Bro Cheat Sheet

Version: November 28, 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 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, higher values first, default 0
`&group="x"`	Events in the same group can be jointly activated/deactivated
`&log`	Write record field to log

Built-In Functions (BIFs)

Core

- `getenv(var: string): string`
  Returns the system environment variable identified by `var`, or an empty string if it is not defined.
- `setenv(var: string, val: string): bool`
  Sets the system environment variable `var` to `val`.
- `syslog(s: string)`
  Send the string `s` to syslog.
- `system(s: string): int`
  Invokes a command via the `system` function. Returns true if the return value of `system` was non-zero. Returns the return value from the `system()` call. Note that this corresponds to the status of backgrounding the given command, not to the exit status of the command itself. A value of 127 corresponds to a failure to execute `sh`, and -1 to an internal system failure. Furthermore, the command is run in the background with `stdout` redirected to `stderr`. Here is a usage example: `system(fmt("rm \"%s\"", str_shell_escape(sniffed_data)));`
- `system_env(s: string, env: any): int`
  Same as `system`, but prepare the environment before invoking the command `s` with the set/table `env`.
- `piped_exec(program: string, to_write: string): bool`
  Opens the application `program` with `popen` and writes the string `to_write` to `stdin` of the opened program.
- `srand(seed: count)`
  Set the seed for subsequent `rand` calls.
- `rand(max: count): count`
  Returns a random value from the interval $[0, \texttt{max})$.
- `md5_hash(...): string`
  Computes the MD5 hash value of the provided list of arguments.
- `md5_hash_init(index: any): bool`
  Initializes MD5 state for `index` to allow for computing hash values incrementally via the function `md5_hash_update`. For example, when computing incremental MD5 values of transferred files in multiple concurrent HTTP connections, it is necessary to call `md5_hash_init(c$id)` once before invoking `md5_hash_update(c$id, some_more_data)` in the `http_entity_data` event handler.
- `function md5_hash_update(index: any, data: string): bool`
  Update the MD5 value associated with `index`. Note that it is necessary to call `md5_hash_init(index)` once before calling this function to initialize the MD5 state.
- `md5_hash_finish(index: any): string`
  Returns the final MD5 digest associated with the internal state identified by `index`.
- `md5_hmac(...): string`
  Computes an HMAC-MD5 hash value of the provided list of arguments. The HMAC secret key is generated from available entropy when Bro starts up, or it can be specified for repeatability using the `-K` flag.
- `file_size(f: string): double`
  Returns the file size in bytes of the file identified by `f`.
- `strftime(fmt: string, d: time): string`
  Formats the time value `d` according to the format string `fmt`. See `man strftime` for the format of `fmt`.
- `lookup_addr(host: addr): string`
  Issues an asynchronous reverse DNS lookup and delays the function result. Therefore, it can only be called inside a `when`-condition, e.g., `when ( local host = lookup_addr(10.0.0.1) ) { f(host); }`. Returns the DNS name of `host`.
- `lookup_hostname(host: string): set[addr]`
  Issues an asynchronous DNS lookup and delays the function result. Returns a set containing the addresses that `host` resolves to. See `lookup_addr` for a usage example.
- `identify_data(data: string, return_mime: bool): string`
  Invokes `libmagic` on `data` to determine its MIME type. If `return_mime` is true, the function returns a MIME type string instead of a textual description.

- `unique_id(prefix: string): string`
  Creates an identifier that is unique with high probability, with `prefix` prepended to the result.
- `unique_id_from(pool: int, prefix: string): string`
  Same as `unique_id`, except that the additional argument `pool` specifies a seed for determinism.
- `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.
- `exit()` Shuts down the Bro process immediately.

