Binspector
Documentation & Reference

Table of Contents

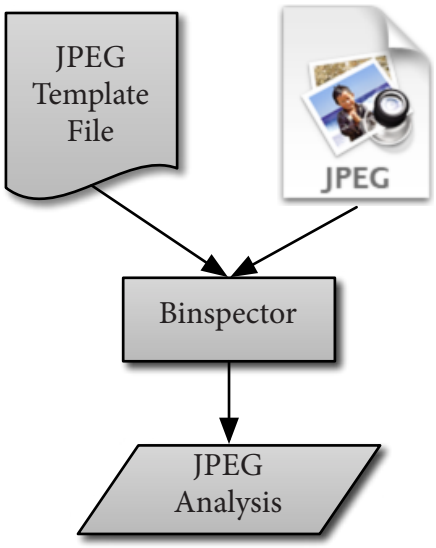
Document History

Author	*Date*	*Description*
`fbrereto`	2011-05-20	Initial Draft
`fbrereto`	2011-06-03	Added info about `eof` slot
`fbrereto`	2011-06-07	Added info about bit work and `peek`
`fbrereto`	2012-02-28	Added info about the `enumerated` language construct
`fbrereto`	2012-03-13	Added info about sentries, `shuffle`, `die`, `default`, etc.
`fbrereto`	2012-03-30	Added info about `include`, `position` and its operations.
`fbrereto`	2012-04-02	Merged Binspector and Hairbrain (tools and docs)

Overview

Binary file formats have been around as long as computing. Though they are an ideal format for a machine to read and write, for the average human the results of these I/O operations can be very hard (if not impossible) to interpret contextually.

Binspector gives a user the ability to inspect and analyze the contents of a binary file *in-situ*. In order to accomplish this the Binspector first requires a *template file* which describes the internal structure of a binary file format. The template file is then given to Binspector to provide a means of interpreting the contents of a binary file:



The purpose of this document is to give the user a high-level overview of the binary file format template language. With this language a user will be able to create their own template files for most binary file formats. From there it should be possible to process binary images and use the Binspector command line interface to further analyze the results.

Binspector Uses

Given a well-formed binary file format template, there are several possible use cases for Binspector:

Validation

Does a given document parse successfully? In other words if the template is a format representation, does a given file model that representation? Can it be asserted that a binary file is an instance of a certain format?

Interpretation

What does an arbitrary bit sequence within a binary file *mean*? What is the context of the sequence and what does it describe in light of that context?

Analysis

Are there any errors within a binary file? Are there any logical incongruities, invariants, or other requirements a binary file fails to meet? What might need altering in an attempt to repair a corrupt file?

Fuzzing

What bit sequences should be changed that might cause adverse side effects within a body of code trying to read this file? When a corrupt file reveals weaknesses within a body of code, what in its contents caused the unintended behavior?

Language Grammar

> » *Note:* For some, the language grammar specification may be completely uninteresting; in such case you can skip to the next section. I promise you won't miss much.

The language is built on top of ASL's common expression language (CEL), so there are constructs mentioned in the grammar that are not detailed here. (For more information on CEL see the ASL documentation at its web site.)

```
        translation_unit   = { struct_set }
        struct_set         = [ struct | pp_statement ] { struct_set }
        struct             = “struct” identifier ‘{‘ { statement_set } ‘}’
        statement_set      = scope_or_statement { statement_set }
        scope_or_statement = conditional_scope | enum_scope | sentry_scope | statement
        conditional_scope  = if_scope { else_scope }
        if_scope           = “if” ‘(‘ expression ‘)’ scope_content
        else_scope         = “else” scope_content
        scope_content      = ‘{‘ { statement_set } ‘}’ | scope_or_statement
        enum_scope         = “enumerate” ‘(‘ expression ‘)’ enum_content
        enum_content       = enum_entry_list | enum_entry_map
        enum_entry_list    = ‘[‘ enum_list_item_set ‘]’
        enum_list_item_set = enum_list_item { ‘,’ enum_list_item_set }
        enum_list_item     = expression
        enum_entry_map     = ‘{‘ enum_map_item_set { enum_map_default } ‘}’
        enum_map_item_set  = enum_map_item { enum_map_item_set }
        enum_map_item      = expression ‘:’ scope_content
        enum_map_default   = “default” ‘:’ scope_content
        sentry_scope       = sentry ‘(‘ expression ‘)’ scope_content
        statement          = [ typedef | unnamed_statement | named_statement ] ‘;’
        unnamed_statement  = [ notify | summary | die ]
        named_statement    = [ invariant | constant | skip | slot | signal | field ]
        typedef            = “typedef” field_type identifier
        field_type         = named_field | atom_field
        named_field        = identifier
        atom_field         = [ “float” | “unsigned” | “signed” ] expression [ “big” |
“little” | expression ]
        notify             = “notify” argument_list
        summary            = “summary” argument_list
        die                = “die” argument_list
        invariant          = “invariant” identifier ‘=’ expression
        constant           = “const” identifier ‘=’ expression { “noprint” }
        skip               = “skip” identifier ‘[‘ expression ‘]’
        field_size         = ‘[‘ { [ “while” | “terminator” | “delimiter” ] ‘:’ }
expression ‘]’ { “shuffle” }
        slot               = “slot” identifier ‘=’ expression
        signal             = “signal” identifier ‘=’ expression
        field              = field_type identifier { field_size } { offset }
        offset             = ‘@’ expression
        pp_statement       = pp_include
        pp_include         = “include” string
```

» *Note:* There are bugs in the grammar and not all corner cases in the grammar are currently implemented.

» *Note:* Comments are not part of the grammar. However the lexical analyzer *does* allow for and will skip C- and C++-style comments within the template. They are not passed on to the parser and are not a part of the resulting AST.

Hello, World!

As no language is ever truly valid until one can write *Hello World* for it, let us get the formalities out of the way:

```
struct main
{
    notify “Hello, World!”;
}
```

All of the above will be further explained below.

Basic Building Blocks

Structures

Structures are the top-level encapsulating constructs used to build out the template file. Each template file is required to have at least one structure, and one of those structures must be named `main`. At the time Binspector interprets a binary file it will begin with the `main` structure and work its way through the rest of the template file based on its description. The structures can be defined in any order in the template file, but no two may have the same name.

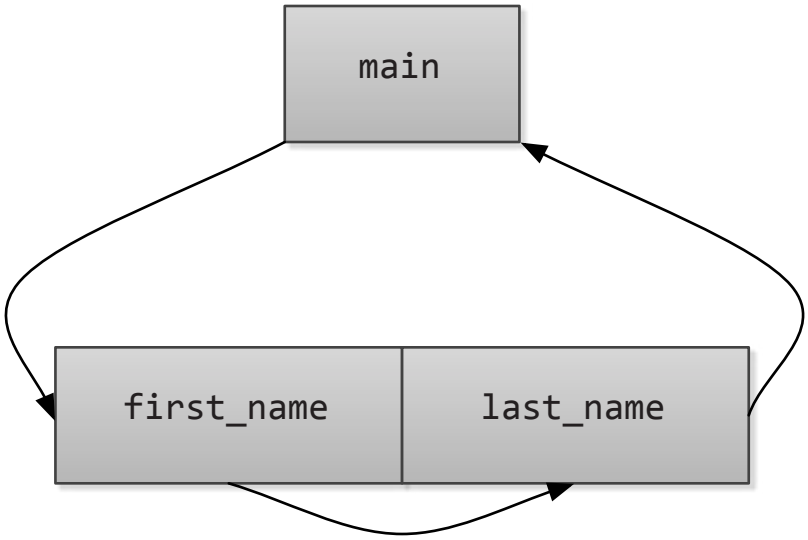
Here is an example of a complete, well-formed and utterly boring template file:

```
struct main
{
}
```

In the above example we have a single structure, `main`, which is empty. Structures can refer to other structures that have already been declared, and it is through this method that more complicated structural relationships can be defined within the binary file format:

```
struct c_string
{
}

struct main
{
    c_string first_name;
    c_string last_name;
}
```

Which will result in the following *structure hierarchy*, where `first_name` and `last_name` are both *children* of `main` and *siblings* of one another. Binspector processes earlier fields in the definition first:



Although in the above example we are using multiple structures to define a more complicated binary file format, we still have not given Binspector anything really useful information. What is missing from this equation is the second essential building block: *atoms*.

