Azinix, pre-alpha

Adnan Aziz

Abstract

Azinix is Linux-based software for monitoring and controlling networks. It receives packets on the host's Ethernet interfaces, and applies user-specified rules to these packets. Each rule consists of a condition and an action. The condition may refer to the packet's header and payload, as well as network state. Azinix natively implements control, forwarding, and queuing; additionally, diverse actions such as encryption, compression, logging, and statistics gathering can be invoked through dynamically linked scripts and object code.

The key advantages of Azinix include

- high performance: Azinix supports Gigabit speeds on cheap commodity hardware
- scalability: Axinix suffers no performance degradation as the number of rules and the size of the network increases
- flexibility: Azinix can implement arbitrary actions, using dynamically linked compiled code as well as scripts
- content inspection: Azinix rules can refer to the packet header as well as payload
- ease-of-use: Azinix uses a Tcl-based user-interface, and include sample scripts, and code for crafting test cases

Azinix consists of 25,000 lines of custom C code. It is designed to be extensible, and is extensively documented using `doxygen`.

Contents

1 Overview

Azinix is Linux-based software for monitoring and controlling networks. A common used model is for Azinix to connect an internal network and the Internet, as shown in Figure 1.

Azinix receives packets on the host's Ethernet interfaces, and applies user-specified rules to these packets. Each rule consists of a condition and an action. The condition may refer to the packet's header and payload, as well as network state.

Rules can be broadly classified as follows:

Access control and forwarding : Azinix has the ability to forward packets based on conditions on the packet header and payload. Azinix implements basic firewalling, in which conditions refer to packet header fields, as well as advanced access control, wherein conditions can refer to packet content.

Queuing : Azinix implements schemes to partition and prioritize bandwidth based on the packet header, payload, and flow, and network state. By flow, we refer to an established TCP or UDP connection, defined by source and destination addresses and ports; network state refers to statistics on average bandwidth, number of flows, etc.

Monitoring : Azinix can be used to monitor an interface, in a manner similar to `tcpdump` and `Snort`. In addition to printing information about individual packets, Axinix can report network statistics as they evolve.

In addition, Axinix can dynamically load Tcl scripts and object code to perform a diverse set of actions. These include the ability perform logging (entire packets, or just their destination IP addresses, track flows, rates, and send email alerts in specific situations); encrypting and decrypting packets based on user provided keys; checks for and responses to abnormal traffic, such as repeated login attempts; and applying `lzo`—compression/decompression on packets.

Benefits of Azinix include:

1. performance—Azinix can implement a rule-set with 2000 rules on at Gigabit Ethernet speeds on commodity hardware costing under 1000$.

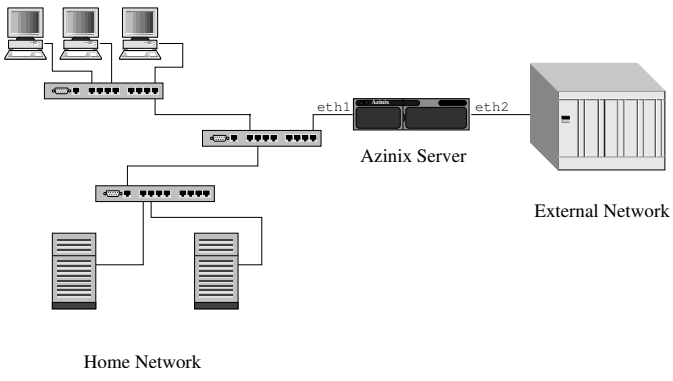

Figure 1: Azinix deployment.

2. scalability—Azinix can be used with rules defined on 100,000 prefixes with no loss of performance.

3. content-inspection—Azinix provides the ability to perform substring matching and regular expression search on packet payload.

4. ease-of-use—Azinix uses a `Tcl`-based user interface and includes model rules and action scripts.

5. extendibility—Azinix offers users the ability to define custom actions; these actions could be `Tcl` scripts or dynamically linked `C` code.

