution is a 'a soon as possible" schedule, the `schedule 5 sec {}` executes either in 5 seconds or upon Zeek shutting down, whichever happens first.

Run the code and follow the order in which the events are executed.

main.zeek + Add File

```
global myevent: event(s: string);

global n = 0;

event myevent(s: string) &priority = -10
    {
    ++n;
    }

event myevent(s: string) &priority = 10
    {
    print "myevent", s, n;
    }

event zeek_init()
    {
    print "zeek_init()";
    event myevent("hi");
    schedule 5 sec { myevent("bye") };
    }

event zeek_done()
    {
    print "zeek_done()";
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File **No file chosen** Run ▶

Output

```
zeek_init()
myevent, hi, 0
myevent, bye, 1
zeek_done()
```

- hook

Previous Next

Hook

Hooks are yet another flavor of function. They are similar to events in that they can also have multiple bodies. However they are different in two regards:

- They do execute immediately when invoked (i.e. they're not scheduled like events).
- The way the body of a hook handler terminates determines if further handlers get executed. If the end of the body, or a `return` statement, is reached, the next hook handler will be executed. If, however, a hook handler body terminates with a `break` statement, no remaining hook handlers will execute.

Hooks are useful to provide customization points for modules, as they allow to outsource decisions to site-specific code.

In this example we included the mentioned break statement, so the hook with priority `-5` is never executed. Try to play with this statement and the priorities to change the behavior of this example code.

main.zeek + Add File

```
global myhook: hook(s: string);

hook myhook(s: string) &priority = 10
    {
    print "priority 10 myhook handler", s;
    s = "bye";
    }

hook myhook(s: string)
    {
    print "break out of myhook handling", s;
    break;
    }

hook myhook(s: string) &priority = -5
    {
    print "not going to happen", s;
    }

event zeek_init()
    {
    local ret: bool = hook myhook("hi");
    if ( ret )
        {
        print "all handlers ran";
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
priority 10 myhook handler, hi
break out of myhook handling, hi
```

Output Logs

- Set

Set

A `set` is a collection of unique values. Sets use the `add` and `delete` operators to add and remove elements, and the `in` operator to test for membership.

Run the example.

In this example we first define a set of strings, containing the words "one", "two", and "three". The we add the string "four" to it. Thus the test for membership of "four" will result in "T" for true. The same way we can delete "two" from the set, testing if "two is not a member will result in "T" again. Adding the string "one" has no effect since it is already in the set. We can also use a for loop to print each member of a set.

```
event zeek_init()
    {
    local x: set[string] = { "one", "two", "three" };
    add x["four"];
    print "four" in x; # T
    delete x["two"];
    print "two" !in x; # T
    add x["one"]; # x is unmodified since 1 is already a member.

    for ( e in x )
        {
        print e;
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
T
T
three
one
four
```

- Table

Previous Next

Table

A `table` is an associative collection that maps a set of unique indices to other values. The same way as for sets, the `delete` operator is used to remove elements, however, adding elements is done just by assigning to an index as shown in this code example.

Tables are comparable to arrays, hashes, or maps in other languages.

Run the example. Most of it is the same as for sets. You can experiment with searching in sets and tables for example.

main.zeek + Add File

```
event zeek_init()
    {
    local x: table[count] of string = { [1] = "one",
                                        [3] = "three",
                                        [5] = "five" };
    x[7] = "seven";
    print 7 in x; # T
    print x[7]; # seven
    delete x[3];
    print 3 !in x; # T
    x[1] = "1"; # changed the value at index 1

    for ( key in x )
        {
        print key;
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen

Output

```
T
seven
T
5
7
1
```

- Vector

Vector

A `vector` is a collection of values with 0-based indexing. In comparison to sets this allows to store the same value twice.

Line 6 shows an example of the length operator. This line adds a new element at the end of the list.

```
event zeek_init()
    {
    local x: vector of string = { "one", "two", "three" };
    print x; # [one, two, three]
    print x[1]; # two
    x[|x|] = "one";
    print x; # [one, two, three, one]

    for ( i in x )
        {
        print i;  # Iterates over indices.
        }
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
[one, two, three]
two
[one, two, three, one]
0
1
2
3
```

- record

Previous Next

Record

A `record` is a user-defined collection of named values of heterogeneous types, similar to a struct in C. Fields are dereferenced via the `$` operator (`.`, as used in other languages, would be ambiguous in Zeek because of IPv4 address literals). Optional field existence is checked via the `?$` operator.