Atoms

Atoms are discrete fields within a structure that give that structure context and size. Atoms can be defined in many number of ways that add power and flexibility to the template file as Binspector uses it for interpretation. There are a handful of components to an atom, each of which is worth going into detail. They are the *field type*, *identifier*, *field size* and *offset*.

> » *Note:* Though we are talking about *identifier*, *field size* and *offset* as attributes of atoms, they also apply to structures (unless otherwise noted). (*Field type* is an attribute unique to atoms.)

Field Type

The field type for an atom is broken up into three distinct parts: the declarations for signedness, size, and endianness. Each atom declaration requires all three an in the order mentioned.

Signedness

The sign of the atom can be one of three values: `signed`, `unsigned` or `float`. The first two each have the meaning you would expect, while the latter refers to a floating-point representation of the data.

> » *Note:* At this time the use of `float` is restricted to atoms of 32 and 64 bits in size. The use of `float` with other sizes will result in an error.

Size

The size of the atom is defined in the number of bits the atom consumes in the binary file format. Typically the values are byte-aligned (e.g., 8, 16, etc.) but do not need to be.

> » *Note:* The maximum size of a single atom is limited to 64 bits.

Endianness

The endianness of the atom can be one of two values: `big` and `little`. It is also possible to use an expression to represent the endianness of an atom. This is useful for some binary formats (e.g., TIFF) where the endianness could be either way, and it is up to the reader of the binary file to discern the endianness of the data. In those situations the expression must return a boolean value. When the value is `true` the atom is `big` endian, otherwise it is `little` endian.

> » *Note:* Though unnecessary, atoms of bit sizes 8 or less still must specify an edian interpretation. This is a current limitation of the language grammar.

Example

```
struct main
{
    float 32 big scale;
}
```

Here we have a single 32-bit atom in the `main` structure named `scale` to be interpreted as a floating-point value. We have just given substance to a template file, as we have instructed Binspector how to interpret the first 32 bits of some imaginary binary format. Atoms inserted into a structure will be analyzed based on the current "read position", which advances by the number of bytes interpreted by an atom when one is specified.

Identifier

The identifier for the atom is its name. These must be unique within a structure. Additionally no field may be named `main` or `this`.

Example

```
struct main
{
    unsigned 16 big some_word;
}
```

Here we have a single atom in the `main` structure with the identifier (name) `some_word`.

Field Size

Atoms can describe static and dynamic arrays in addition to singletons. The field size is an optional parameter for atoms that instruct Binspector to bundle up multiple atoms under a single field in the structure. There are several field size declarations, each with different nuances, however all field size declarations will build contiguous arrays. The declaration types are the

integer, *while*, *terminator*, and *delimiter*; each will be detailed below. All field size descriptions are wrapped in brackets to denote the fact an array is being constructed. Additionally, all field size expressions are evaluated at the time the field is to be read.

Integer

The integer field size is the most familiar and is used to define a discrete array of entries. For example:

```
struct main
{
    unsigned 16 big magic_word[2];
}
```

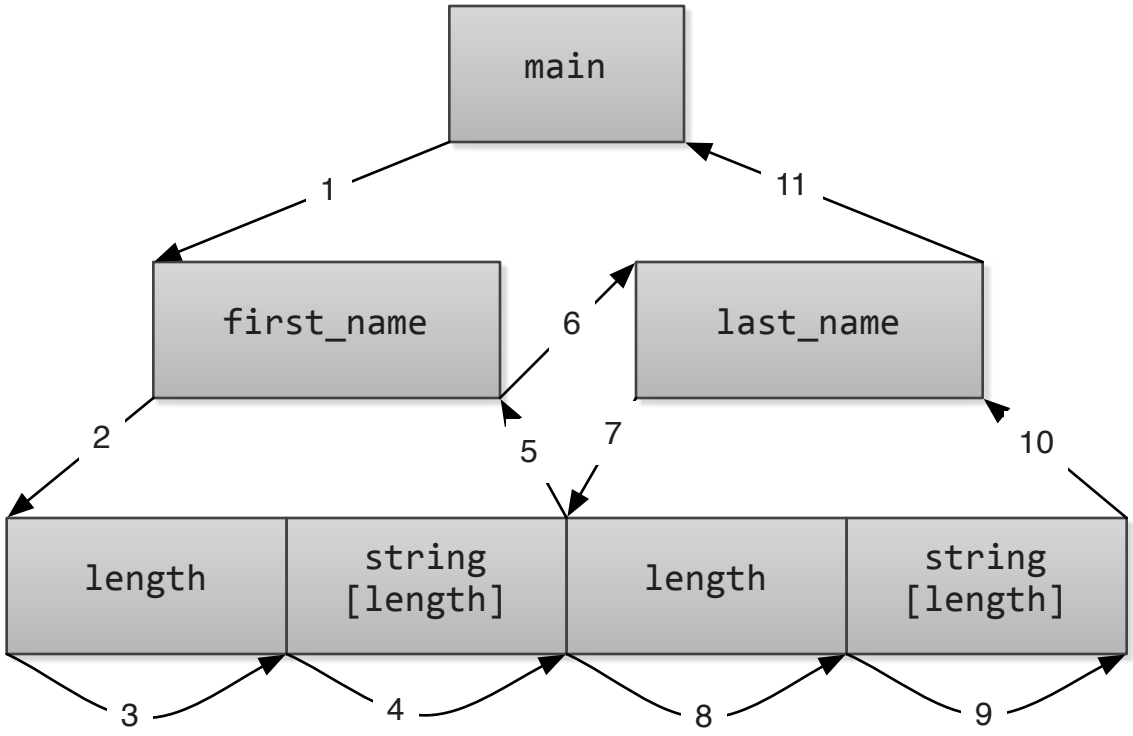
Here we have a single field `magic_word` that Binspector should use to interpret the first 32 bits of the binary format in the form of two 16-bit values.

> » *Note:* An array of zero elements is valid. That being said it is possible to use Binspector's template language to avoid their use to make output easier to read. See the section on conditional statements for more information.

In addition to a static field size it is possible to derive the length of a field based on values previously found in the binary format. For example, a pascal string is an 8-bit length value followed by that length's worth of 8-bit characters that constitute the string. To describe such a structure to Binspector one might declare it as follows:

```
struct pascal_t
{
    unsigned 8 big length;
    unsigned 8 big string[length];
}

struct main
{
    pascal_t first_name;
    pascal_t last_name;
}
```

In this example we are using values found within the binary file to continue the file's interpretation. We started by wrapping the length-prefixed pascal string into its own structure, `pascal_t`. The first field is an 8-bit atom we called `length`. We then used the value of `length` to define the field size for the second atom, `string`. Finally we leveraged the `pascal_t` structure twice within `main`. Visually, Binspector is walking the structural hierarchy created by the interpretation of the binary file's bytes based on the template file:



It is the bottom-most row of nodes that represent actual data in the binary file; the structures are important as they add additional context and structure to the data, but it is the atoms themselves that define the data's actual meaning.

