Bro Cheat Sheet

Version: November 15, 2011
Website: http://www.bro-ids.org
Email: info@bro-ids.org
Download: https://github.com/broids/cheat-sheet

Startup

`bro` *`[options] [file ...]`*

`file` ..... Bro policy script or `stdin`
`-e` *`code`* ..... Augment policies by given code
`-h` ..... Display command line options
`-i` *`iface`* ..... Read from given interface
`-p` *`pfx`* ..... Add given prefix to policy resolution
`-r` *`file`* ..... Read from given PCAP file
`-w` *`file`* ..... Write to given file in PCAP format
`-x` *`file`* ..... Print contents of state file
`-C` ..... Ignore invalid checksum

Language

Lowercase letters represent instance variables and uppercase letters represent types. In general, `x` is an instance of type `T` and `y` and instance of type `U`. Argument names and record fields begin begin with `a`, `b`, `...`, and `z` represents a default instance variable which takes on the type of the right-hand side expression. For notational convenience, `x` can often replaced with an expression of type `T`.

Variables

Constant qualifier ..... `const`
Constant redefinition ..... `redef x` *`op expr`*
Scope qualifier ..... `local`, `global`
Declaration ..... *`scope`* `x: T`
Declaration & Definition ..... *`scope`* `z =` *`expr`*

Declarations

Type ..... `type` *`name`*`: T`
Function ..... `function f(a: T, ...): R`
Event ..... `event e(a: T, ...)`

Modules

Script import ..... `@load` *`path`*
Set current namespace to `ns` ..... `module` *`ns`*
Export global symbols ..... `export { ... }`
Access `module` or `enum` namespace ..... `T::a`

Statements

Basic statement ..... *`stmt`*`;` or *`expr`*`;`
Code block ..... `{` *`stmt`*`; ... }`
Assignment ..... `z =` *`expr`*
Function assignment ..`z = function(...): R {..}`
Event queuing ..... `event e(...)`
Event scheduling ...`schedule 10 secs { e(...) }`
Print expression to `stdout` ..... `print` *`expr`*

Branching

```
if (expr)
    ...
else if (expr)
    ...
else
    ...
```

Iteration

```
for (i in x)
    ...
```

Conditional

```
when (expr)
    ...
```

Control

```
break
continue
next
return
```

Expressions

Operators

`!` ..... Negation
`$`, `?$` ..... Dereference, record field existence
`+`, `-`, `*`, `/`, `%` ..... Arithmetic
`++`, `--` ..... Post-increment, post-decrement
`+=`, `-=`, `*=`, `/=` ..... Arithmetic and assignment
`==`, `!=` ..... Equality, inequality
`<`, `<=`, `>=`, `>` ..... Less/greater than (or equal)
`&&`, `||` ..... Conjunction, disjunction
`in`, `!in` ..... Membership or pattern matching
`[x]` ..... Index strings and containers
`|x|` ..... Cardinality/size for strings and containers
`f(...)` ..... Function call
`expr` `?` *`expr`* `:` *`expr`* ..... Ternary if-then-else

Types

Basic

`addr` ..... IP address (`127.0.0.1`)
`bool` ..... Boolean flag (`T`, `F`)
`count` ..... 64-bit unsigned integer (`42`)
`double` ..... Double-precision floating point (`99.9`)
`int` ..... 64-bit signed integer (`-7`)
`interval` .... Time interval (`8 sec`/`min`/`hr`/`day[s]`)
`pattern` ..... Regular expression (`/^br[oO])$/`)
`port` ..... Transport-layer port (`22/tcp`, `53/udp`)
`string` ..... String of bytes (`"foo"`)
`subnet` ..... CIDR subnet mask (`10.0.0.0/8`)
`time` ..... Absolute epoch time (`1320977325`)

Enumerables

Declaration ..... `enum { FOO, BAR }`
Assignment ..... *`scope`* `x = FOO`

Records

Declaration ..... `record { a: T, b: U, ... }`
Assignment ..... *`scope`* `r = [$a=x, $b=y, ...]`
Access ..... `z = r$a`
Assignment ..... `r$b = y`
Deletion ..... `delete r$a`

Sets

Declaration ..... `set[T]`
Constructor ..... `set()`
Assignment ..... *`scope`* `s = { x, ... }`
Access ..... `z = s[x]`
Addition ..... `add s[x]`
Deletion ..... `delete s[x]`

Tables

Declaration ..... `table[T] of U`
Constructor ..... `table()`
Assignment ..... *`scope`* `t = { [x] = y, ... }`
Access ..... `z = t[x]`
Assignment ..... `add t[x] = y`
Deletion ..... `delete t[x]`

Vectors

Declaration ..... `vector of T`
Constructor ..... `vector()`
Assignment ..... *`scope`* `v = { x, ... }`
Access ..... `z = v[0]`
Assignment ..... `v[42] = x`

Attributes

Attributes occur at the end of type/event declarations and change their behavior. The syntax is `&key` or `&key=val`, e.g., `type T: set[count] &read_expire=5min` or `event foo() &priority=-3`.