Packet Filtering

- `precompile_pcap_filter(id: PcapFilterID, s: string): bool`
  Precompiles the PCAP filter `s` and binds it to the identifier `id` in `libpcap`. Returns true if the filter expression is valid. See `install_pcap_filter`.
- `install_pcap_filter(id: PcapFilterID): bool`
  Installs a PCAP filter precompiled via `precompile_pcap_filter`. Returns true if the installation succeeds.
- `install_src_addr_filter(ip: addr, flags: count, p double): bool`
  Installs a filter to drop packets from the IP source address `ip` with probability $p \in [0, 1]$ if none of the TCP flags given by `flags` are set.
- `install_src_net_filter(s: subnet, flags: count, p: double): bool`
  Same as `install_src_addr_filter` but for subnets instead of IP addresses.
- `uninstall_src_addr_filter(ip: addr): bool`
  Removes an IP source address filter for `ip` installed with `install_src_addr_filter`.
- `uninstall_src_net_filter(snet: subnet): bool`
  Removes an IP source subnet filter for `snet` installed with `install_src_net_filter`.
- `install_dst_addr_filter(ip: addr, flags: count, p: double): bool`
  Same as `install_src_addr_filter` but for IP destination addresses.
- `install_dst_net_filter(s: snet, flags: count, p: double): bool`
  Same as `install_dst_addr_filter` but for subnets instead of IP addresses.
- `uninstall_dst_addr_filter(ip: addr): bool`
  Removes an IP destination address filter for `ip` installed with `install_dst_addr_filter`.
- `uninstall_dst_net_filter(snet: subnet): bool`
  Removes an IP destination subnet filter for `snet` installed with `install_dst_net_filter`.
- `pcap_error(): string`
  Returns a descriptive error message if the last PCAP function failed.

Introspection

- `bro_version(): string`
  Returns the Bro version string.
- `getpid(): count`
  Returns Bro's process ID.
- `do_profiling()`
  Enable detailed collections of statistics about CPU/memory usage, connections, TCP states/reassembler, DNS lookups, timers, and script-level state. The script variable `profiling_file` holds the name of the log file.
- `gethostname(): string`
  Get the value of the hostname of the machine Bro runs on.
- `current_time(): time`
  Returns the current wall-clock time.
- `network_time(): time`
  Returns the timestamp of the last packet processed. Returns the timestamp of the most recently read packet, whether read from a live network interface or from a save file. Compare against `current_time`. In general, you should use `network_time` unless you're using Bro for non-networking uses (such as general scripting; not particularly recommended), because otherwise your script may behave very differently on live traffic versus played-back traffic from a save file.
- `reading_live_traffic(): bool`
  Checks whether Bro reads traffic from one or more network interfaces (as opposed to from a network trace in a file). Note that this function returns true even after Bro has stopped reading network traffic, for example due to receiving a termination signal.
- `reading_traces(): bool`
  Checks whether Bro reads traffic from a trace file (as opposed to from a network interface).
- `bro_is_terminating(): bool`
  Returns true if Bro is in the process of shutting down.
- `net_stats(): NetStats`
  Returns statistics about the number of packets *(i)* received by Bro, *(ii)* dropped, and *(iii)* seen on the link (not always available).
- `resource_usage(): bro_resources`
  Returns Bro process statistics, such as real/user/sys CPU time, memory usage, page faults, number of TCP/UDP/ICMP connections, timers, and events queued/dispatched.
- `get_matcher_stats(): matcher_stats`
  Returns statistics about the regular expression engine, such as the number of distinct matchers, DFA states, DFA state transitions, memory usage of DFA states, cache hits/misses, and average number of NFA states across all matchers.