To bring the two uses of an integer field size together one could also write:

```
struct main
{
    pascal_t name[2]; // first and last name
}
```

And the template file would produce a functionally equivalent result.

While

The basics of a `while` declaration are that Binspector will continue to extend the size of the field until the `while` expression evaluates to `false`. An an example:

```
struct main
{
    slot done = false;

    other_structure_t my_array[while: !done];
}
```

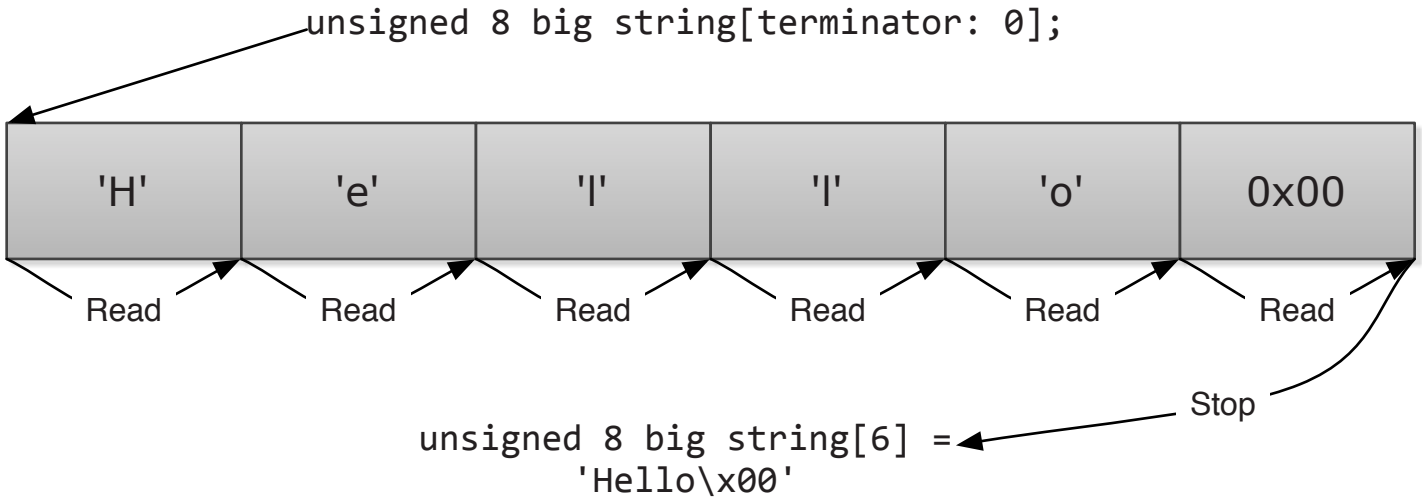
(We'll get more into slots and signals later; for the sake of the example a slot is a variable.) Here the `while` expression will continue to return `true` (and thus expand upon the array) until something inside `other_structure_t` alters the value of `done` from `false` to `true`, ending the while loop.

Terminator

An atom with a terminated field size is one where Binspector will continue to inspect a binary file's contents and grow the array until said terminator is read in the file. At that time, the terminator will be included in the array before Binspector continues to the next atom for interpretation. The best example of this kind of field in a template file would be the traditional NULL-terminated string:

```
struct main
{
    unsigned 8 big string[terminator: 0];
}
```

Here Binspector will continue to read bytes in the file until the terminator's value (`0`) is found. The value is included as the last entry in the atom's array, and the field is complete. During interpretation, here is Binspector's process is forming the string:



» *Note:* The bit size of the terminator must match the bit size of its field.

» *Note:* Terminators cannot be applied to structures.

Delimiter

An atom with a delimited field size is very similar to the terminated type, however there are two main differences. The first is that the delimited value is *not included* in the atom's resulting array. As a consequence of the this the second difference is that the delimiter's bit size need not be the same as the atom's. The most common use case for a delimited atom declaration is when Binspector should skip over some uninteresting portion of a binary file until a sought-after piece is found. For example in a JPEG template file one might want to skip over the image data stream, requiring a delimiter field until the end of image marker is found:

```
struct main
{
    //... prior JPEG template file declaration
    unsigned 8  big image_stream[delimiter: 0xFFD9];
    unsigned 16 big eoi_marker; // will be 0xFFD9
}
```

Here we have an 8-bit array filled with image stream data that the template file is otherwise uninterested in. The delimiter however is a 16-bit value, which is legal because the value will not be included in the `image_stream` array. As noted in the example the following 16-bit value will be `0xFFD9`, the end of image marker Binspector found while processing the `image_stream` atom.

» *Note:* At this time the bit size of the delimiter is deduced by the number of bits required to represent the value described in the template file.

» *Note:* Delimiters cannot be applied to structures.

» *Note:* When you have the option, use delimiters in lieu of `peek`; the former is far more efficient.

Shuffle

Whenever an field size is specified for a field the option is available to add the `shuffle` keyword to the end of the definition. While this does nothing for analysis it does earmark this field for fuzzing, which will then produced alternatives of this array by shuffling around the its elements in place.

Offset

Binary file formats are not always written to disk in a contiguous fashion. Sometimes formats specify offsets relative to locations in the data where other data can be found. An example of this is the TIFF IFD metadata format, where values beyond a certain size are appended to the end of the metadata block, and offsets are specified within the metadata itself as to where the extended information resides. Binspector provides means for fetching remote data and analyzing it as part of a structure located elsewhere in the file.

Within Binspector's template files offsets are always specified with absolute values. Many binary formats use relative offsets so Binspector's language has primitives to assist in converting between relative and absolute offsets.

Example

In the following example let us extend our `pascal_t` structure from above to include an extra field that specifies the remote location for the string data as an absolute offset:

```
struct pascal_remote_absolute_t
{
    unsigned 8  big length;
    unsigned 32 big absolute_offset;
    unsigned 8  big string[length] @ absolute_offset;
}
```

In the example our first atom is the length of the string and the second is the remote offset. Since the value is absolute we can use it without modification in the third atom definition. Binspector will seek to that absolute offset within the file and proceed to read `length` 8-bit values to constitute the string atom for our structure.

In the case we are dealing with remote offsets we need to convert them to absolute offsets before Binspector can find the data correctly. In such case we need to know which field is the basis for the relative offset, and construct an expression to use its offset to compute the final, absolute offset. In a relative-offset-based remote `pascal_t` structure, given that the remote offset is relative to the length byte of the string, we might have:

```
struct pascal_remote_relative_t
{
    unsigned 8  big length;
    unsigned 32 big relative_offset;
    unsigned 8  big string[length] @ startof(@length) +
                                          relative_offset;
}
```

In this example we use the `startof()` routine to fetch the absolute offset of the `length` atom in the file and use it in conjunction with the relative offset of the string to compute the absolute offset of the string data.

> » *Note:* Within routine calls it is necessary to refer to fields (structures or atoms) with a prefixed `@` symbol, otherwise the value of the field will be used instead of the field itself. This is loosely similar to pass-by-reference v. pass-by-value. However when the field is part of an expression the `@` should be omitted (e.g., `startof(main.foo.bar)`).

For readability's sake it would be fine to use a `const` field to construct the absolute offset before using it:

```
struct pascal_remote_relative_t
{
    unsigned 8  big length;
    unsigned 32 big relative_offset;
    const absolute_offset = startof(@length) + relative_offset;
    unsigned 8  big string[length] @ absolute_offset
}
```

Reading remote data does not affect the read position once the reading is complete. For example given the following fields:

```
unsigned 8 big offset;
unsigned 8 big remote_data @ offset;
unsigned 8 big some_value;
```

The bytes `offset` and `some_value` are adjacent to one another in the binary file. However in the analysis `remote_data` will be between them as the remote data is brought into its interpreted location in the file at that time. In addition an analyzed structure's starting and ending offsets (`startof()` and `endof()` values) are only affected by its nonremote (local) data, though this may change in a later version.

