Writing Application Protocol Parsers

Jeffrey Kirby

Overview

- Introduction
- Motivation
- Related Work
- Assumptions
- binpac Language
- Evaluation
- Future Work

Introduction

- binpac is a declarative programming language and compiler used to write application protocol parsers

Motivation

- Why do we need application protocol parsers?
  - Network Intrusion Detection Systems (NIDS)
  - Network monitors
  - Smart firewalls
  - Application layer proxies

Motivation

- Why do we need binpac?
- Difficulties of writing parsers by hand
    - Tedious and error prone
    - Protocols are complex
    - Need to think about corner, or rare, cases
    - Hacker purposefully injects non-conforming data
    - Need to handle thousands of connections in real-time
- Vulnerabilities have been discovered in existing protocol parsers

Motivation

- Reusability
  - Protocol parsers used in one application cannot be easily used in another application
- Lack of abstraction

Motivation

- Protocol parsers differ from language parsers
  - Network protocols are not easily expressed as a Context Free Grammar
  - Need for correlation across different directions of a single connection
  - Language processors are not designed to concurrently parse multiple, incomplete input streams

Related Work

- Augmented BNF (ABNF)
  - Concise, but incomplete, description of a protocol
- Generic Application-level Protocol Analyzer (GAPA)
  - Protocol analyzer used for traffic analysis at end host machines
- PACKETTYPES
  - Language which treats network packet data structures as C types

Related Work

- binpac, on the other hand
    - Designed to process high-volume traffic at network gateways
    - Abstraction
    - Modularity

Assumptions

- binpac focuses only on application protocol parsing and assumes existence of lower level protocol analyzers

binpac Language

- Declarative language
  - Describes what computation should be performed but not how to compute it
  - Not Imperative
  - e.g. Functional, Logic

binpac Language

- Features
  - Elementary types
    - Similar to C++ integer and string types
  - Composite types
    - record, array, case
  - Type parameters
    - Allow for passing information between types
    - Avoids need to keep external state

binpac Language

- Features
  - Derivative fields
    - Useful for intermediate computation results
  - Byte order (Big-endian v. Little-endian)
    - User may specify which field to use for byte order
  - State management
    - *flow* - sequence of messages
    - *connection* – pair of flows

binpac Language

- Features
  - Integrating custom computation
    - C/C++ code may be embedded
  - Error detection / recovery
    - Can't just “stop and complain” like a language parser
    - Upon error, throws C++ run-time exception
  - Separation of concerns
    - “breaking a program into distinct features that overlap in functionality as little as possible”