- `dump_rule_stats(f: file): bool`
  Write rule matcher statistics (DFA states, transitions, memory usage, cache hits/misses) to the file `f`. Returns true on success.
- `get_gap_summary(): gap_info`
  Returns statistics about TCP gaps.
- `global_sizes(): table[string] of count`
  Returns a table containing the size of all global variables, where the index is the variable name and the value the variable size in bytes.
- `global_ids(): table[string] of script_id`
  Returns a table with information about all global identifiers. The table value is a record containing the type name of the identifier, whether it is exported, a constant, an enum constant, redefinable, and its value (if it has one).
- `lookup_ID(id: string): any`
  Returns the value of associated with the global identifier `id`. If `id` does not describe a valid identifier, the function returns the string `"<unknown id>"` or `"<no ID value>"`.
- `record_fields(r: any): table[string] of record_field`
  Returns meta data about a record instance `r`, which includes the type name, whether the field is logged, its value (if it has one), and its default value (if specified).
- `is_local_interface(ip: addr): bool`
  Returns true if the address `ip` is a valid DNS entry for `localhost`.
- `is_external_connection(c: connection): bool`
  Returns true if the connection `c` has been received externally. *TODO: What does this mean?*
- `disable_print_hook(f: file)`
  Function equivalent to the `&disable_print_hook` attribute, *TODO: which does what?*
- `enable_raw_output(f: file)`
  Function equivalent to the `&raw_output` attribute, which prevents escaping of non-ASCII characters when writing to `f`.
- `enable_communication()`
  Enables the communication system. By default, communication is off until explicitly enabled and all other calls to communication-related BiF's will be ignored until done so.
- `suspend_processing()`
  Stop Bro's packet processing. *TODO: When would that be called?*
- `continue_processing()`
  Resume Bro's packet processing. *TODO: When would that be called?*
- `suspend_state_updates()`
  Stop propagating `&synchronized` accesses.
- `resume_state_updates()`
  Resume propagating `&synchronized` accesses.
- `enable_event_group(group: string)`
  Enables all event handlers in the group `group`. This affects all handlers that have been tagged with the attribute `&group="group"`.
- `disable_event_group(group: string)`
  Disables all event handlers in the group `group`. This affects all handlers that have been tagged with the attribute `&group="group"`.

Independent State

- `checkpoint_state(): bool`
  Flushes in-memory state with the `&persistence` attribute to the state file `.state/state.bst`.
- `dump_config(): bool`
  Flushes configuration into the file `.state/config.bst`. *TODO: What is meant with "configuration?"*
- `rescan_state(): bool`
  Reads persistent configuration and state from the `.state` directory.
- `capture_events(filename: string): bool`
  Writes the event stream generated by the core to `filename`. Use the `-x` command line switch to replay the saved events.
- `capture_state_updates(filename: string): bool`
  Writes state updates generated by `&synchronized` variables to the file `filename`. *TODO: please double-check*
- `connect(ip: addr, p: port, our_class: string, retry: interval,`
  `ssl: bool): count`
  Establishes a connection to a remote Bro instance or Broccoli application at IP address `ip` and port `p`. If the connection fails, Bro tries to reconnect with the peer after the time interval `retry`. If `ssl` is true, the connection uses uses SSL to encrypt the session. Returns the locally unique ID of the new peer. *TODO: what does `our_class` do?*
- `disconnect(p: event_peer): bool`
  Disconnects the peer identified by `p`.
- `request_remote_events(p: event_peer, handlers: pattern): bool`
  Subscribes to all events from remote peer `p` whose names match the pattern `handlers`.
- `request_remote_sync(p: event_peer, auth: bool): bool`
  Requests synchronization of IDs with remote peer `p`. If `auth` is true, the local Bro instance considers its current state authoritative and sends it to `p` right after the handshake.
- `request_remote_logs(p: event_peer): bool`
  Requests logs from remote peer `p`. *TODO: what does it do exactly?* Returns true on success.