Advanced Building Blocks

There are several additional parts of the Binspector language intended to make life a little simpler for those who use it. Though not strictly necessary like atoms and structures, these additions can help users ensure correctness, catch errors (both in the template file and in the binary), and increase clarity and flow of a template.

Invariants

Invariant declarations are expressions used to increase the reliability of a binary file's interpretation. At the point an invariant field is declared it will be evaluated within the structure and must return a boolean. If the boolean is `false` a notification is posted of the invariant's failure and analysis of the binary file is halted.

As an example consider the segmented JPEG file format. At the start of the format one is told to expect a start of image marker which is a specific value. Given that requirement within the binary file format one can add an invariant to assert the file is meeting the specification requirements of the start of image marker's presence and value:

```
struct header_t
{
    unsigned 16 big soi; // start of image marker
    invariant is_soi = soi == 0xFFD8;
}
```

In the structure above `is_soi` will be evaluated by Binspector the moment it is reached. If the invariant is `true` file processing will proceed, otherwise it will cease (implying e.g. an ill-defined template file, a corrupt binary file, etc).

Die

A `die` statement gives the template writer the ability to prematurely halt binary file analysis with an output message. Typically `die` statements are used when something unexpected has happened that could jeapordize further analysis.

Example

```
struct main
{
    unsigned 32 big version;
    if (version != 42) die “Wrong version number found”;
}
```

» *Note:* Though invariants and `die` statements are similar, `die` statements always fail and allow a descriptive message to be added to the failure.

Constants

Constant fields are a way for the writer of a template file to encapsulate value calculations while still keeping the template file itself readable and (hopefully) maintainable. They do not consume any bytes in the binary file and have no “size”.

As an example a constant field was used to make the `pascal_remote_relative_t` structure easier to read by calling out the absolute value offset calculation and giving it a label:

```
struct pascal_remote_relative_t
{
    unsigned 8  big length;
    unsigned 32 big relative_offset;
    const absolute_offset = startof(@length) + relative_offset;
    unsigned 8  big string[length] @ absolute_offset
}
```

Constant fields are strongly typed and their type is deduced from the result of the expression used to define them:

```
struct main
{
    const string_const = “Hello”; // OK- string type
    const integer_const = 42; // OK- integer type
    const sum_const = string_const + integer_const; // Error!
          // The types in the addition expression do not match.
}
```

» *Note:* Constants use a caching mechanism to speed them up, so the first time they are evaluated is the only time they are evaluated.

Type Definitions

Type definitions are a way of coalescing commonly used atoms and (less frequently) structures to ensure the template file is properly constructed.

Consider the TIFF binary file format which permits data to be either little- or big-endian. In such a case we need to discern the endianness of the data and use that in further specification of the template file, or we run the risk of misinterpreting the remainder of the TIFF. Within TIFF there is a marker that allows the reader to deduce the endianness of the values to come:

```
unsigned 16 big tiff_header; // 0x4949 (little) or 0x4d4d (big)
invariant valid_tiff_header = tiff_header == 0x4949 ||
                              tiff_header == 0x4d4d;
```

From here we want to inform whatever atoms follow which endianness should be used in interpreting the data found. As such we can construct a solution like so:

```
const is_big_endian = tiff_header == 0x4D4D;
unsigned 16 is_big_endian tiff_tag_mark; // 42
unsigned 32 is_big_endian ifd0_offset;   // usually 8 for IFD 0
```

(Recall that we can specify an expression to define endianness: a true expression means big-endian while a false one means little-endian.) The problem is that atom definitions quickly become hard to read and maintain. The `typedef` was introduced to allow the user to specify an atom type once and reference it whenever that type should be interpreted by Binspector:

```
typedef unsigned 16 is_big_endian tiff_word_t;
typedef unsigned 32 is_big_endian tiff_long_t;
tiff_word_t tiff_tag_mark; // 42
tiff_long_t ifd0_offset;   // usually 8 for IFD 0
```

The result is an atom definition that is easier to read and those fields depending on the TIFF endian marker are easier to maintain.

Typedefs are scoped to the structure in which they are defined *and any substructures* it may have. In addition typedefs can be redefined in substructures and that redefinition will have the same lifetime as the sub-structure that provided it. When the substructure falls out of scope, the typedef will return to its prior value.

Notifications

Notify is a simple way to get live feedback of Binspector's progress during analysis. With Binspector any data can be presented in a comma-delimited set of expressions, each of which will be evaluated and their results send to stdout.

As an example, consider our *Hello World!* template file from the beginning of this document:

```
struct main
{
    notify “Hello, World!”;
}
```

Comma separated values are concatenated and output:

```
unsigned 32 big relative_offset;
const absolute_offset = startof(@length) + relative_offset;
notify “Reading value from offset ”, absolute_offset;
```

Summaries

Many file formats have complex structures that take some drilling down in order to figure out exactly what's inside. Summaries add user-friendly notes to structures during output to make it easier to navigate large and/or complex structures.

Example

```
struct rect_t
{
    unsigned 32 big top;
    unsigned 32 big left;
    unsigned 32 big bottom;
    unsigned 32 big right;

    summary 'L: ', left, ', T: ', top,
            ', R: ', right, ', B: ', bottom;
}
```

When the above structure is analyzed and output to an interface, the summary might come back looking something like this:

```
(rect_t) bounds (L: 0, T: 0, R: 0, B: 0)
```

» *Note:* The `summary` construct does not create an AST node; rather it is evaluated and stored in an internal field to the structure in which it is defined. Therefore the summary value cannot be used in subsequent expressions. Subsequent `summary` constructs in the same structure override whatever was set previously.

Skip Fields

Many times a particular template file will be written to focus on a specific aspect of a binary file format. For example a template file may only want to analyze the Exif metadata embedded within a JPEG file and avoid the rest. (Other good uses of the skip field are unimplemented substructures and remote data passover). The skip field is used to pass over a specified number of bytes in the file:

```
struct pascal_t
{
    unsigned 8 big length;
    skip dont_care[length]; // not interested what's in the string
}
```

In this case then Binspector will pass over `length` bytes to preserve the correctness of further interpretation in the binary file.

» *Note:* Unlike other size values in the template language, skip field sizes are specified in *bytes*, not bits.

Slots and Signals

Slots and signals are a means by which ancestral structures can communicate with one another. A slot is a variable specified in a structure and given a value- in this way they are very much like constant fields. The way in which they differ though is that they can be redefined by a signal defined in a sub-structure, or even later on in the same structure.

Example

```
struct main
{
    slot done = false; // done is now set to false
    //... do something!
    signal done = true; // done is now set to true
}
```

Signals will trigger Binspector to search up the current structure hierarchy to find a slot with the same name. Once the slot is found its *value expression* is set to the value expression of the signal. Note that its value is not set: the expression is. This means that one could make reference to variables in the value expression of a signal and the slot will then reference those variables without having evaluated them yet. The evaluation of those variables will take place when the slot is first used.

» *Note:* Slots use a caching mechanism to speed them up, so the first time they are evaluated is the only time they are evaluated. This cache is reset when a signal modifies a slot's value expression.

As a real-world use case, the JPEG binary format specifies that the start of stream marker will indicate the image data stream will immediately follow. In order to describe such behavior in Binspector we would use a *slot* to set up a segment-reading while loop and a *signal* to trigger when the start of stream marker is found.

