Bro 2.1 Cheat Sheet

Version: December 13, 2012
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 addresses, strings, containers
`f(...)` ..... Function call
`expr` `?` *`expr`* `:` *`expr`* ..... Ternary if-then-else

Types

Basic

`addr` IP address (v4: `127.0.0.1`, v6: `[fe80::db15]`)
`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

- `syslog(s: string)`
  Send the string `s` to syslog.
- `system(s: string): int`
  Invokes a command via the `system` function. Returns the return value from the `system()` call. The command is run in the background, `stdout` redirects to `stderr`. Here is a usage example: `system(fmt("rm \"%s\"", str_shell_escape(sniffed_data)));`
- `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)`
  Sets 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(): opaque of md5`
  Retrieves an opaque handle for incremental MD5 hash computation. Add data with via `md5_hash_update`.
- `md5_hash_update(handle: opaque of md5, data: string): bool`
  Feeds an incremental MD5 computation with `data`. Call `md5_hash_finish(handle)` to retrieve the final hash digest.
- `md5_hash_finish(handle: opaque of md5): string`
  Returns the final MD5 digest of `handle`. This invalidates `handle`, i.e., the function can only be called once.
- `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.
- `sha1_hash(...): string`
  Computes the SHA1 hash value of the provided list of arguments. Analogous to `md5_hash`.
- `sha1_hash(...): string`
  Computes the SHA1 hash value of the provided list of arguments.
- `sha1_hash_init(): opaque of sha1`
  Retrieves an opaque handle for incremental SHA1 hash computation. Add data with via `sha1_hash_update`.
- `sha1_hash_update(handle: opaque of sha1, data: string): bool`
  Feeds an incremental SHA1 computation with `data`. Call `sha1_hash_finish(handle)` to retrieve the final hash digest.
- `sha1_hash_finish(handle: opaque of sha1): string`
  Returns the final SHA1 digest of `handle`. This invalidates `handle`, i.e., the function can only be called once.
- `sha256_hash(...): string`
  Computes the SHA256 hash value of the provided list of arguments. Analogous to `md5_hash`.
- `sha256_hash(...): string`
  Computes the SHA256 hash value of the provided list of arguments.
- `sha256_hash_init(): opaque of sha256`
  Retrieves an opaque handle for incremental SHA256 hash computation. Add data with via `sha256_hash_update`.
- `sha256_hash_update(handle: opaque of sha256, data: string): bool`
  Feeds an incremental SHA256 computation with `data`. Call `sha256_hash_finish(handle)` to retrieve the final hash digest.
- `sha256_hash_finish(handle: opaque of sha256): string`
  Returns the final SHA256 digest of `handle`. This invalidates `handle`, i.e., the function can only be called once.
- `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 shuts down Bro by terminating outstanding processing. Returns true after successful termination and false when Bro is still in the process of shutting down.
- `exit(code: int)` Shuts down the Bro process immediately and returns with `code`.

Introspection

- `bro_version(): string`
  Returns the Bro version string.
- `getpid(): count`
  Returns Bro's process ID.
- `gethostname(): string`
  Returns 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.
- `is_local_interface(ip: addr): bool`
  Returns true if the address `ip` is a valid DNS entry for `localhost`.

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.
- `file_size(f: string): double`
  Returns the file size in bytes of the file identified by `f`.
- `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 false if the creation fails or if `f` exists already.
- `enable_raw_output(f: file)`
  Function equivalent to the `&raw_output` attribute, which prevents escaping of non-ASCII characters when writing to `f`.

Generic Programming

- `length(v: any): count`
  Returns the number of elements in the container `v`.
- `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`
  Tests 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`
  Tests whether all elements of the boolean vector (`vector of bool`) are true, i.e., checks whether $\forall x \in \mathtt{v} : x = \mathtt{T}$. 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 must be `function(a: T, b: T): bool`, which 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`.

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) 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`
  Extracts a substring of `s`, starting at position `start` and having length `n`.
- `split(s: string, re: pattern): table[count] of string`
  Splits `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`.
- `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.
- `string_cat(...): string`
  Concatenes a variable number of string arguments into a single string.
- `cat_string_array(a: table[count] of string): string`
  Same as `string_cat`, except that it takes an array of strings as argument and concatenates its values into a single string.
- `cat_string_array_n(a: table[count] of string, start: count, end: count): string`
  Same as `cat_string_array`, but only concatenates the strings from index `start` to `end`.
- `join_string_array(sep: string, a: table[count] of string): string`
  Concatenates all elements in `a` into a single string, with `sep` placed between each element.
- `join_string_vec(v: vector of string, sep: string): string`
  Concatenates all elements in `v` into a single string, with `sep` placed between each element.
- `sort_string_array(a: table[count] of string): string`
  Sorts the string array `a` and returns a sorted copy.
- `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 compares `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`
  Locates 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`
  Substitutes 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 uppercase letters (as indicated by `isascii` and `isupper`) folded to lowercase (via `tolower`).

- `to_upper(s: string): string`
  Returns a copy of `s` with the lowercase letters (as indicated by `isascii` and `islower`) folded to lowercase (via `toupper`).
- `is_ascii(s: string): bool`
  Returns false if any byte value of `s` is greater than 127, and true otherwise.
- `edit(s: string, 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`
  Replaces non-printable characters in `s` with escaped sequences, with the mappings `NUL` $\rightarrow$ `\0`, `DEL` $\rightarrow$ `^?`, values $\leq$ 26 $\rightarrow$ `^[A-Z]`, and values not in $[32, 126] \rightarrow$ `%XX`. If the string does not yet have a trailing NUL, one is added.
- `to_string_literal(s: string): string`
  Same as clean, but with different mappings: values not in $[32, 126] \rightarrow$ `%XX`, `\` $\rightarrow$ `\\`, `'` $\rightarrow$ `\'`, `"` $\rightarrow$ `\"`.
- `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.
- `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`.
- `entropy_test_init(): opaque of entropy`
  Retrieves a data structures for incremental entropy calculation. Returns true on success. See `entropy_test_add` and `entropy_test_finish`.
- `entropy_test_add(handle: opaque of entropy, data: string): bool`
  Adds `data` to the incremental entropy calculation identified by `handle`. Returns true on success.
- `entropy_test_finish(handle: opaque of entropy): entropy_test_result`
  Finalizes the incremental entropy calculation identified by `handle`.

Network Type Processing

- `is_v4_addr(a: addr): bool`
  Checks whether an address is IPv4. Returns `true` for IPv4 and `false` for IPv6 addresses.
- `is_v6_addr(a: addr): bool`
  Checks whether an address is IPv6. Returns the opposite of `is_v4_addr`.
- `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 merges 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 generates a run-time error and returns a dummy value.
- `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 country, region, city, latitude, and longitude. Needs Bro to built with `libgeoip`.

- `lookup_asn(a: addr): count`
  Performs an AS lookup of the IP address `a`. Needs `libgeoip`.
-

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 à la `printf`. 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.
- `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`.
- `to_double(s: string): double`
  Converts a `string` into a `double`.
- `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`.
- `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`.
- `addr_to_counts(a: addr): vector of count`
  Converts an IP address into a vector of of `count`s in host byte-order. Returns 4 elements for IPv6 and one for IPv4 addresses.
- `counts_to_addr(v: vector of count): addr`
  The dual to `addr_to_counts`: converts a vector of counts to and IP address.
- `to_subnet(ip: string): subnet`
  Converts a `string` into a subnet type. Returns `../0` if the input does not parse correctly.
- `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`. Concretly, 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.