Bro Cheat Sheet

Version:	November 18, 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` an 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 be 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)
    { ... }
```

Asynchronous

```
when (expr) { ... }
when (local x = 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, ... }`
Constructor	`record($a=x, $b=y, ...)`
Assignment	*scope* `r = [$a=x, $b=y, ...]`
Access	`z = r$a`
Field assignment	`r$b = y`
Deletion	`delete r$a`

Sets

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

Tables

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

Vectors

Declaration	`vector of T`
Constructor	`vector(x, ...)`
Assignment	*scope* `v = { x, ... }`
Access	`z = v[0]`
Insertion	`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`	Allow for redefinition of initial object value
`&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, high to low, $x \in [-10, 10]$
`&group="x"`	Events in the same group can be jointly activated/deactivated
`&log`	Write record field to log

Built-In Functions (BIFs)

Core

- `current_time(): time`
  Returns the current wall-clock time.
- `network_time(): time`
  Returns the timestamp of the last packet processed.
- `reading_live_traffic(): bool`
  Checks whether Bro reads traffic from a network interface (as opposed to from a network trace).
- `reading_traces(): bool`
  Checks whether Bro reads traffic from a trace file (as opposed to from a network interface).
- `getenv(var: string): string`
  Returns the system environment variable identified by `var`.
- `setenv(var: string, val: string): bool`
  Sets the system environment variable `var` to `val`.
- `exit(): int` Shuts down the Bro process immediately and returns 0.
- `terminate(): bool` Gracefully shut down Bro by terminating outstanding processing. Returns true after successful termination and false when Bro is still in the process of shutting down.
- `system(s: string): int`
  Invokes a command via the `system` function. Returns true if the return value of `system` was non-zero.
- `system_env(s: string, env: any): int`
  Same as `system`, but prepare the environment before invoking the command `s` with the set/table `env`.
- `skip_further_processing(id: conn_id): bool`
  Stops processing packets belonging to the connection identified by `id`. Returns false if `id` does not point to an active connection and true otherwise. Note that this does not in itself imply that packets from this connection will not be recorded, which is controlled separately by `set_record_packets`. *TODO: Someone please verify this.*
- `set_record_packets(id: conn_id, do_record: bool): bool`
  Controls whether packet contents belonging to the connection identified by `id` should be recorded. Note that this is independent of whether Bro processes the packets of this connection, which is controlled separately by `skip_further_processing`. *TODO: Someone please verify this.*
- `set_contents_file(id: conn_id, direction: count, f: file): bool`
  Associates the file handle `f` with the connection identified by `id` for writing TCP byte stream contents. The argument `direction` controls what sides of the connection contents are recorded; it can take on four values: `CONTENTS_NONE` to turn off recording of contents, `CONTENTS_ORIG` to record originator contents, `CONTENTS_RESP` to record responder contents, and `CONTENTS_BOTH` to record both originator and responder contents. Returns false if `id` does not point to an active connection and true otherwise.
- `get_contents_file(id: conn_id, direction: count): file`
  Returns the file handle associated with the connection identified by `id` and `direction`. If the connection exists but no contents file for `direction`, the function returns a handle to new file. If not active connection for `id` exists, it returns an error.
- `same_object(o1: any, o2: any): bool`
  Checks whether `o1` and `o2` reference the same internal object.
- `length(v: any): count`
  Returns the number of elements in the container `v`.
- `clear_table(v: any): any`
  Removes all elements from the set or table `v`.

Files and Directories

- `open(f: string): file`
  Opens the file identified by `f` for writing. Returns a handle for subsequent file operations.
- `open_for_append(f: string): file`
  Same as `open`, except that `f` is not overwritten and content is appended at the

end of the file.

- `close(f: file): bool`
  Closes the file handle `f` and flushes buffered content. Returns true on success.
- `active_file(f: file): bool`
  Checks whether `f` is open.
- `write_file(f: file, data: string): bool`
  Writes `data` to `f`. Returns true on success.
- `get_file_name(f: file): string`
  Returns the filename of the file identified by the handle `f`.
- `set_buf(f: file, buffered: bool): any`
  Alters the buffering behavior of `f`. When `buffered` is true, the file is fully buffered, i.e., bytes are saved in a buffered until the block size has been reached. When `buffered` is false, the file is line buffered, i.e., bytes are saved up until a newline occurs. The return value is `void` and can be discarded. *TODO: Why is it not void then?*
- `flush_all(): bool`
  Flushes all open files to disk. Returns true when the operations(s) succeeded.
- `mkdir(f: string): bool`
  Creates a new directory identified by `f`. Returns true if the operation succeeded and `f` does not exist already.