Example

```
struct segment_t
{
    unsigned 16 big marker;

    if (marker == 0xFFD9)
          signal sos_found = true;

    // further jpeg segment processing...
}

struct main
{
    slot sos_found = false;
    segment_t segment_set[while: !sos_found];
    unsigned 8 big image_stream[delimiter: 0xFFD9];
    segment_t end_of_image_marker;
}
```

Special `eof` *Slot*

Most file formats are structured such that the end of the file is deduced based on the data in the file (markers, length specifiers, etc.) However some formats have no such indication (an example is (oftentimes human readable) UTF-8 encoded multibyte text) and we need to be notified when the end of the input has been reached. Therefore a special-use slot exists to signal when the end of the binary file has been reached. Specifically, Binspector will set the slot `eof` to `true` in that event.

Example

```
struct utf8_t
{
    unsigned 8 big byte;
    // further utf-8 processing...
}

struct main
{
    slot eof = false;

    utf8_t string[while: !eof];
}
```

In the above example analysis will end normally when the end of the binary file has been reached.

» *Note:* You must define the `eof` slot in your template file explicitly: it will not be defined for you. In the event the `eof` slot is not defined for the template file and the end-of-file is reached an analysis exception will be thrown. At that time the binary file's analysis will halt.

» *Note:* It is not always desireable to define the `eof` slot in a template file. The general rule is to define it only when the binary format provides no other means of end-of-file notification. That way if you are expecting to know when the end of the file should be and you hit it beforehand you will get an error. From that error one can start to discern what went wrong (bad template file, bad binary file, PEBKAC, etc).

Include File Support

Binspector scripts can include others to promote reuse. Currently the only directory used as an include file directory is the same one as the template file passed when Binspector is launched. When an include file is specified the current document being parsed is put on hold until parsing of the include file (and any other which it may include itself).

» *Note:* It is possible to specify other include files with relative file paths, but given that this will be within a template file it is discouraged.

» *To Do:* Add support for multiple include directories from the command line.

Example

```
include ‘descriptor.bfft‘

struct main
{
    // defined in the descriptor template file
    descriptor_t descriptor;
}
```

Conditional Evaluation

As hinted in the slots and signals example, Binspector provides support for basic `if`/`else` conditionals. This gives template files real flexibility in being able to evaluate the data in a binary file and respond accordingly.

The `if`/`else` structures are familiar to C/C++ programmers.

Example

```
struct main
{
    unsigned 16 big version;

    if (version == 1)
    {
        notify “Version 1 file found”;
        version_1_t more_data;
    }
    else if (version == 2)
    {
        notify “Version 2 file found”;
        version_2_t more_data;
    }
    else
        notify “Unknown version found: ”, version;
}
```

» *Note:* In the example `more_data` is the name of two different fields, but since they are defined in separate scopes it is acceptable.

Enumerated Evaluation

Many fields within a binary file can be only one of a small set of values. For example the Photoshop descriptor format uses predefined four-character codes to convey the type of a descriptor entry’s value. Another example would be the BMP file format, whose `header_size` is one of a specific set of values based on the version of the BMP spec a given file adheres to. In fact the example given in the *Conditional Evaluation* section might be better written using the `enumerate` construct, as there are only two valid values expected.

The advantage of using `enumerate` over a cascading lineup of `if/else` statements is twofold. First it is much more readable and the intent of what is being done is more clear. Second (and more importantly) the fuzzer uses the `enumerate` construct to develop intelligent analysis about a field and how it might be modified to expose weaknesses in file import code. Knowing the set of enumerated options that are technically valid but may not be contextually appropriate lets the fuzzer generate 'better' corruptions than guesswork alone.

Example

In the JPEG file format the second byte of every segment marker denotes the type of segment that is to follow:

```
enumerate (app_marker2)
{
    0xC0: nonapp_marker_t xc0;
    0xC2: nonapp_marker_t xc2;
    0xC4: nonapp_marker_t xc4;
    0xDA: sos_marker_t    sos;
    0xDB: nonapp_marker_t xdb;
    0xE0: app0_marker_t   app0;
    0xE1: app1_marker_t   app1;
    0xE2: appN_marker_t   app2;
    0xE3: appN_marker_t   app3;
    0xEC: appN_marker_t   app12;
    0xED: appN_marker_t   app13;
    0xEE: appN_marker_t   app14;
    0xDD: dri_marker_t    dri;
}
```

In the example what we see looks very much like a C switch statement: For every possible enumerated value there are a collection of field(s) that should be processed for that value.

» *Note:* If an identifier's value cannot be found as an enumerated option, an error will be emitted and analysis of the binary file is halted. The exception to this rule is when a `default` case is provided.

» *Note:* Currently the `enumerate` construct only works for atoms. That is to say it won't work with arrays (strings), structs, etc.

Default Enumerate Cases

Enumerate constructs allow for the last item in the scope to be a `default` case, or the path chosen when no other enumerated options match the found value.

Example

```
enumerate (appN_marker)
{
    0xE0:    app0_marker_t   app0;
    0xE1:    app1_marker_t   app1;
    default: appN_marker_t   appN;
}
```

Here only values `0xE0` and `0xE1` are explicitly handled; all others will use the `appN_marker_t` struct.

» *Note:* In general one should supply a `default` case only when needed (e.g., when an enumeration is not required to meet a fixed set of values, allowing for alternatives.) In many cases it is correct to omit a `default` case, permitting Binspector to halt analysis when an unknown enumerated value is found.

» *Note:* The `default` case (if supplied) must be the last option in an `enumerate` construct.

Empty Enumerate Constructs

Though they add little for analysis (as they are functionally invariants), "empty" `enumerate` constructs find their real value when fuzzing.

Example

The two `enumerate` constructs below are functionally equivalent (the former being syntactic sugar for the latter.)

```
unsigned 16 big depth; // supported values are 1, 8, 16 and 32

enumerate (depth) [ 1, 8, 16, 32 ]

enumerate (depth)
{
    1:  { }
    8:  { }
    16: { }
    32: { }
}
```

» *Note:* There is no semicolon after the `[ ... ]`-style `enumerate` construct, no matter how tempting it may be to add one!

» *Note:* There is no `default` case for the empty enumerate construct.

Sentries

Oftentimes file formats contain length-prefixed blocks within them, meaning a block of data is preceeded (usually immediately) by its size in the file. Complex structures may exist within that block, however, and it can be hard to assert the block being analyzed is still within the bounds set by the length prefix.

When length-prefixed blocks are defined in your template file you can use *sentries* to assert that Binspector does not read beyond a specific offset in the file. After all to do so would be a violation of the file's internal structure; with sentries you will catch the range error long before the analysis has gone off into the weeds. In addition the fuzzer can leverage `sentry` constructs to identify which values in the file are used as block length prefixes and can modify its fuzzing accordingly.

There are two types of sentry values: *relative* and *absolute*. Relative values are specified with integers and are byte-relative to the current position of analysis within the file. Absolute values are specified with an address expression and are relative to the beginning of the file.

» *Note:* It is technically possible to have sub-byte barriers (that is, sentries at the bit level) though specifying them can be tricky.

» *Note:* Relative sentries are generally useful when the length field is not included in the length value (e.g., a pascal string.) Absolute sentries are generally useful when the length field value includes the length field itself (e.g., JPEG APPN segments.)