Azinix is only as good as the rule-set it is given to implement. It is easy to write rules that have unintended consequences, such as blocking all packets. The distribution includes a number of example policies that can serve as reasonable defaults.

1.1 Getting started

Azinix is distributed as tarball. It should build and run on Linux 2.4 onward. It requires a number of libraries; the exact library versions as well as sources are available from the Azinix website.

The Azinix user-interface is based on `Tcl`. The most common use model is for the user to define parameters such as interfaces, parameters (such as buffer space to allocate), queuing structures, rules, and custom actions, in a `Tcl` script. This script can then be sourced from Azinix on startup. Azinix exports custom functions to the shell for creating rule managers, packets, testing, as well as the core routine implementing the rules. The `azinix-sample.tcl` file, distributed with Axinix, includes examples of the above; you are strongly encouraged to read through it, paying particular attention to the comments.

Ousterhout's original text [3] on `Tcl` is still an excellent resource for learning `Tcl`; Flynt's more recent book [2] covers newer features as well as programming with `Tcl`.

1.2 Overview of rules

Rules, as well as other structures such as queues, are specified using a set of fields; each field consists of an attribute and a value.

A rule formalizes a policy such as "packets to the HTTP server containing the string `.cgi` are to be dropped". A rule consists of a *condition* and an *action* to be taken if that condition holds. If multiple rules are applicable to a packet, all corresponding actions are applied. If no rules are applicable, the packet is dropped.

Rules are applied in the order in which they appear in the file. In particular, if two rules conflict, e.g., one says to forward the packet to interface `eth1`, and the other to interface `eth2`, then the packet will be forwarded to the interface that corresponds to the rule that appears later in the sequence of applicable rules.

Poorly thought-out rulesets can lead to packets being set to the wrong interface (errors in forwarding table), dropped (access control mis-configuration),

reordered (two packets from the same flow being assigned different classes of service), and corrupted (encrypted with the wrong key).

Azinix has the ability to generate packets. This feature can be very helpful when testing and debugging rulesets, and measuring system performance. See the `azinix-test` function in `azinix-sample.tcl` for usage and examples.

1.3 Manual organization

The remainder of the manual is structured as follows: Azinix usage is illustrated in Section 2; this is the most important section for most users. Conditions are covered in Section 4.

2 Using Azinix

We now illustrate the kinds of policies that Azinix can implement, and the corresponding rules. For the sake of these examples, assume that Azinix is running on a machine that has two Ethernet interfaces—one connected to the home network, the other connected to the external network, as shown in Figure 1. Let the interfaces be `eth1` and `eth2`, respectively.

Rules are specified in text. The `Tcl` script is specified as a text file, organized as lines. Lines starting with a `#` are comments; blank lines are ignored. The `include` and `var` constructs are used as preprocessor directives to include files, and define symbols for textual substitution. A backslash (\) can be used to split rules across lines.

The Azinix preprocessor allows for a file to include other files through the `include` directive, e.g., `include firewall_rules.txt`. Commonly used strings can be defined using macros. For example, it is convenient to use macros to define sets of IP addresses and port ranges; they are more concise as well document the rules better. Macros are specified using the `var` construct:

```
# Home network IP addresses
var HN 4.22.41.17,163.12.241.11/30
# Good IP addresses
var GN HN,212.145.12.0/255.255.0.0
# Port the http server listens on
var HTTP_PORT 80
```

2.1 Forwarding and access control

2.1.1 Forwarding

Azinix can be used to forward packets. The following rule bridges the home and external networks in Figure 1, i.e., it sends all Ethernet packets originating on the home network to the external network, and vice versa.

```
generic ( interface:eth1; ) route:eth2;
generic ( interface:eth2; ) route:eth1;
```

We can add firewalling capability, e.g., by blocking all packets from the external network to the home network that do not originate from the set of IP addresses in `212.145.44.*`:

```
generic ( interface:eth1; ) route:eth2;
generic ( interface:eth2; ) route:eth1;
ip !212.145.44.* any -> any any ( interface:eth2; ) drop
```

We can print all the payload for all TCP packets originating within the home network destined to port 80 that have the text `HTTP GET`, independent of case:

```
generic ( interface:eth1; ) route:eth2;
generic ( interface:eth2; ) route:eth1;
ip !212.145.44.* any -> any any ( interface:eth2; ) drop
tcp any any -> any 80 ( interface:eth1; \
content:"HTTP GET"; nocase; ) tcl-ext:printPayload;
```

Here `printPayload` is a predefined `Tcl` function which prints the payload of a TCP packet.