- `set_accept_state(p: event_peer, accept: bool): bool`
  Sets a boolean flag whether Bro accept state from the remote peer `p`. Returns true on success.
- `set_compression_level(p: event_peer, level: count): bool`
  Sets the compression level of the session with remote peer `p`. values for `level` are in $[0, 9]$, where 0 is the default and means no compression) Returns true on success.
- `listen(ip: addr, p: port, ssl: bool): bool`
  Listens on address `ip` and port `p` for remote connections. If `ssl` is true, the Bro uses SSL to encrypt the session. Returns true on success.
- `is_remote_event(): bool`
  Returns true if the last raised event stemmed from a remote peer.
- `send_state(p: event_peer): bool`
  Sends all persistent state to the remote peer `p`. Returns true on success.
- `send_id(p: event_peer, id: string): bool`
  Send the value of the global identifier `id` to the remote peer `p`, which might then install it locally.
- `terminate_communication(): bool`
  Gracefully finishes communication by first making sure that all remaining data from parent and child has been sent out. Returns true if the termination process has been started successfully.
- `complete_handshake(p: event_peer): bool`
  Signals the remote peer `p` that the local Bro instance finished the initial handshake. Returns true on success.
- `send_ping(p: event_peer, seq: count): bool`
  Sends a ping with a sequence number `seq` to the remote peer `p`. Returns true on success. *TODO: what's the purpose of this function? Measuring latency?*
- `send_current_packet(p: event_peer): bool`
  Sends the currently processed packet to the remote peer `p`. Returns true on success.
- `get_event_peer(): event_peer`
  Returns the peer who generated that last raised event.
- `get_local_event_peer(): event_peer`
  Returns the local peer.
- `send_capture_filter(p: event_peer, s: string): bool`
  Sends the capture filter `s` to the remote peer `p`. Returns true on success.
- `make_connection_persistent(c: connection)`
  Makes the connection `c` persistent. *TODO: What does it mean for a connection to be persistent?*

Analyzer Behavior

- `current_analyzer(): count`
  Returns the ID of the analyzer which raised the current event, or 0 if no analyzer has been instantiated.
- `analyzer_name(aid: count): string`
  Translates the analyzer ID `aid` to a string representation.
- `expect_connection(orig: addr, resp: addr, resp_p: port,`
  `analyzer: count, tout: interval): bool`
  Schedules the analyzer identified by the ID `analyzer` for a future connection from IP address `orig` to `resp` at port `resp_p`. The function ignores the scheduling request if the connection did not occur within the specified time interval `tout`.
- `disable_analyzer(id: conn_id, aid: count): bool`
  Disables the analyzer `aid` which raised the current event if it belongs to connection identified by `id`.
- `skip_further_processing(id: conn_id): bool`
  Informs Bro that it should skip any further processing of the contents of the connection identified by `id`. In particular, Bro will refrain from reassembling the TCP byte stream and from generating events relating to any analyzers that have been processing the connection. Bro will still generate connection-oriented events such as `connection_finished`. 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`.
- `set_record_packets(id: conn_id, do_record: bool): bool`
  Controls whether packet contents belonging to the connection identified by `id` should be recorded (when `-w` *out.pcap* is provided on the command line). Note that this is independent of whether Bro processes the packets of this connection, which is controlled separately by `skip_further_processing`.
- `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`: Stop recording the connection's content.
  - `CONTENTS_ORIG`: Record the data sent by the connection originator (often the client).
  - `CONTENTS_RESP`: Record the data sent by the connection responder (often the server).
  - `CONTENTS_BOTH`: Record the data sent in both directions. Results in the two directions being intermixed in the file, in the order the data was seen by Bro.

  Returns false if `id` does not point to an active connection and true otherwise. Note that the data recorded to the file reflects the byte stream, not the contents of individual packets. Reordering and duplicates are removed. If any data is missing, the recording stops at the missing data; this can happen, e.g., due to an `ack_above_hole` event.