Conversion

- `cat(...): string`
  Concatenates all given arguments into a single string.
- `cat_sep(sep: string, default: string, ...): string`
  Similar to cat, but places `sep` between each given argument. If any of the variable arguments is an empty string it is replaced by `default` instead.
- `fmt(...): string`
  Produces a formatted string. The first argument is the *format string* and specifies how subsequent arguments are converted for output. It is composed of zero or more directives: ordinary characters (not `%`), which are copied unchanged to the output, and conversion specifications, each of which fetches zero or more subsequent arguments. Conversion specifications begin with `%` and the arguments must properly correspond to the specifier.
  After the `%`, the following characters may appear in sequence:

`%`		Literal `%`
`-`		Left-align field
`[0-9]+`		The field width (< 128)
`.`		Precision of floating point specifiers `[efg]` (< 128)
`A`		Escape NUL bytes, i.e., replace `0` with `\0`
`[DTdxsefg]`		Format specifier
	`[DT]`	ISO timestamp with microsecond precision
	`d`	Signed/Unsigned integer (using C-style `%lld`/`%llu` for `int`/`count`)
	`x`	Unsigned hexadecimal (using C-style `%llx`); addresses/ports are converted to host-byte order
	`s`	Escaped string
	`[efg]`	Double

- `type_name(t: any): string`
  Returns the type name of `t`.
- `record_type_to_vector(rt: string): vector of string`
  Converts the record type name `rt` into a vector of strings, where each element is the name of a record field. Nested records are flattened.
- `to_int(s: string): int`
  Converts a `string` into a (signed) integer.
- `int_to_count(n: int): count`
  Converts a positive integer into a `count` or returns 0 if `n < 0`.
- `double_to_count(d: double): count`
  Converts a positive `double` into a `count` or returns 0 if `d < 0.0`.
- `to_count(s: string): count`
  Converts a `string` into a `count`.
- `interval_to_double(i: interval): double`
  Converts an `interval` time span into a `double`.
- `double_to_interval(d: double): interval`
  Converts a `double` into an `interval`.
- `time_to_double(t: time): double`
  Converts a `time` value into a `double`.
- `double_to_time(d: double): time`
  Converts a `double` into a `time` value.
- `double_to_time(d: double): time`
  Converts a `double` into a `time` value.
- `port_to_count(p: port): count`
  Returns the port number of `p` as `count`.
- `count_to_port(c: count, t: transport_proto): port`
  Create a `port` with number `c` and transport protocol `t`.
- `to_port(c: count, t: transport_proto): port`
  Same as `count_to_port`.

- `addr_to_count(a: addr): count`
  Converts an IP address into a 32-bit unsigned integer.
- `count_to_v4_addr(ip: count): addr`
  Converts an unsigned integer into an IP address.
- `to_addr(ip: string): addr`
  Converts a `string` into an IP address.
- `raw_bytes_to_v4_addr(b: string): addr`
  Converts a `string` of bytes into an IP address. It interprets the first 4 bytes of `b` as an IPv4 address in network order.

Network Type Processing

- `mask_addr(a: addr, top_bits_to_keep: count): subnet`
  Creates a subnet mask from `a` by specifying the number of top bits to keep. For example, `mask_addr(10.5.1.3, 8)` would return `10.0.0.0/8`.
- `remask_addr(a1: addr, a2: addr, top_bits_from_a1: count): count`
  Takes some top bits (e.g., subnet address) from `a1` and the other bits (intra-subnet part) from `a2` and merge them to get a new address. This is useful for anonymizing at subnet level while preserving serial scans.
- `is_tcp_port(p: port): bool`
  Checks whether `p` is a TCP port.
- `is_udp_port(p: port): bool`
  Checks whether `p` is a UDP port.
- `is_icmp_port(p: port): bool`
  Checks whether `p` is an ICMP port.
- `active_connection(id: conn_id): bool`
  Checks whether the connection identified by `id` is (still) active.
- `connection_exists(id: conn_id): bool`
  Same as `active_connection`. *TODO: Which one should we remove?*
- `connection_record(id: conn_id): connection`
  Returns the `connection` record for `id`. Note that you *must* first make sure that the connection is active (e.g., by calling `active_connection`).
- `lookup_connection(id: conn_id): connection`
  Same as `lookup_connection`. *TODO: Which one should we remove?*
- `get_conn_transport_proto(id: conn_id): transport_proto`
  Returns the transport protocol of the connection identified by `id`. As with `connection_record`, `id` must point to an active connection.
- `get_port_transport_proto(p: port): transport_proto`
  Returns the transport protocol of `p`.

Math

- `floor(x: double): double`
  Chops off any decimal digits of `x`, i.e., computes $\lfloor \mathtt{x} \rfloor$.
- `sqrt(x: double): double`
  Returns the square root of `x`, i.e., computes $\sqrt{\mathtt{x}}$.
- `exp(x: double): double`
  Raises e to the power of `x`, i.e., computes $e^{\mathtt{x}}$.
- `ln(x: double): double`
  Returns the natural logarithm of `x`, i.e., computes $\ln \mathtt{x}$.
- `log10(x: double): double`
  Returns the common logarithm of `x`, i.e., computes $\log_{10} \mathtt{x}$.

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. The returned table starts at index 1. Note that conceptually the return value is meant to be a vector and this might change in the future.
- `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 occurrence of `little` 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` → `\0`, `DEL` → `^?`, values ≤ 26 → `^[A-Z]`, and values not in $[32, 126]$ → `%XX`.
- `to_string_literal(s: string): string`
  Same as clean, but with different mappings: values not in $[32, 126]$ → `%XX`, `\` → `\\`, `'` → `\'`, `"` → `\"`.
- `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 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) : set of string`
  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.