main.zeek + Add File

```
type MyRecord: record {
    a: string;
    b: count;
    c: bool &default = T;
    d: int &optional;
};

event zeek_init()
    {
    local x = MyRecord($a = "vvvvvv", $b = 6, $c = F, $d = -13);
    if ( x?$d )
        {
        print x$d;
        }

    x = MyRecord($a = "abc", $b = 3);
    print x$c;  # T (default value of the field)
    print x?$d; # F (optional field was not set)
    }
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen

Output

```
-13
T
F
```

Output Logs

- Redefinitions

Previous Next

Redefinitions for records

`redef` not only works with values, but also certain *types*. Namely `record` (and `enum`) may be extended, which is shown in this code example.

Redefining records is especially helpful when working with given modules.

Run this code example. Play with the the printing command, change the `redef` and see what effects happen.

main.zeek + Add File

```
type MyRecord: record {
    a: string &default="hi";
    b: count  &default=7;
} &redef;

redef record MyRecord += {
    c: bool &optional;
    d: bool &default=F;
    #e: bool; # Not allowed, must be &optional or &default.
};

event zeek_init()
    {
    print MyRecord();
    print MyRecord($c=T);
    }
```

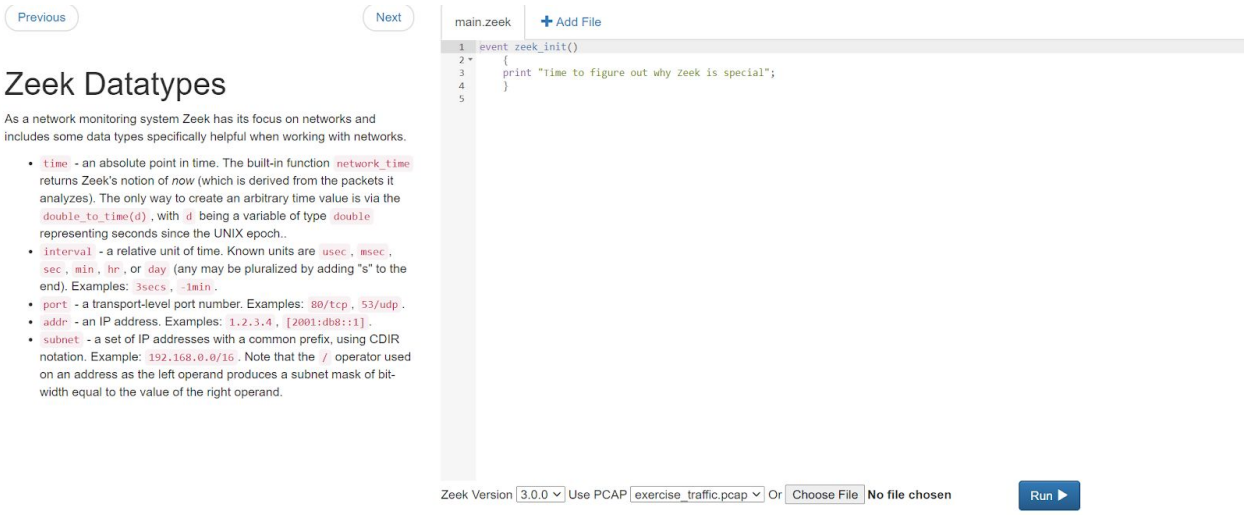
Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
[a=hi, b=7, c=<uninitialized>, d=F]
[a=hi, b=7, c=T, d=F]
```

Output Logs

- Bro Datatypes



- Exercise

```
global subnet_set: set[subnet] = { 192.168.1.0/24, 192.68.2.0/24, 172.16.0.0/20, 172.16.16.0/20, 172.16.32.0/20, 172.16.48.0/20 };
global my_count = 0;
global max_counter = 10;
global internal_networks: set[addr];
global external_networks: set[addr];

event new_connection(c: connection)
    {
    my_count = my_count + 1;
    if ( my_count <= max_counter )
        {
        print fmt("The connection %s from %s on port %s to %s on port %s started at %s.", c$uid, c$id$orig_h, c$id$orig_p, c$id$resp_h, c$id$resp_p, strftime("%D %H:%M", c$start_time));
    }
    if ( c$id$orig_h in subnet_set)
```

```
        {
        add internal_networks[c$id$orig_h];
      }
   else
      add external_networks[c$id$orig_h];

   if ( c$id$resp_h in subnet_set)
      {
      add internal_networks[c$id$resp_h];
      }
   else
      add external_networks[c$id$resp_h];
   }

event removed_connection(c: connection)
   {
   if ( my_count <= max_counter )
        {
        print fmt("The connection: %s took %s seconds", c$uid, c$duration);
        }
   }

event zeek_done()
   {
   print fmt("New connection: %d", my_count);
   print "local IPs:";
   for (ip_addr in internal_networks)
      {
      print ip_addr;
      }
   print "External IPs: ";
   for (ip_addr in external_networks)
      {
```

```
    print ip_addr;
    }
}
```