Example

The following is an example of a relative sentry, in that the value passed to the sentry is an integer value.

```
struct pascal_t
{
     unsigned 8 big length;

     sentry (length)
            if (length != 0)
                  unsigned 8 big string[length];
}
```

The following is an example of a relative offset, as the expression used to define the sentry results in a position within the file instead of an integer.

```
struct jpeg_appn_segment
{
     unsigned 16 big length;

     sentry (startof(@length) + length)
     {
            // Rest of APPN segment...
     }
}
```

Path Deduction

Path deduction in a template file or by the Binspector leverages the notion that a forest of data is being built up about the file. As such there are two fixed-name nodes: `main` which always refers to the topmost node the tree and `this` which always refers to the current structure being analyzed.

To specify an absolute path one should always start from `main`. To specify a relative path from the current node to any child nodes, one should always start the path from `this`:

```
unsigned 16 big absolute_remote_data @ main.foo.bar.offset;
unsigned 16 big other_remote @ startof(this.foo.bar.offset);
```

Paths that begin with neither `main` nor `this` go through a lookup process. Starting at the current node a child field with the specified name is searched for. If it is found it is returned, and the path is further deduced from that node. If it is not found the same search is conducted on parent node. This lookup process repeats until it fails at `main`, at which point an error is emitted to the user.

Using this deduction method it is possible to reach up the tree and across to siblings that have already been analyzed to leverage the information they posses. As an example we look at a portion of the BMPV1 template definition from the end of this file:

```
struct pixel_row_t
{
     struct pixel_t pixel_set[dib_header.width];
     unsigned 8 big padding[dib_header.width % 4];
}

struct main
{
     struct header_t    header;
     struct dib_t       dib_header;
     struct pixel_row_t pixel_row_set[dib_header.height];
}
```

Note how the `pixel_row_set_t`'s `pixel_set` declaration sets its fixed size to `dib_header.width`. Because that path begins with neither `main` nor `this` path deduction is invoked. The first parent of `pixel_row_t` in this description is `main`, which does have a `dib_header` field. From there the `width` subfield of `dib_header` is traversed and its value used for the size of the `pixel_set` array.

» *Note:* Any path specification is only valid for nodes that have already been analyzed by the time the path is evaluated. It is not possible, then, to "look ahead" into an unanalyzed portion of the binary file to use a value. The proper way to fetch such a value would be to use a remote data specification to fetch it prior to requiring its use.

Builtin Expression Commands

There are several commands one can use within template file expressions. Note that many of these commands take "addresses" of fields in the analysis instead of the fields themselves. In the event a template file is ill-formed and a field is used instead of an address you will see an error like:

```
bad_cast: adobe::implementation::forest_iterator<node_t>* ->
name_t:version_1:adobe
```

» *Note:* We'll try to clean up error reporting in future versions of Binspector.

The following is a list of commands available to template file developers along with basic syntax as to how they are invoked.

The position type

Most of the types inherent in these routines are straightforward (integer, string, etc.) That being said there is a custom Binspector type used in some of these routines that should be kept in mind. A `position` is an offset (or address) into the file being analyzed. In one sense it behaves much like an integer (in fact there are operations to covert between them) however it is worth noting that positions can specify bit-level offsets, not just byte-level offsets. As such it is important to note most of the operations to convert between positions and integers are bytes, not bits. The documentation tries to be clear about when positions are expected or returned.

Template File Command Reference

```
integer sizeof(@field)
```

Returns the size of the `field` specified in bytes.

```
integer sizeof(@field1, @field2)
```

Returns the number of bytes used from the start of `field1` to the end of `field2`.

```
position startof(@field)
```

Returns the offset to the first byte of `field`.

```
position endof(@field)
```

Returns the offset to the last byte of `field`.

» *Note:* In STL parlance we're talking about the offset to `back()`, not `end()`.

```
integer byte(integer)
```

Returns the value of the byte found at the specified offset as an unsigned integer.

```
integer peek()
```

Returns the next byte in the binary file without advancing the read pointer.

» *Note:* It might be worthwhile to extend this at some point to be able to peek a certain number of bits up to 64. Currently you'll always get the next byte.

```
integer card(@field)
```

Returns the cardinality of `field` if it is an array root. Example:

```
unsigned 8 big string[terminator: 0];
const string_length = card(@string); // set to string array size
```

`string str(@field)`

Returns a character string representation of `field`.

» *Note:* In the event `str` is used to convert an array that terminates with a zero-value to a string the NULL will be omitted from the string. This gives template file writers the ability to interpret zero-terminated character arrays as C strings (i.e., not counting the terminator towards the string.)

`string path(@field)`

Returns the absolute path to `field` as a string. If `field` is omitted, `this` is implied.

`integer indexof(@field)`

Returns the array element index of the specified field. If `field` is not an element in an array an error is emitted. Typically this is used as `indexof(@this)` to get the array element index of the current structure being analyzed. This can be useful for cross-hierarchy access when two arrays are related and are the same size.

Example

```
struct main
{
    unsigned 8 big length;
    foo_t foo[length];
    baz_t something_else; // is not an array, preventing foo
                          // and bar from being wrapped into
                          // a structure of their own.
    bar_t bar[length];
}
```

Above, if an element in `bar` wanted to access data in its corresponding element in `foo`, it could fetch its own index with `indexof(@this)` and pass that to an expression index into `foo`.

» *Note:* This command promotes strangely-defined template files, and *may* be removed in the future. Its use is not encouraged.

`integer fcc(string)`

Returns an unsigned integer equivalent of the string passed. The most common use case for this would be to convert four character codes to integer values for comparison (hence the name, `fcc`.)

» *Note:* Though its name implies otherwise, `fcc` will convert any string up to four characters in length.)

Example

```
unsigned 32 big signature;

invariant ok_signature = signature == fcc('8BIM');
```

`position gtell()`

Returns the current read head position.

Example

```
const my_offset = gtell();
```

`integer ptoi(position)`

Returns the byte portion of the position as an integer, which is useful for performing math operations on the value.

Example

```
const padding_amount = 4 - (ptob(gtell()) & 3);
```

» *Note:* Although byte positions are 64 bit values they are truncated to 32 bits by this operation.

`position itop(integer)`

Takes an integer as a byte position and returns that position.

Example

```
const padding_amount = 4 - (ptob(gtell()) & 3);
```

`position padd(...)`

Takes one or more positions or doubles and returns their sum as a position.

Example

```
const sentry_end = padd(@startof(length), length, -1);
```

`position psub(arg1, arg2)`

Takes two of either positions or doubles and returns the subtraction of the second from the first as a position.

Example

```
const prior_byte = psub(startof(@index), 1);
```

Command Line Interface

After the binary file is analyzed a command line is presented to the user that allows them to explore the results of the analysis. We'll use a 50x50 BMPV1 file as a running example throughout this section to highlight the features of the command line interface. A template file for BMPV1 can be found in the section following.

The first command line presented to the user looks like this:

```
$main$
```

The value between the `$` is the current *path*. `main` is always the first structure used to interpret the binary file and as such will always be at the beginning of any path. As the user moves in an out of substructures and arrays the path will tell the user where they are in the binary file interpretation.

Binspector uses a directory structure metaphor for navigating a file's analysis. Under the metaphor atoms would be considered files and structures would be considered directories. Therefore, many of the commands below that deal with navigation have a very POSIX-like syntax about them (e.g., `ls` and `cd`).

Command Line Commands

The following is a list of commands available to the user while interacting with Binspector's command line interface. Alternate shortcuts to each command are described in parentheses after the command itself.

`quit (q)`

Terminates Binspector

`help (?)`

Prints Binspector help

`print_struct (ps) (ls) (ll)`

Displays a synopsis of the structure at the current path. For example:

```
$main$ ll
(main) main
{
    (header_t) header
    (dib_t) dib_header
    (pixel_row_t) pixel_row_set[50]
}
```

`print_branch (pb)`

Displays a complete synopsis of the current structure at the current path. Executing `print_branch` at `$main$` will output the entire contents of the analysis. For leaf structures this command is equivalent to `print_struct`.

