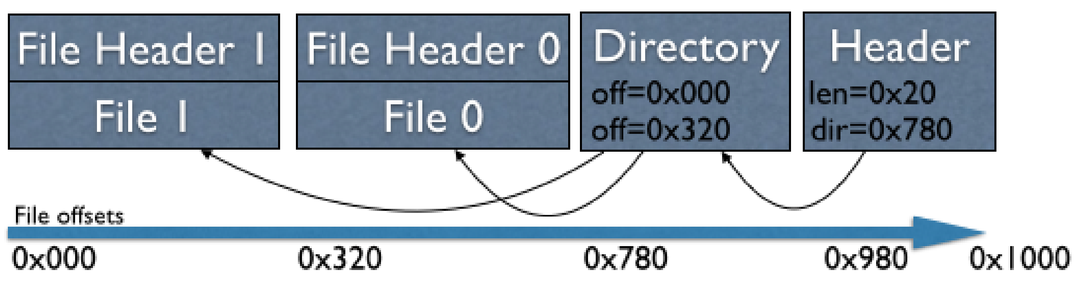
NAIL: A PRACTICAL TOOL FOR PARSING AND GENERATING DATA FORMATS

Julian Bangert and Nickolai Zeldovich
MIT CSAIL

Parsing vulnerabilities

- Hand-written parsers introduce bugs.
- Different parsers not equivalent
  - Evasi0n jailbreaks on iOS.
  - PKI layer cake.
  - Android master key.

ZIP: A tricky file format



Case study of ZIP vulnerabilities in the CVE database:

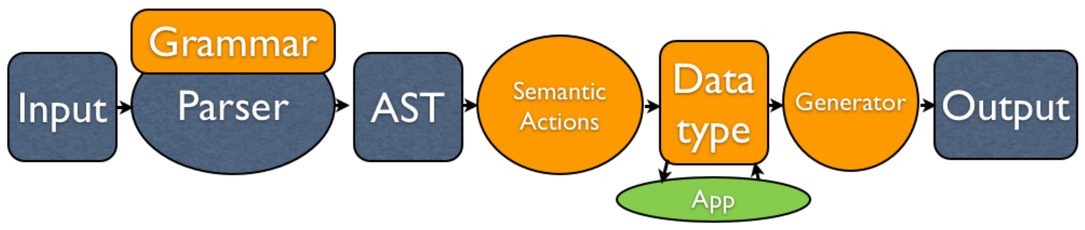
Classification	Example description	Count
Memory corruption	Buffer overflow	11
Parsing inconsistency	Virus scanners interpret ZIP files incorrectly	4
Semantic misunderstanding	Weak cryptography used even if user selects AES	1
Total of all vulnerabilities related to .zip processing		**16**

Existing parsers

Automatically generated parsers, such as `bison` or Hammer are

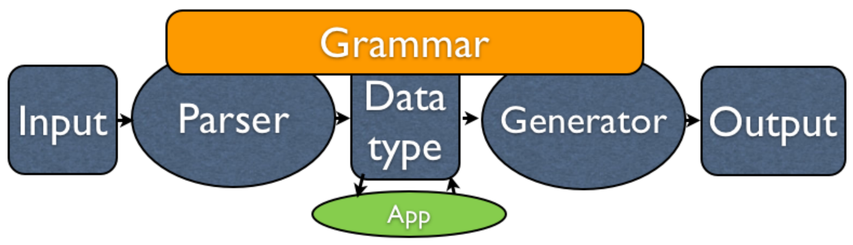
- Immune to classes of bugs (e.g. memory corruption)
- Easier to use
- Re-useable, eliminating parser ambiguities

However, traditional parsers are inconvenient to use, because the programmer has to write all the yellow-colored parts:



Nail

Nail grammars describe both the format and a data type to represent it, an idea introduced by data description languages such as PacketTypes.



Dependent Fields

- Redundant Information in protocol
  - Length field
  - Repeated information
- Confuses application, e.g.:
  - Android master key: Uses local name length for extracting, directory for signature
  - PKI layer cake: Mix implicit (terminator) string length and length field.

Dependent Fields **hide redundant information**and automatically verify and generate it.

Streams and Transformations

- Existing parsers are linear, consuming input front to back.
- Nail grammars feature multiple streams.
- *Transformations* modify and create streams.
- Pair of functions operating on streams and dependent fields.
- Standard library of Transformations enough for many formats.
- Programmer can write their own for complex formats.

Nail Grammar for .ZIP

Nail Grammar

```
zip = {    /*simplified and cut for brevity*/
    /* Call zip_eod transform to isolate end_directory and contents streams*/
     $contents, $end_directory transform zip_eod ($current)
     /* Parse end_directory stream. The end_of_directory parser takes the
      content stream as a parameter */
     contents apply $end_directory end_of_directory($contents)
}
end_of_directory($filestream) = { /*Grammar rules can have parameters*/
     uint32 = 0x06054b50 /* constant*/
     disks uint16 | [0]  /* constraint*/
     directory_disk uint16 | [0]
     @this_records uint16 /*dependent field*/
     @total_records uint16
     /*The following transform ensures these two fields are always equal*/
     transform uint16_depend (@this_records @total_records)
     @directory_size uint32 /*These two dependent fields are used by the  */
     @directory_start uint32/* transformations below to find the directory*/
     /*dirstr1 is the the suffic of filestream starting @directory_start*/
     $dirstr1 transform offset_u32 ($filestream @directory_start)
     /*Another stream with @directory_size bytes starting at that offset*/
     $directory_stream transform size_u32 ($dirstr1 @directory_size)
     @comment_length uint16 /*Dependent field used with the built in n_of*/
     comment n_of @comment_length uint8 /* Variable-length comment*/
     files apply $directory_stream n_of @total_records
        dir_fileheader($filestream) /*Array of directory entries*/
}
dir_fileheader($filestream) = {
     uint32 = 0x02014b50
     @compressed_size uint16
     @crc32 uint32
     @file_name_len uint16
     @off uint32
     filename n_of @file_name_len uint8
     $cstream transform offset_u32 ($filestream @off)
     contents apply $cstream fileentry(@crc32,@compressed_size)
}
```

Excerpt of the generated API

```
struct zip{
  end_of_directory contents;
};
struct end_of_directory{
  uint16_t disks;
  uint16_t directory_disk;/*...*/
  struct {
    dir_fileheader* elem;
    size_t count;
  } files;
};
struct dir_fileheader{
  struct {
    uint8_t*elem;
    size_t count;
  } filename;
  fileentry contents;
};/*..*/
//The programmer calls these generated functions.
int gen_zip(NailArena *tmp_arena,NailStream *out,zip *val);
zip*parse_zip(NailArena *arena,
              const uint8_t *data, size_t size);
//The programmers implements these two transformation
//functions and two similar ones for output.
extern int zip_eod_parse(NailArena *tmp,
 NailStream *filestream, NailStream *,NailStream *current);
extern int zip_compression_parse(NailArena *tmp,
 NailStream *uncomp,NailStream *compressed,uint32_t* size);
```

Evaluation

Protocol	Grammar	Transform	Application	App LoC	Total LoC	Alternative
DNS	48	64	Server	186	298	683 (Hammer)
			Resolver	97	209	>3000 (libc)
ZIP	92	78	Extractor	49	219	1600(Info-ZIP)
Ethernet	16	0				
ARP	10	0				
IP	25	0				
UDP	7	0				
ICMP	5	0				

Nail and our example applications are available at `https://github.com/jbangert/nail`.