Example: basics/exercise2: Exercise Hide Text ✖

Previous Next

Exercise

By now we have all basic concepts of the Zeek scripting language. To finish the first part of your journey into the Zeek language solve the following exercise.

Consider the following list of subnets as your given local subnets:

192.168.1.0/24, 192.68.2.0/24, 172.16.0.0/20, 172.16.16.0/20, 172.16.32.0/20, 172.16.48.0/20.

Write a script that:

- tells for the first 10 new connections source IP and port, destinations IP and port, connection ID, time when the connection started.
- counts all connections seen and prints them in the end.
- prints out for each unique IP address if its is local or external.

To solve this exercise please load the traffic sample exercise_traffic.pcap.

main.zeek + Add File

```
1  global subnet_set: set[subnet] = { 192.168.1.0/24, 192.68.2.0/24, 172.16.0.0/20, 172.16.16.0/20, 172.16.32.0/20, 172.16.48.0/20 };
2  global my_count = 0;
3  global max_counter = 10;
4  global inside_networks: set[addr];
5  global outside_networks: set[addr];
6
7  event new_connection(c: connection)
8      {
9      my_count = my_count + 1;
10     if ( my_count <= max_counter )
11     {
12         print fmt("The connection %s from %s on port %s to %s on port %s started at %s.", c$uid, c$id$orig_h, c$id$orig_p, c$id$resp_h,
13     }
14     if ( c$id$orig_h in subnet_set)
15         {
16     add inside_networks[c$id$orig_h];
17         }
18     else
19         add outside_networks[c$id$orig_h];
20
21     if ( c$id$resp_h in subnet_set)
22         {
23         add inside_networks[c$id$resp_h];
24         }
25     else
26         add outside_networks[c$id$resp_h];
27     }
28
29 event connection_state_remove(c: connection)
30     {
31     if ( my_count <= max_counter )
32         {
33         print fmt("Connection %s took %s seconds", c$uid, c$duration);
34
```

Zeek Version 3.0.0 Use PCAP exercise_traffic.pcap Or Choose File No file chosen Run ▶

Output

```
The connection CMaAbQ1GoINBTr3vmd from 192.168.1.1 on port 626/udp to 224.0.0.1 on port 626/udp started at 11/18/09 08:00.
The connection C85ipw1K7dPtqnyP1j from 192.168.1.102 on port 68/udp to 192.168.1.1 on port 67/udp started at 11/18/09 08:00.
Connection C85ipw1K7dPtqnyP1j took 0.16382 seconds
The connection CLRuBi421cmD9Rh2P1 from 192.168.1.103 on port 138/udp to 192.168.1.255 on port 138/udp started at 11/18/09 08:07.
The connection CIoQu03b4LpP0ohJCg from 192.168.1.103 on port 137/udp to 192.168.1.255 on port 137/udp started at 11/18/09 08:08.
Connection CIoQu03b4LpP0ohJCg took 3.780125 seconds
The connection CY7LfMEN0EtXsY9Eh from 192.168.1.102 on port 137/udp to 192.168.1.255 on port 137/udp started at 11/18/09 08:08.
```

```
These IPs are considered external
98.137.88.34
198.189.255.81
66.235.138.19
65.54.95.77
87.106.13.61
198.189.255.76
209.84.4.126
fe80::219:e3ff:fee7:5d23
74.125.19.102
224.0.0.1
199.7.50.72
63.245.209.91
216.218.224.241
208.97.132.223
64.4.20.169
198.189.255.89
74.125.19.149
77.67.44.206
ff02::2
224.0.0.251
198.189.255.74
65.54.234.75
216.168.253.44
129.6.15.28
65.54.95.198
72.5.123.29
67.195.146.230
216.252.124.30
74.125.19.104
96.6.245.186
0.0.0.0
255.255.255.255
68.216.79.113
fe80::2c23:b96c:78d:e116
63.245.209.10
```