`print_string <path> (str)`

Displays the field specified by the path as a string. Values that have no graphical representation (i.e., `std::isgraph(c) == false`) are output as their ASCII value in hex prepended with an `\x`. For example:

```
$main.dib_header$ str bpp
\x18\x0
```

An equivalent command is (note the difference in current path):

```
$main$ str dib_header.bpp
\x18\x0
```

» *Note:* The example isn't very useful as there are no strings in a BMPv1 file, but you get the idea.

`step_in <path> (si) (cd)`

Sets the path to the structure defined by the path. This can be done both relative to the current path and absolutely from `$main$`. For example:

```
$main$ cd header
$main.header$
```

or

```
$main$ cd pixel_row_set[5].pixel_set[5]
$main.pixel_row_set[5].pixel_set[5]$
```

You can also specify an absolute path from `$main$`

```
$main.pixel_row_set[5].pixel_set[5]$ cd main.header
$main.header$
```

For arrays it is possible to step into the array without stepping into an element of that array. It is known as the *array root*:

```
$main$ cd pixel_row_set
$main.pixel_row_set$
```

You can also refer to the current structure with the keyword `this`:

```
$main.pixel_row_set$ cd this[4]
$main.pixel_row_set[4]$
```

`step_out (so) (cd ..)`

Sets the path to be the parent structure of the current path.

You can also step out to the parent structure with `..`, akin to `cd` on a *NIX command line. For example:

```
$main.header$ cd ..
$main$
```

» *Note:* In the case of an array element the parent structure is the array root and not the structure that contains the array.

`top (t)`

Sets the current path to `$main$`

`detail_field <path> (df)`

Prints out detailed information about the path specified. For example:

```
$main$ df this
    path: main
    type: struct
    struct: main
    bytes: [ 0 .. 7625 ]
    size: 7626 bytes (7.44727 KB)
```

Note that the information displayed differs on the field type:

```
$main.header$ df file_size
    path: main.header.file_size
    format: 32-bit unsigned little
    offset: 2
    raw: 0xca 0x1d 0x00 0x00
    value: 7626 (0x1dca)
```

or

```
$main$ df pixel_row_set[1].pixel_set
    path: main.pixel_row_set[1].pixel_set
    type: array
    struct: pixel_t
    elements: 50
    bytes: [ 178 .. 327 ]
    size: 150 bytes
```

For fields that start within a byte their offset will be represented using an `X.Y` notation, where `X` is the byte and `Y` is a number from `1-7` denoting the bit at which the field starts:

```
$main.string[7]$ df b1p2
    path: main.string[7].b1p2
    format: 4-bit unsigned
    offset: 8.2
    raw: 0x0e
    value: 14 (0xe)
```

In the above example the field begin's at bit 2 of byte 8 and extends for 4 bits.

`detail_offset <offset> (do)`

Searches the binary file analysis for the atom that interprets the byte at the provided offset. Currently only local data is included in the search (not remote data) though its inclusion is planned for a future release. For example:

```
$main$ do 1234
    path: main.pixel_row_set[7].pixel_set[48].blue
    format: 8-bit unsigned
    offset: 1234
    raw: 0xff
    value: 255 (0xff)
```

Sometimes the result will be in the middle of an atom, in which case the whole atom will be detailed. For example:

```
$main$ do 16
    path: main.dib_header.header_size
    format: 32-bit unsigned little
    offset: 14
    raw: 0x0c 0x00 0x00 0x00
    value: 12 (0xc)
```

`evaluate_expression <expression> (eval) (ee)`

Allows for the evaluation of an expression whose result is immediately output. For example the following prints the starting offset of the `main.dib_header.bpp` field:

```
$main.dib_header$ ee startof(@bpp)
24
```

`dump_field <field1> <field2> (duf)`

Dumps the on-disk bytes interpreted by the field (in the case one field is supplied) or range of fields (in the case two fields are supplied). The dump format is in five columns: the first four columns are the hexidecimal representation of 4 bytes each. The fifth column is the ASCII representation of the first four columns. If a byte fails `std::isgraph` a `.` is subsituted as the glyph in the fifth column.

`dump_offset <start_offset> <end_offset> (duo)`

Same behavior as `dump_field()` but takes a starting and ending offset into the file instead of field(s). The byte at the end offset is included in the dump.

`find_field <name> (ff)`

`find_field` will start from the current node and search down the analysis tree to find any nodes with the `name` passed. If any are found their complete path will be printed out. This makes it easy to find fields that are optional to the file format or may be in one of several locations (e.g., locating the exif metadata within a JPEG.)

BMP Template File Example

The following is a specification for version 1 of the BMP image file format in a template file. Note how expressions inside some of the structures leverage path deduction heuristics in their specifications (`dib_header.width` within `pixel_row_t`).

```
struct header_t
{
    unsigned 8  big    magic_number[2];
    unsigned 32 little file_size;
    unsigned 16 little creator1;
    unsigned 16 little creator2;
    unsigned 32 little bmp_offset;

    invariant ok_magic_number str(@magic_number) == 'BM';
}

struct dib_t
{
    unsigned 32 little header_size;

    invariant ok_header = header_size == 12;

    unsigned 16 little width;
    unsigned 16 little height;
    unsigned 16 little color_plane_count;
    unsigned 16 little bpp;
}

struct pixel_t
{
    unsigned 8 big blue;
    unsigned 8 big green;
    unsigned 8 big red;
}

struct pixel_row_t
{
    pixel_t        pixel_set[dib_header.width];
    unsigned 8 big padding[dib_header.width % 4];
}

struct main
{
    header_t    header;
    dib_t       dib_header;
    pixel_row_t pixel_row_set[dib_header.height];
}
```