Used in conjunction with a managed Ethernet switch, Azinix can function as a router with a high-port density.

2.1.2 Access control

Conventional firewall Let's say we want to drop all packets, which are not from the good network to the http server. We express this with the following rule:

```
tcp !GN any -> HN HTTP_PORT () drop
```

Content firewalling We can also write rules on packet content. Let's say we want to drop all HTTP requests containing the string `cgi-bin` that are not from the home network.

```
tcp !HN any -> HN HTTP_PORT ( content:"cgi-bin"; ) drop
```

Note that the TCP connection may already exist before the condition is satisfied, and without specifying that the connection has to be reset, the HTTP server will be left with a hanging connection.

2.2 Queuing

Each `qos` declaration defines a new queue [4]. A queuing rule is one where the corresponding action is to insert the packet in a queue. If multiple queuing rules are applicable to a packet, it will be inserted into these queues in the order in which the corresponding rules appear.

Although queuing rules focus on allocation of network resources, these rules can be used to allocate server resources too—consider, for example, the rule restricting the number of connections to an HTTP server.

First we describe how Azinix can be used to prioritize packets using a class-of-service queuing system. Our convention is that class-0 is the highest priority queue; a packet in a class-k queue is processed only if there are no packets in a class-$(k+1)$ or higher queue.

We define a queue as follows:

```
queue q0 type:cos; numclasses:2;  \
  maxbytes:10000; maxrate:1000; drop:red;
```

The `maxbytes` field is used to set the maximum number of bytes the queue is allowed to hold. The `maxrate` field is the maximum rate, in bytes-per-second, that the queue is allowed to process packets through. Azinix can use tail-drop (wherein packets are dropped when the queue is full) or random-early-discard (wherein packets are discarded with some probability as the queue starts to get full). Azinix uses a simple window-based scheme for computing the rate. Specifically, the rate is 0 if no packets have arrived in that queue in the past T_{win} seconds (where T_{win} is the user-settable window size, with a default value is 10 seconds); otherwise it is the number of bytes that have exited in the past T_{win} seconds, divided by T_{win}.

Now we define two queuing rules used for prioritizing packets. Suppose we want the following assignment of packets to queues: if a packet has source and destination IP addresses within the home network, is destined to the SQL server port, and contains the string `mySQL GET` we assign its class to 0 in queue `q0`, otherwise its class is 1. The rules that implement this are expressed as follows:

```
# every packet from HN to HN SQL_PORT is class 1
ip HN any -> HN SQL_PORT ( ) queue:q0; class:1;

# now set those packets containing mySQL GET to class 0
ip HN any -> HN SQL_PORT ( content:"mySQL GET"; ) \
  queue:q0; class:0;
```

Note that one packet of a flow may be in a given class, and another may be in a different class, or not even in the queuing system, leading to packets being reordered within a flow.

2.2.1 Partitioning

Azinix can also be used to partition network bandwidth. The following defines a queuing system in which packets are assigned to different queues based on their source IP address; the queues are served in round-robin order.

```
queue q1 type:drr; maxbytes:10000; drop:red; flow:srcip;
```

The `flow:srcip;` construct is used to specify that all packets from the same source ip are assigned to the same queue.