`&optional` ..... Allow record field to be missing
`&default=x` ..... Use default value `x` for record fields and container elements
`&redef` ..... Make constants redefinable
`&expire_func=f` ..... Call `f` right before container element expires
`&read_expire=x` ..... Remove element after not reading it for time `x`
`&write_expire=x` ..... Remove element after not writing it for time `x`
`&create_expire=x` ..... Remove element after time `x` from insertion
`&persistent` ..... Write state to disk (per default on shutdown)
`&synchronized` ..... Synchronize variable across nodes
`&raw_output` ..... Do not escape non-ASCII characters when writing to a file
`&mergeable` ..... Prefer set union to assignment for synchronized state
`&priority=x` ..... Execution priority of event handler
`&group="x"` ..... Events in the same group can be jointly activated/deactivated
`&log` ..... Write record field to log

Built-In Functions (BIFs)

Core

- `length(v: any): count`
  Returns the number of elements in the container `v`.
- `same_object(o1: any, o2: any): bool`
  Check whether `o1` and `o2` reference the same internal object.
- `clear_table(v: any): any`
  Remove all elements from the set or table `v`.

String Processing

- `byte_len(s: string): count`
  Returns the number of characters (i.e., bytes) of `s`.
- `sub_bytes(s: string, start: count, n: int): string`
  Get a substring of `s`, starting at position `start` and having length `n`.
- `split(s string, re: pattern): table[count] of string`
  Split `s` into an array using `re` to separate the elements.
- `split1(s string, re: pattern): table[count] of string`
  Same as `split`, but `s` is only split once (if possible) at the earliest position and an array of two strings is returned. An array of one string is returned when `s` cannot be split.
- `split_all(s: string, re: pattern): table[count] of string`
  Same as `split`, but also include the matching separators, e.g., `split_all("a-b--cd", /(\-)+/)` returns `{"a", "-", "b", "--", "cd"}`. Odd-indexed elements do not match the pattern and even-indexed ones do.
- `split_n(s: string, re: pattern, incl_sep: bool, max_num_sep: count): table[count] of string`
  Similar to `split1` and `split_all`, but `incl_sep` indicates whether to include matching separators and `max_num_sep` the number of times to split `s`.
- `sub(s: string, re: pattern, repl: string): string`
  Substitutes `repl` for the first occurrence of `re` in `s`.
- `gsub(s string, re: pattern, repl: string): string`
  Same as `sub` except that all occurrences of `re` are replaced.
- `strcmp(s1: string, s2: string): int` Lexicographically compare `s1` and `s2`. Returns an integer greater than, equal to, or less than 0 according as `s1` is greater than, equal to, or less than `s2`.
- `strstr(big: string, little: string): count`
  Locate the first occurence of `litle` in `big`. Returns 0 if `little` is not found in `big`.
- `subst_string(s: string, from: string, to: string): string`
  Substitute each (non-overlapping) appearance of `from` in `s` to `to`, and return the resulting string.
- `to_lower(s: string): string`
  Returns a copy of `s` with each letter converted to lower case.
- `to_upper(s: string): string`
  Returns a copy of `s` with each letter converted to upper case.
- `clean(s: string): string`
  Replace non-printable characters in `s` with escaped sequences, with the mappings `0` $\rightarrow$ `\0`, `DEL` $\rightarrow$ `^?`, values $\leq$ 26 $\rightarrow$ `^[A-Z]`, and values not in $[32, 126]$ $\rightarrow$ `%XX`.
- `to_string_literal(s: string): string`
  Same as clean, but with different mappings: values not in $[32, 126]$ $\rightarrow$ `%XX`, `\` $\rightarrow$ `\\`, `'` $\rightarrow$ `\'`, `"` $\rightarrow$ `\"`.
- `is_ascii(s: string): bool`
  Returns false if any byte value of `s` is greater than 127, and true otherwise.
- `escape_string(s: string): string`
  Returns a printable version of `s`. Same as `clean` except that non-printable characters are removed.
- `string_to_ascii_hex(s: string): string`
  Returns an ASCII hexadecimal representation of a string.
- `str_split(s: string, idx: vector of count): vector of string`
  Splits `s` into substrings, taking all the indices in `idx` as cutting points; `idx` does not need to be sorted, and can have multiple entries. Out-of-bounds indices are ignored.

- `strip(s: string): string`
  Strips whitespace at both ends of `s`.
- `string_fill(len: int, source: string): string`
  Generates a string of size `len` and fills it with repetitions of `source`.
- `str_shell_escape(source: string): string`
  Takes a string and escapes characters that would allow execution of commands at the shell level. Must be used before including strings in `system()` or similar calls.
- `find_all(s: string, re: pattern) : string_set`
  Returns all occurrences of `re` in `s` (or an empty empty set if none).
- `find_last(s: string, re: pattern) : string`
  Returns the last occurrence of `re` in `s`. If not found, returns an empty string. Note that this function returns the match that starts at the largest index in the string, which is not necessarily the longest match. For example, a pattern of `/.*/` will return the final character in the string.
- `hexdump(data: string) : string`
  Returns a hex dump for `data`. The hex dump renders 16 bytes per line, with hex on the left and ASCII (where printable) on the right. Based on Netdude's hex editor code.