UTF-8 Template File Example

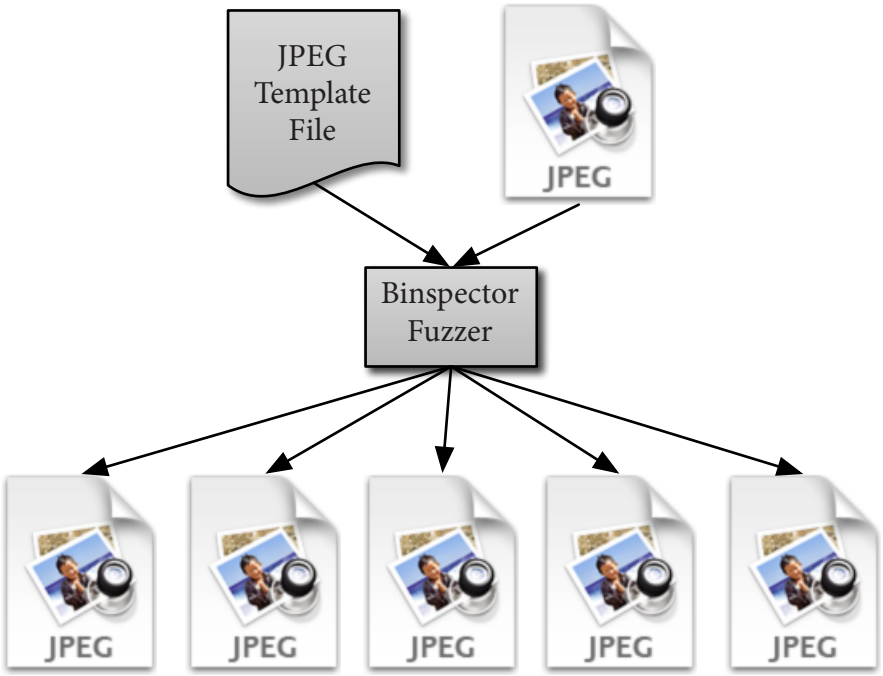
The following is a snippet from a UTF-8 template file specification. The following will correctly analyze 3-byte UTF-8 sequences (the most common when working with multibyte characters). Though not a complete template file for all possible UTF-8 characters it serves as an example of sub-byte field specifications.

```
struct utf8_t
{
    const byte = peek() noprint;

    if ((byte & 0x80) == 0x80)
    {
          if ((byte & 0xE0) == 0xE0)
          {
                // first byte
                unsigned 4 big header;     // should be 1110b
                unsigned 4 big b1p1;       // byte 1, part 1

                // second byte
                unsigned 2 big utf8_cont1; // should be 10b
                unsigned 4 big b1p2;       // byte 1, part 2
                unsigned 2 big b2p1;       // byte 2, part 1

                // third byte
                unsigned 2 big utf8_cont2; // should be 10b
                unsigned 6 big b2p2;       // byte 2, part 2

                invariant valid_header = header == 0xE;
                invariant valid_cont1 = utf8_cont1 == 2;
                invariant valid_cont2 = utf8_cont2 == 2;

                const byte1 = (b1p1 << 4) | b1p2 noprint;
                const byte2 = (b2p1 << 6) | b2p2 noprint;

                // Finally we compose the utf-16 code point
                const utf16 = byte1 << 8 | byte2;
          }
          else
                invariant unhandled_utf8 = false;
    }
    else
    {
          unsigned 8 big char; // 7-bit ASCII character
    }
}

struct main
{
    slot eof = false;

    utf8_t string[while: !eof];
}
```

Intelligent File Fuzzing

Fuzzing is an application hardening technique where corrupted files are opened by an application to see how well it handles unaniticipated data. Fuzzing specifically takes known good documents and changes them either deterministically or randomly in the hopes of exposing weaknesses in file import code.

The Binspector fuzzing engine attempts the former of these two fuzzing approaches. By leveraging template files and a source (known good) binary file, the fuzzing engine will produce a series of documents fuzzed at locations in the file where the import code might be most weak:



The purpose of this section is to give the user a high level overview of the fuzzing engine and what to expect from its use.

Fuzzing Engine Usage

To run the fuzzer from the command line the following argument structure should be used

```
binspector template_file binary_file fuzz
```

The resulting output from Binspector will be in a folder called `fuzzed` and be placed as a sibling to `binary_file`. Within that folder you will find a set of fuzzed binary files and a summary document.

Fuzzed File Details

Each fuzzed file written by the fuzzing engine will generally follow this pattern

```
basename_offset_type{_extra}.extension
```

basename

The basename of the file matches the input file specified. So if your input file was `ducky.jpg`, your basename will be `ducky`.

offset

Each fuzzed document attacks a single possible weakness within the file format. As such the offset specified is the number of bytes into the binary file where the fuzzing began.

type

There are several types of fuzzing attacks. Currently Binspector has four:

Zeroes

The field will be filled with all zeroes. The type identifier for these files is '`z`'.

Ones

The field will be filled with all ones. The type identifier for these files is '`o`'.

Enumerated

Enumerated values are those specified with the `enumerated` construct within the binary file template. When one of these is found by the fuzzer the value will be given each possible enumerated value in lieu of whatever is currently there. The type identifier for these files is '`e`'.

> » *Note:* Assuming the enumerated value in the source input file was valid one of the files output by this method will be identical with the source file. (Broken clocks being right twice a day, and all that.)

Shuffle

Many file types (PNG, TIFF, Exif) use a block-based structure in whole or in part. When described appropriately in a template file these blocks can be reordered arbitrarily by the fuzzing engine. The type identifier for these files is '`s`'.

> » *Note:* This type of attack can generate many files quickly. It is advised to reserve block shuffling to larger-scale constructs (e.g., PNG chunks) and avoid it for smaller ones (e.g., NULL-terminated 8-bit strings.)

> » *Note:* Currently the shuffling algorithm simply shifts the array elements once to the left at every iteration. Far more complex shuffling attacks are planned, however.

extra

In some cases more information is provided about the value used in the fuzzing attack. Currently the only type of fuzzing that provides this information is the enumerated fuzz; what is supplied is the enumerated option used in lieu of the original input value.

extension

This is the same as the extension of the original input source file.

Examples

The following are examples and descriptions of how the engine has fuzzed the document:

```
ducky_12_e_19789.jpg
```

`ducky.jpg` had whatever value lies at offset 12 modified to the enumerated option 19789.

» *Note:* The enumerated option is output in decimal though the value in the binary file template may not be. The extra information is more about making the filename unique than being perfectly descriptive.

```
ducky_1986_o.jpg
```

`ducky.jpg` had whatever value lies at offset 1986 modified to be all ones.

```
sliced_34_s_5.psd
```

`sliced.psd` had whatever contiguous array beginning at offset 34 block-shuffled. This file was produced on the fifth iteration of the block shuffling algorithm.

Summary Document

During the fuzzing process the engine will keep track of all the work it is doing with a summary document output in the `fuzzed` directory. The name of the file will be `basename_fuzzed_summary.txt`, where `basename` is the same value as found in the *Fuzzed File Details* section above.

The contents of this document are a series of fuzz attack summaries like the example below:

```
main.segment[0].app1.exif.tiff_header
    ? use_count : 8
    ? bits : 16
    ? type : unsigned
    ? big_endian : true
    ? enumerated_option_size : 2
    > ducky_12_z.jpg
    > ducky_12_o.jpg
    > ducky_12_e_19789.jpg
    > ducky_12_e_18761.jpg
```

Path

The start of each summary is the path to the location in the file where the fuzzing is taking place. This path is constructed during the source binary file's analysis.

Informative Lines

Lines prefixed with the '`?`' symbol are informative in nature, describing details about the field that is being fuzzed in subsequent files. Most of the informative lines are self explanatory.

attack_type

Currently there are two basic attacks: atom fuzzing and block shuffling. Both attacks are *in-situ* and will not alter the overall size of the document.

`usage`: Atom fuzzing takes a single value and changes it in a random or meaningful way. This is the most common type of attack and is triggered for an atom when it is used in the template to drive futher binary file interpretation.

`shuffle`: Block shuffling takes a contiguous block of data spread over two or more array elements and rearranges them in place. This is triggered by the `shuffle` keyword appended to an array-type field.

use_count

`use_count` is a rough metric used by the fuzzing engine during analysis to find possible weaknesses in the binary file. Higher `use_count` values imply the field is depended upon more heavily within the file's internal structure (and thus using a corrupt value unchecked could do more damage.)

All fuzzing locations will have a `use_count` of at least 1. If a location is never used in the binary file template it is assumed to be a value that does not affect document parsing, and is not fuzzed.

> » *Note:* `use_count` is derived from the binary file template description, and is not necessarily an accurate reflection of how a value is actually depended upon in the file import code.

enumerated_option_size

When a field is an enumerated value the `enumerated_option_size` tells the reader the total number of options available for that enumeration. As such subsequent files will be written, one for each possible enumeration option.

array_size

When a field is specified for shuffling, this value will report the number of elements the shuffling algorithm will be dealing with.

Fuzzed File Lines

Lines prefixed with the '>' symbol denote the names of fuzzed files written to the output directory. These will be named in the format specified earlier in this document.

Fuzzing Engine Errors and Warnings

Lines prefixed with the '!' symbol denote issues that may prevent the fuzzing engine from fuzzing a file completely.