- `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.
- `skip_http_entity_data(c: connection, is_orig: bool)`
  Skips the data of the HTTP entity in the connection `c`. If `is_orig` is true, the client data is skipped and the server data otherwise.
- `skip_smtp_data(c: connection)`
  Skips SMTP data until the next email in `c`.
- `dump_current_packet(file_name: string): bool`
  Writes the current packet to the file identified by `file_name`. Returns true on success.
- `get_current_packet(): pcap_packet`
  Returns the currently processed PCAP packet, which is a record containing a timestamp, the "snaplen," and the packet data.
- `dump_packet(pkt: pcap_packet, file_name: string): bool`
  Writes the packet `pkt` to the file identified by `file_name`. Returns true on success.
- `set_inactivity_timeout(id: conn_id, t: interval): interval`
  Sets an individual inactivity timeout for the connection identified by `id` (overrides the global inactivity timeout). Returns the previous timeout interval.
- `get_login_state(id: conn_id): count`
  Returns the state of the given login (Telnet or Rlogin) connection identified by `id`. Returns false if the connection is not active or is not tagged as a login analyzer. Otherwise the function returns the state, which can be one of:
  - `LOGIN_STATE_AUTHENTICATE`: The connection is in its initial authentication dialog.
  - `OGIN_STATE_LOGGED_IN`: The analyzer believes the user has successfully authenticated.
  - `LOGIN_STATE_SKIP`: The analyzer has skipped any further processing of the connection.
  - `LOGIN_STATE_CONFUSED`: The analyzer has concluded that it does not correctly know the state of the connection, and/or the username associated with it.
- `set_login_state(id: conn_id, new_state: count): bool`
  Sets the login state of the connection identified by `id` to `new_state`. See `get_login_state` for potential values of `new_state`. Returns false if `id` is not an active connection or does not tagged as login analyzer, and true otherwise.

Generic Programming

- `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`.
- `val_size(v: any): count`
  Returns the number bytes that `v` occupies in memory.
- `clear_table(v: any)`
  Removes all elements from the set or table `v`.
- `resize(v: any, newsize: count): count`. Resizes the vector `v` to the size `newsize`. Returns the old size of `v` and 0 if `v` is not a `vector` type.
- `any_set(v: any): bool`
  Test whether the boolean vector (`vector of bool`) has any true element, i.e., checks whether $\exists x \in \texttt{v} : x = \texttt{T}$.
- `all_set(v: any): bool`
  Test whether all elements of the boolean vector (`vector of bool`) are true, i.e., checks whether $\forall x \in \texttt{v} : x = \texttt{T}$. Missing elements count as false.
- `sort(v: any, ...): any`
  Sorts the vector `v` in place and returns the original vector. The second argument is a comparison function that takes two arguments: if the type of `v` is `vector of T`, then the comparison function is `function(a: T, b: T): bool` that returns `a < b` for some type-specific notion of the less-than operator.
- `order(v: any, ...): vector of count`
  Returns the order of the elements in the vector `v` according to some comparison function. See `sort`.

String Processing

- `byte_len(s: string): count`
  Returns the number of characters (i.e., bytes) in `s`. This includes any embedded NULs, and also a trailing NUL, if any (which is why the function isn't called `strlen`; to remind the user that Bro strings can include NULs).
- `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 the given string with the lowercase letters (as indicated by `isascii` and `islower`) folded to uppercase (via `toupper`).
- `to_upper(s: string): string`
  Returns a copy of `s` with the uppercase letters (as indicated by `isascii` and `isupper`) folded to lowercase (via `tolower`).
- `function edit(arg_s: string, arg_edit_char: string): string`
  Returns a version of s assuming that `edit_char` is the "backspace character" (usually `\x08` for backspace or `\x7f` for DEL). For example, `edit("hello there", "e")` returns `"llo t"`. The argument `edit_char` must be a string of exactly one character, or Bro generates a run-time error and uses the first character in the string.
- `clean(s: string): string`
  Replace non-printable characters in `s` with escaped sequences, with the mappings `NUL` → `\0`, `DEL` → `^?`, values ≤ 26 → `^[A-Z]`, and values not in $[32, 126]$ → `%XX`. If the string does yet have a trailing NUL, one is added.
- `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.
- `find_entropy(data: string): entropy_test_result`
  Performs an entropy test on `data`. The result is a record with the following fields:
  - `entropy`: The information density expressed as a number of bits per character.
  - `chi_square`: The χ^2 test value expressed as an absolute number and a percentage which indicates how frequently a truly random sequence would exceed the value calculated, i.e., the degree to which the sequence tested is suspected of being non-random.
  - `mean`: The arithmetic mean of all the bytes. If the data are close to random, it should be around 127.5. If the percentage is greater than 99% or less than 1%, the sequence is almost certainly not random. If the percentage is between 99% and 95% or between 1% and 5%, the sequence is suspect. Percentages between 90% and 95% and 5% and 10% indicate the sequence is "almost suspect".
  - `monte_carlo_pi`: Each successive sequence of six bytes is used as 24-bit x and y co-ordinates within a square. If the distance of the randomly-generated point is less than the radius of a circle inscribed within the square, the six-