Language Construct	Brief Explanation
`%header{ ... %}`	Copy the C++ code to the generated header file
`%code{ ... %}`	Copy C++ code to the generated source file
`%member{ ... %}`	C++ declarations of private class members of connection or flow
`analyzer ... withcontext`	Declare the beginning of a parser module and the members of `$context`
`connection`	Define a connection object
`upflow/downflow`	Declare flow names for two flows of the connection
`flow`	Define a flow object
`datagram = ... withcontext`	Declare the datagram flow unit type
`flowunit = ... withcontext`	Declare the byte-stream flow unit type
`enum`	Define a "enum" type
`type ... =`	Define a `binpac` type
`record`	Record type
`case ... of`	Case type—representing an alternation among case field types
`default`	The default case
⟨`type`⟩`[]`	Array type
`RE/.../`	A string matching the given regular expression
`bytestring`	An arbitrary-content byte string
`extern type`	Declare an external type
`function`	Define a function
`refine typeattr`	Add a type attribute to the `binpac` type
⟨`type`⟩ `withinput` ⟨`input`⟩	Parse ⟨type⟩ on the given ⟨input⟩ instead of the default input
`&byteorder`	Define the byte order of the type and all enclosed types (unless otherwise specified)
`&check`	Check a predicate condition and raise an exception if the condition evaluates to false
`&chunked`	Do not buffer contents of the bytestring, instead, deliver each chunk as `$chunk` to `&processchunk` (if any is specified)
`&exportsourcedata`	Makes the source data for the type visible through a member variable `sourcedata`
`&if`	Evaluate a field only if the condition is true
`&length = ...`	Length of source data should be ...
`&let`	Define derivative types
`&oneline`	Length of source data is one line
`&processchunk`	Computation for each `$chunk` of bytestring defined with `&chunked`
`&requires`	Introduce artificial data dependency
`&restofdata`	Length of source data is till the end of input
`&transient`	Do not create a copy of the bytestring
`&until`	End of an array if condition (on `$element` or `$input`) is satisfied

```
analyzer HTTP withcontext {  # members of $context
    connection: HTTP_Conn;
    flow:       HTTP_Flow;
};
enum DeliveryMode {
    UNKNOWN_DELIVERY_MODE,
    CONTENT_LENGTH,
    CHUNKED,
};
# Regular expression patterns
type HTTP_TOKEN = RE/[^()<>@,;:\\"\/\[\]?={} \t]+/;
type HTTP_WS    = RE/[ \t]*/;
extern type BroConn;
extern type HTTP_HeaderInfo;
%header{
    // Between %.*{ and %} is embedded C++ header/code
    class HTTP_HeaderInfo {
    public:
        HTTP_HeaderInfo(HTTP_Headers *headers) {
          delivery_mode = UNKNOWN_DELIVERY_MODE;
          for ( int i = 0; i < headers->length(); ++i ) {
            HTTP_Header *h = (*headers)[i];
            if ( h->name() == "CONTENT-LENGTH" ) {
              delivery_mode = CONTENT_LENGTH;
              content_length = to_int(h->value());
            } else if ( h->name() == "TRANSFER-ENCODING"
                    && has_prefix(h->value(), "CHUNKED") ) {
              delivery_mode = CHUNKED;
            }
          }
        }
        DeliveryMode delivery_mode;
        int content_length;
    };
%}
# Connection and flow
connection HTTP_Conn(bro_conn: BroConn) {
    upflow = HTTP_Flow(true);  downflow = HTTP_Flow(false);
};
flow HTTP_Flow(is_orig: bool) {
    flowunit = HTTP_PDU(is_orig)
                     withcontext(connection, this);
};
# Types
type HTTP_PDU(is_orig: bool) = case is_orig of {
    true  -> request: HTTP_Request;
    false -> reply:   HTTP_Reply;
};
type HTTP_Request = record {
    request:    HTTP_RequestLine;
    msg:        HTTP_Message;
};
type HTTP_Reply = record {
    reply:      HTTP_ReplyLine;
    msg:        HTTP_Message;
};
type HTTP_RequestLine = record {
    method:     HTTP_TOKEN;
    :           HTTP_WS;   # an anonymous field has no name
    uri:        RE/[[:alnum:][:punct:]]+/;
    :           HTTP_WS;
    version:    HTTP_Version;
} &oneline, &let {
    bro_gen_req: bool = bro_event_http_request(
        $context.connection.bro_conn,
        method, uri, version.vers_str);
};
type HTTP_ReplyLine = record {
    version:    HTTP_Version;
    :           HTTP_WS;
    status:     RE/[0-9]\{3\}/;
    :           HTTP_WS;
    reason:     bytestring &restofdata;
} &oneline, &let {
    bro_gen_resp: bool = bro_event_http_reply(
        $context.connection.bro_conn,
        version.vers_str, to_int(status), reason);
};
type HTTP_Version = record {
    :           "HTTP/";
    vers_str:   RE/[0-9]+\.[0-9]+/;
};
type HTTP_Message = record {
    headers:    HTTP_Headers;
    body:       HTTP_Body(HTTP_HeaderInfo(headers));
};
type HTTP_Headers = HTTP_Header[] &until($input.length() == 0);
type HTTP_Header = record {
    name:       HTTP_TOKEN;
    :           ":";
    :           HTTP_WS;
    value:      bytestring &restofdata;
} &oneline, &let {
    bro_gen_hdr: bool = bro_event_http_header(
        $context.connection.bro_conn,
        $context.flow.is_orig, name, value);
};
type HTTP_Body(hdrinfo: HTTP_HeaderInfo) =
            case hdrinfo.delivery_mode of {
    CONTENT_LENGTH -> body: bytestring &chunked,
                            &length = hdrinfo.content_length;
    CHUNKED        -> chunks: HTTP_Chunks;
    default        -> other: HTTP_UnknownBody;
};
type HTTP_Chunks = record {
    chunks:     HTTP_Chunk[] &until($element.chunk_length == 0);
    headers:    HTTP_Headers;
};
type HTTP_Chunk = record {
    len_line:   bytestring &oneline;
    data:       bytestring &chunked, &length = chunk_length;
    opt_crlf:   case chunk_length of {
        0       -> none: empty;
        default -> crlf: bytestring &oneline;
    };
} &let {
    chunk_length: int = to_int(len_line, 16);  # in hexadecimal
};
```

Evaluation

- Comparison of hand-written parsers and binpac generated parsers for the Bro traffic analysis engine

Protocol	Hand-written			binpac		
	LOC	CPU Time (seconds)	Throughput	LOC	CPU Time (seconds)	Throughput
HTTP	1,896	538–541	244 Mbps / 36.7 Kpps	676	442–444	298 Mbps / 44.7 Kpps
DNS	1,425	37.3–37.5	18.6 Mbps / 13.3 Kpps	698	44.7–44.8	15.6 Mbps / 11.1 Kpps

Future Work

- Add support for languages other than C++
- Evaluate reusability by using code with systems other than Bro

- Note:

  binpac is open-source and is now a part of the Bro distribution

References

- P. Ruoming, V. Paxson, L. Peterson, R. Sommer. binpac: A yacc for Writing Application Protocol Parsers. *IMC'06*. October 25-27, 2006. http://conferences.sigcomm.org/imc/2006/papers/p29-pang
- Declarative Programming. http://en.wikipedia.org/wiki/Declarative_programming
- binpac User Guide http://www.bro-ids.org/wiki/index.php/BinPAC_Userguide

Questions?