The following rule ensures all hosts within the home network get equal bandwidth to the external network:

```
ip HN any -> EN any () queue:q1;
```

The following rule ensures that all TCP/IP flows from the external network to the HTTP server get equal shares of the network bandwidth:

```
queue q2 type:drr; maxbytes:10000; drop:red; flow:srcdesttcpip;
ip EN any -> HS any () queue:q1;
```

3 Extensions

Azinix provides users the ability to customize rule actions. These customizations can be defined by `Tcl` scripts, or by `C` code. Extensions can be used to implement checks beyond the standard checks available in Azinix, as well as actions that are not available in Azinix.

3.1 Preprocessing

Given a packet, Azinix computes the set of applicable rules, and then performs the corresponding actions in sequence, stopping if the packet is dropped (either in a queuing rule, or because of an explicit drop). In particular the packet is unchanged.

There are situations where rules do change the packet; examples include rules which encrypt/decrypt the packet, as well as rules which compress/decompress packets.

Consider the case where a packet is decrypted. In this case, the rule set for the packet has to be computed based on the decrypted packet. For this purpose, Azinix rules can be qualified with the `preprocess` directive; all rules labelled with the preprocess directive are applied first, in the sequence in which they appear.

3.1.1 Tcl extensions

The following example illustrates the use of extensions. The `uscript_1` procedure takes a packet, extracts its source IP address, and stores that in a hash.

```
set allIps("dummy") 1
proc uscript_1 { aPkt } {
  global allIps
  set srcAddr "[pktReadSrcIp $aPkt]"
  set count  [array size allIps]
  set allIps($srcAddr) $count
}

tcp any any -> any any \
  ( content:"http get"; nocase; ) \
  tcl-ext:uscript_1;
```

As it stands the procedure is not very interesting, since it does nothing with the hash, but in a more realistic setting the procedure would perform logging with the database.

Azinix comes with procedures for logging data, generating alerts, etc. These are described in the file `azinix-sample.tcl`.

For example, the following rule results in any external IP address initiating more than 100 flows in 60 seconds being blocked from initiating any more flows for 10 minutes.

```
ip EN any -> any any (mesg:"flow limit";) \
 tcl-ext:limit; period:60; flows:100; \
 action:block; duration:1200;
```

3.1.2 C code

In some cases it may be preferable to write C code rather than `Tcl` to implement the action. The most natural reason is performance. In addition it may easier to write, test, and debug C functions, because of familiarity as well as the sophisticated libraries and tools that exist for C code compared to `Tcl`.

Azinix provides a mechanism for letting the user write his own C code which is then called as the action.

```
tcp any any -> any any \
  ( content:"http get"; nocase; ) \
  tcl-ext:fastLogReq;
```

The C function called `fastLogReq` is invoked for packets which satisfy the rule conditions.

The C code is dynamically linked in as follows:

```
register_c-ext $mgr fastLogReq $pathToUcode/libucode.so
```

Here `$mgr` is a data structure which encapsulates the ruleset.

Arguments can be optionally passed in to the call:

```
tcp any any -> any any \
  ( content:"http get"; nocase; ) \
  c-ext:fastLogReq:"name:123";
```

The function can be called in initialize mode:

```
register_c-ext $mgr fastLogReq $argument
```

This is used for initializing any persistent data the function needs to keep around. The argument variable is used to pass in values. The call to `register_c-ext` can be performed multiple times. This may be useful for example when the function `fastLogReq` may be passed different arguments, and each argument needs some preprocessing.

The user-defined function must have the following prototype:

```
int fastLogReq( Evl_Manager_t *, Pkt_ProcessPkt_t *,
void **, bool initFlag, char *argument );
```

The arguments are the manager, a pointer to a `Pkt_ProcessPkt_t`, which is an encapsulation of the packet, and a pointer to a `void *`, which can be used to manage persistent state. On the first call to the function this points to `NULL`, unless it has otherwise been set through `initialize_c-ext`. The function should return 1 if the packet is to be dropped, and 0 otherwise.

Azinix exports a large number of useful functions, e.g., code to insert the packet into a particular queue, code to mangle the packet header, etc.