  byte sequence is considered a “hit.” The percentage of hits can be used to calculate the value of π. For very large streams the value will approach the correct value of π if the sequence is close to random.

  - `serial_correlation`: This quantity measures the extent to which each byte in the file depends upon the previous byte. For random sequences this value will be close to zero. Also known as *autocorrelation*.

- `entropy_test_init(index: any): bool`
  Initializes data structures for incremental entropy calculation. The `index` argument is an arbitrary unique value per distinct computation. Returns true on success. See `entropy_test_add` and `entropy_test_finish`.
- `entropy_test_add(index: any, data: string): bool`
  Add `data` to the incremental entropy calculation identified by `index`. Returns true on success.
- `entropy_test_finish(index: any): entropy_test_result`
  Finalizes the incremental entropy calculation identified by `index`. When all data has been added, this function returns the result record which is described above in `find_entropy`.

Math

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

Conversion

- `cat(...): string`
  Returns the concatenation of the string representation of its arguments, which can be of any type. For example, `cat("foo", 3, T)` returns `"foo3T"`.
- `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

  Given no arguments, `fmt` returns an empty string. Given a non-string first argument, `fmt` returns the concatenation of all its arguments, per `cat`. Finally, given the wrong number of additional arguments for the given format specifier, `fmt` generates a run-time error.
- `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(num: count, t: transport_proto): port`
  Creates a `port` with number `num` and transport protocol `t`.
- `to_port(s: string): port`
  Converts a `string` into a `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.
- `ptr_name_to_addr(s: string): addr`
  Converts a reverse pointer name to an address, e.g., `1.0.168.192.in-addr.arpa` to `192.168.0.1`.
- `addr_to_ptr_name(a: addr): string`
  Converts an IP address to a reverse pointer name, e.g., `192.168.0.1` to `1.0.168.192.in-addr.arpa`.
- `parse_dotted_addr(s: string): addr`
  Converts a decimal dotted IP address in a string to an address type.
- `parse_ftp_port(s: string): ftp_port`
  Converts a string representation of the FTP PORT command to an `ftp_port`, e.g., `"10,0,0,1,4,31"` to `[h=10.0.0.1, p=1055/tcp, valid=T]`
- `parse_eftp_port(s: string): ftp_port`
  Same as as `parse_ftp_port`, but instead for EPRT (see RFC 2428) whose format is `EPRT<space><d><net-prt><d><net-addr><d><tcp-port><d>`, where `<d>` is a delimiter in the ASCII range 33-126 (usually `|`).
- `parse_ftp_pasv(s: string): ftp_port`
  Converts the result of the FTP PASV command to an `ftp_port`.
- `parse_ftp_epsv(s: string): ftp_port`
  Same as `parse_ftp_pasv`, but instead for the EPSV (see RFC 2428) whose format is `<text> (<d><d><d><tcp-port><d>)`, where `<d>` is a delimiter in the ASCII range 33-126 (usually `|`).
- `fmt_ftp_port(a: addr, p: port): string`
  Formats the IP address `a` and TCP port `p` as an FTP PORT command, e.g., `10.0.0.1` and `1055/tcp` to `"10,0,0,1,4,31"`.
- `decode_netbios_name(name: string): string`
  Decode a NetBIOS name, e.g., `"FEEIEFCAEOEFFEECEJEPFDCAEOEBENEF"` to `"THE NETBIOS NAME"`.
- `decode_netbios_name_type(name: string): count`
  Converts the NetBIOS name type to the corresponding numeric value.
- `bytestring_to_hexstr(bytestring: string): string`
  Converts a string of bytes into its hexadecimal representation, e.g., `"04"` to `"3034"`.
- `decode_base64(s: string): string`
  Decodes the Base64-encoded string `s`.
- `decode_base64_custom(s: string, a: string): string`
  Decodes the Base64-encoded string `s` with alphabet `a`.
- `uuid_to_string(uuid: string): string`
  Converts a bytes representation of a UUID to its string form, e.g., to `550e8400-e29b-41d4-a716-446655440000`.
- `merge_pattern(p1: pattern, p2: pattern): pattern`
  Merges and compiles the regular expressions `p1` and `p2` at initialization time (e.g., in the event `bro_init()`).
- `convert_for_pattern(s: string): string`
  Escapes `s` so that it is a valid pattern and can be used with the `string_to_pattern`. Concretely, any character from the set `^$-:"\/|*+?.(){}[]` is prefixed with `\`.
- `string_to_pattern(s: string, convert: bool): pattern`
  Converts `s` into a pattern. If `convert` is true, `s` is first passed through the function `convert_for_pattern` to escape special characters of patterns.
- `NFS3::mode2string(mode: count): string`
  Convert UNIX file permissions given by `mode` to a string representation of the form `rw[xsS]rw[xsS]rw[xtT]`.