3.2 Encryption

As examples of dynamically loaded code, Azinix can be used to encrypt and decrypt packets using the AES cryptographic system [1]. The following rule encrypts all `tcp` packets from port `80` whose destination is `192.168.2.0/24`.

```
set aKey "01a21bc00128dea1"
register_c-ext $mgr encrypt $partition/libucode.so
ip any 80 -> 192.168.2.0/24 any () c-ext:encrypt:$aKey
```

In my experience, the AES code in Azinix can perform AES encryption at over 1 Gbps, on a 3.6 GHz C2D, with the -O3 flags turned on in the compiler.

3.3 Compression

The following rule uses `lzo` to compress all tcp packets from web servers:

```
register_c-ext $mgr compress $past/libucode.so
tcp HS any -> any any () c-ext:compress;
```

Compression is applied to the IP packet's payload, i.e., the header bytes are not compressed. Since `lzo` can actually lead to a larger payload, we set the IP protocol field to 8, to indicate that the packet is compressed.

In my experience, the `lzo` library used by Azinix can perform compression at over 1 Gbps, on a 3.6 GHz C2D, with the -O3 flags turned on in the compiler. Compression is performed on a packet by packet basis. Compressing packetized data as opposed to the data in its entirety leads to suboptimal reduction. As a reference point, applying `lzo` to 1500 byte chunks of the source of this manual resulted in 14203 bytes after compression; the original was 31500 bytes. Applying `lzo` to the entire file lead resulted in 11410 bytes. The `gzip` program created a file containing 11356 bytes. See the `compTestPerf` and `compTestReduction` commands to get an idea of how much reduction you can achieve, and how fast the compression is.

4 Conditions

In this section we elaborate on the condition component of a rule.

Each condition consists of a required condition and optional conditions. Some of the sub-conditions may be specific to the type of rule.

4.1 Required condition

The required condition is specified as follows:

```
prot srcip srcport -> destip destport
```

Here `prot` is one of `any`, `icmp`, `ip`, `tcp`, and `udp`. Note that `tcp` is a tighter condition on the `prot` field than `ip`.

The keyword `any` can be used to indicate that all addresses/ports are included.

The `srcip` and `destip` fields can be a single IP address in dotted form, an IP address and a mask, or a set of IP addresses with or without masks.

The `srcport` and `destport` fields can be a single port (which must be in the set $\{0, \ldots, 65535\}$), a range (specified as `A:B`), or a set of ports and ranges.

For both IP and port fields, an optional preceding `!` indicates the complement of the set of addresses. Square brackets are recommended for identifying groupings.

Keyword	Argument	Example
IP Header Checks		
`sameip`	none	`sameip;`
`proto`	0–255	`proto:6;`
`type`	0-15	`type:4;`
`ttl`	0–255	`ttl:>255;`
`id`	0-255	`id:66;`
`size`	0–65535	`size:64;`
`fragbits`	`DF,MF,RB`	`fragbits:MF`
`options`	`rr,eol,nop,ts,sec, lsrr,ssrr,satid`	`options:RR`
ICMP header checks		
`itype`	0–255	itype:111;
`icmp_code`	0-255	icmp_id:51201;
`icmp_seq`	0-65535	icmp_seq:0;
TCP header checks		
`seq`	0–$2^{32}-1$	seq:32;
`ack`	0–$2^{32}-1$	ack:0;
`flags`	`F,S,R,P,A,U,2,1,0`	flags:U+;

Table 1: Header checks

Semantically, a packet satisfies a required condition if and only if its header protocol, source, and destination IP address and port numbers all match those specified in the condition.

4.2 Optional conditions

Optional conditions are specified in a parens-enclosed list, e.g.,

```
tcp HN 880 -> !HN 1000  \
  (sameip; content:"good.cookie"; ttl:225;) \
```

All but one of the optional conditions take an argument, which is specified after the colon. There are two kinds of optional conditions: those on the packet header, and those on the packet content.

Header checks, described in Table 1, are checks on the protocol fields. The `sameip` tests to see if the packet source IP address equals its destination IP address, which is a sign of a bad packet used sometimes in denial-of-service attacks. (This is the only test which does not take an argument.) The remaining checks focus on the values in the different header fields.

Recall that a TCP packet consists of a fixed set of header bytes, followed by content bytes. We will refer to the content bytes as a string.

The basic content check is for given substrings using the `content` construct. For example, the following check tests to see if the substring `http get` appears in the content: `content:"http get";`. The content check can be applied to both TCP and UDP packets.

Multiple content checks can be applied to check if a collection of substrings appears in the packet's content. For example, to check if two strings,

say `http get` and `.cgi` both appear, simply use `content:"http get"; content:".cgi";`.

One can also check if a given substring does not appear. The check to see if the string `access denied` does not appear is `content:!"access denied";`. Note that the check `content:!"http get"; content:!".cgi";` holds exactly when both strings `http get` and `.cgi` do not appear.

In some instances we may want to check if a given substring appears independent of the case of the alphabetical characters. For example, suppose we want to check if the string `credit` appears in a packet independent of case. In this case, packets with content including `credit`, `CREDIT`, `CREdit`, etc., should all test positive. We specify this by qualifying the content test with the `nocase` construct: `content:"credit"; nocase;`.

4.2.1 Range checking

There are situations where we may want to test if a given substring appears in the first n bytes of the content, where n is a user specified constant. We do this by qualifying the content test with the `depth` construct. For example to test if the string `bc` appears within the first 10 bytes of the content, we write `content:"bc"; depth:10;`. A packet with content starting with `goodxyzabc` satisfies this check; a packet with content starting with `bgoodxyzabc` does not satisfy this check, i.e., the substring must appear *entirely* within the specified depth.

Similarly, we may want to test if a given substring appears only after the first n bytes of the content. We do this by qualifying the content test with the `offset` construct. For example, to test if the string `odx` appears after the first 8 bytes of content, we write `content:"odx"; offset:8;`. A packet with content starting with `goodxyzabc` does not satisfy this check; however a packet with content starting with `12345678odxyzabc` does satisfy the check.

We may also want to check if a given substring appears within the n bytes following another specified substring. We do this by qualifying the content check with the `within` construct. For example, to test if the string `abc` appears within the first 6 bytes following string `odx`, we write `content:"odx"; content:"abc"; within:6;`. A packet with content `goodx123abc` satisfies this check; however, a packet with content `goodxy1234abc` fails this check, i.e., the the substring must appear *entirely* within the specified number of bytes starting immediately after the last byte of the previous matching substring.

Similarly, we may want to check if a given substring appears only after the n bytes following the previously matched substring. We do this by qualifying the content test with the `distance` construct. For example, to test if the string `abc` appears 6 bytes or later following the string `odx`, we write `content:"odx"; content:"abc"; distance:6;`. A packet with content `goodx12345abc` fails this check, but a packet with content `goodx123456abc` passes it.

The `depth`, `offset`, `distance`, and `within` constructs can be used in conjunction with each other, as in the following check:

```
content:"abc"; depth:6; offset:1; \
```

```
content:"foo"; nocase; distance:4; within:7;
```

This condition is satisfied only when the content contains `abc`, starting at the second or third bytes of the content, with `foo` appearing, independent of case, exactly 3 bytes after `abc` ends. For example, the packet with payload `0abc1234fo0` satisfies this check.

Negated content checks can similarly, be qualified:

```
content:!"abc"; offset:1;
```

This condition is for example by the content `abcd`, but not by `xabcd`.

When checking a condition of the form substring A followed by B followed by C, with range constraints, Azinix employs a greedy strategy. In particular, each time A occurs, it checks for the first B following A within the specified range, and tests for C relative to this B; later occurrences of B in the range are ignored. Consequently, the test fails for the payload `foobarbarxyzabc`:

```
content:"foo"; content:"bar"; within:6; \
  content:"abc"; within 6;
```

It does hold for the payload `foo123barxyzabc`. If you need a more complex check, use the `pcre` construct described below.