Network Type Processing

- `mask_addr(a: addr, top_bits_to_keep: count): subnet`
  Returns the address `a` masked down to the number of upper bits indicated by `top_bits_to_keep`, which must be greater than 0 and less than 33. For example, `mask_addr(1.2.3.4, 18)` returns `1.2.0.0`, and `mask_addr(1.2.255.4, 18)` returns `1.2.192.0`.
- `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.

- `connection_exists(id: conn_id): bool`
  Checks whether the connection identified by `id` is (still) active.
- `lookup_connection(id: conn_id): connection`
  Returns the `connection` record for `id`. If `id` does not point to an existing connection, the function returns a run-time error and returns a dummy value.
- `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`.
- `get_orig_seq(id: conn_id): count`
  Returns the highest sequence number sent by a connection's originator, or 0 if `id` does not point to an active TCP connection. Sequence numbers are absolute (i.e., they reflect the values seen directly in packet headers; they are not relative to the beginning of the connection).
- `get_resp_seq(id: conn_id): count`
  Returns the highest sequence number sent by a connection's responder, or 0 if `id` does not point to an active TCP connection.
- `unescape_URI(URI: string): string`
  Unescapes all characters in `URI`, i.e., decodes every `%xx` group.
- `lookup_location(a: addr) : geo_location`
  Performs a geo-lookup of the IP address `a`. Returns city, region, and country. Needs `libgeoip`.
- `lookup_asn(a: addr): count`
  Performs a AS number lookup of the IP address `a`. Needs `libgeoip`.

-
  ```
  x509_verify(der_cert: string, cert_stack: vector of string,
              root_certs: table[string] of string): count
  ```
  Verifies the X.509 certificate in DER format given by `der_cert`. The argument `cert_stack` specifies a certificate chain to validate against, with index 0 typically being the root CA. Bro uses the Mozilla root CA list by default; `root_certs` extends that list with additional root certificates.
- `x509_err2str(err_num: count): string`
  Converts the X.509 certificate verification error code `err_num` into a string representation.

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 associated with `f`.
- `set_buf(f: file, buffered: bool)`
  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.
- `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 succeeds and `f` does not exist already.