4.2.2 Regular expression search

Azinix also offers the ability to perform arbitrary regular-expression search checks within the content using the `pcre` construct. For example, to test if the content contains the substring `ab` followed at some later point by the substring `cd`, we write `pcre:"ab.*cd"`.

Note that regular expression search is relatively expensive from a computational perspective. In some cases it can be avoided: for the example above, the check `content:"ab"; content:"cd"; within:100000;` is equivalent (assuming contents are not longer than 100000 bytes). In other cases, content checks can be used to avoid performing the check in most cases. For example, if we want to check whether the content matches the regular expression `[g]+[o]+.*abcd`, we can add the additional content check `content:"abcd";`, since the substring `abcd` must appear for the regular expression to match. The test for the presence of `abcd` is very fast, and if in most cases the regular expression check fails because of the absence of `abcd`, the expensive calls to the reg-exp library can be avoided.

4.2.3 Byte jumps and tests

In some situations, it may be desirable to inspect the content in specific places, and perform checks on the numeric values encoded by the bytes therein. Similarly, these numeric values may be used to specify further locations to search the content in. These checks are implemented using the `byte_check` and `byte_jump` constructs.

The general format of the `byte_jump` construct is as follows:

```
byte_jump: <bytes_to_convert>, <offset>
           [, [relative],[big],[little],[string],
              [hex],[dec],[oct],[align]]
```

For example, the following check looks to see if 9 appears n bytes after 0, with n being the integer equivalent of the number encoded by the 4 bytes appearing 20 bytes after the occurrence of 0, with alignment to a 4-byte address.

```
content:"0"; byte_jump:4,20:relative,align; content:"9";
```

The qualifiers `relative, big, little, string, hex, dec, oct, align` refer to whether the test is made relative to the last match or not, whether to use big- (default) or little-endian representation, whether the number is encoded in ascii, hex, decimal, or oct, and whether the test should be made after alignment to a 4-word boundary.

The general format of the `byte_test` construct is as follows:

```
byte_test: <bytes_to_convert>, <operator>,
           <value>, <offset>
           [, [relative],[big],[little],[string],
              [hex],[dec],[oct] ]
```

For example, the following check holds just in case the value encoded by the two bytes in little endian format immediately following the previous match is greater than 1024.

```
byte_test:2,>,1024,0,relative,little
```

The qualifiers `relative, big, little, string, hex, dec, oct` are as before.

4.2.4 Forming strings

In previous examples, we have worked with strings over bytes that are printable. We can specify arbitrary strings using hexadecimal encoding—each byte is represented by a pair of hexadecimal digits. Specifically, we use the `|` symbol to escape to hex-mode. In this fashion, the content checks `content:"JKL";`, `content:"J|4b|K";`, and `content:"|4A 4 B 4c|";` are all equivalent. The characters `|`, and `"` are special, and have to be escaped using a backslash (\); the backslash when not used for escaping must itself be escaped, i.e., written as `\\`.

The `evl.test` script included with the Azinix source code contains a large number of tests for checking the correctness of the Azinix rule matching engine. It is a good place to see examples of rules; it can also be used to try out rules on packets, to ensure that a rule works as intended.

The construct `msg` can be used to embed a comment in a rule; specifically, adding `msg:"Malformed packet:  has source ip = destination ip";` to the set of optional checks simply associates the message with the rule.

5 Software architecture

Azinix runs in user-space. The Azinix code base consists of 30000 lines of C, including test and debug code. It makes extensive use of existing data structures networking libraries. The Azinix has made use of best practices for software engineering—a consistent naming convention, `doxygen` for documentation, coverage-driven unit testing, regression suites, memory and

performance checking. The code exports a number of packet and data structure manipulation routines. See the `doxygen` page for details on how the code is organized.

References

[1] The advanced encryption standard. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[2] C. Flynt. *Tcl/Tk: A Developers Guide, Second Edition*. Morgan-Kaufmann, 2003.

[3] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional, 1994.

[4] L. Peterson and B. Davie. *Computer Networks*. Morgan-Kaufmann, 2000.