RI.
SE
Research Institutes
of Sweden

MOLLY

A tool for dissecting long dead files

Arash Vahidi – arash.vahidi@ri.se 2019-04-15

Abstract

Molly is a general purpose extensible file analysis tool (and library) that can be used to identify, analyze and dissect unknown files. Possible applications include analysis of malicious software, identification of legacy software and extraction of data from exotic archives and other container formats.

Molly was initially developed to analyze legacy firmware images in the SECONDS project. This documents applies to Molly version 0.2.4.

1 Background and motivation

Connected embedded systems are becoming more common, which has unfortunately resulted in a new class of possibly insecure devices. Given the huge numbers and diversity of such devices, security researchers have turned to automated tools for vulnerability analysis. Some notable attempts in large-scale automated vulnerability analysis are [4, 7, 3, 2].

A first obstacle such tools may encounter is extracting software components from the device, or more commonly from firmware images published by the vendors. Firmware images for embedded systems are often container formats that include components such as kernels, bootloaders and various types of filesystems. These filesystems are themselves container formats for binaries and data files. Sometimes the container formats are proprietary or undocumented variation of standard formats. Hence an important role of a file analysis tools may be to understand and extract data from such container formats.

There exist a number of tools for analysis and extraction of firmware. Molly is one such tool.

1.1 Existing tools

The *file* utility from UNIX operating systems [8] is one of the earliest file analysis software still being used today. It attempts to identify a file using three tests (filesystem, magic database and language). It is able to identify simple classes of files such as images and documents.

```
$ file main.tex Makefile biblio.bib main.pdf
main.tex: LaTeX 2e document, ASCII text, with very long lines
Makefile: makefile script, ASCII text
biblio.bib: ASCII text, with very long lines
main.pdf: PDF document, version 1.5
```

Binwalk is a more recent tool that uses an extension of the magic format database used by *file*. It is able to identify many binaries and container formats (e.g. file systems and firmware images) using a fairly large magic database [5]. An example of a magic file is seen below:

```
# Sample magic pattern for a Microsoft executable.
# Note the "(0x3c.l)" which is a 32-bit pointer reference.
0        string MZ\0\0\0\0\0\0 Microsoft executable,
>0x3c    lelong <4            {invalid}
>(0x3c.l) string !PE\0\0      MS-DOS
>(0x3c.l) string PE\0\0       portable (PE)
```

YARA is another tool commonly used for large scale binary analysis [1]. It uses a rule database containing textual or binary patterns to identify classes of files. As seen below, YARA rules are human-readable while magic rules are mainly formatted for machine interpretation.

```
rule maybeexecutable {
   strings:
      $mz="MZ"
   condition:
      ($mz at 0)
}
```

These tools also provide a number of options for integration in a automated system and various methods to extend their core functionality. For example modern versions of *file* provide the *libmagic* library [9] and *binwalk* and *YARA* both provide APIs for adding custom functionality.

It should be mentioned that a virus scanner is normally not seen as a generic binary analysis tool but a highly specialized data analysis software. However, some virus scanners such as the Clam anti virus build upon a general purpose file analysis engine [6].

1.2 Why a new tool?

Molly improves on existing tools in a number of areas:

- Molly has a simple human-readable rule format that resembles C / Java and makes definition of chains of pointers and conditions easier and more efficient,
- Molly has a powerful plugin system,
- Molly can generate machine-readable output (JSON),
- Molly can run on resource-constrained devices,
- Molly was designed to handle possibly malicious binaries.

At the same time, it does have a number of disadvantages:

- Molly does not currently have a comprehensive rule database,
- Molly cannot always be used to look into completely new binary formats files.

As such, we recommend using binwalk and Molly together, the former for the initial analysis and the latter for large-scale identification and extraction.

1.3 About the name

Molly was named after Molly Hooper, the pathologist in the BBC TV-series Sherlock. This seemed appropriate for a software used to dissect long dead binaries.

In North America Molly may have a different meaning to you. Fortunately, the author lives in Europe where the grass is green and nothing else.

2 Introduction to Molly

As an example, consider the ELF file format in Figure 1. Given this information we could define rules to, for example, identify FreeBSD binaries:

- The file must start with the correct four ”magic” bytes,
- The byte at offset 7 (OS ABI) must contain 9.

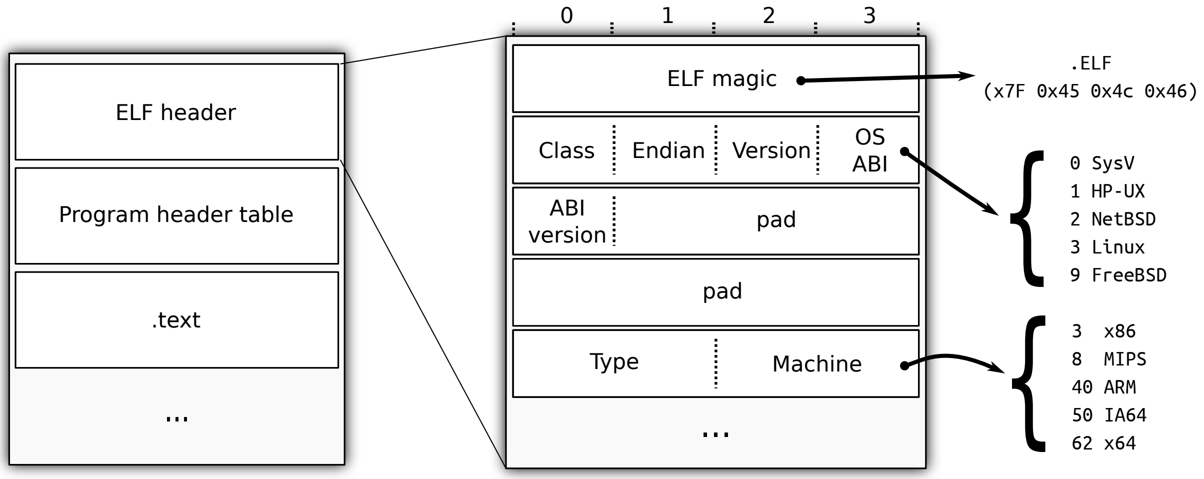

Figure 1: The ELF file format.

This information can be written in a more formal format:

```
// elfbsd.rule
rule bsdelf {
    // variables
    var magic = String(0, 4);
    var osabi = Byte(7);

    // conditions
    if magic == {0x7f, 0x45, 0x4c, 0x46};
    if osabi == 0x09;
}
```

Using this rule Molly can identify FreeBSD binaries in a heap of files [1]:

```
$ molly -R bsdelf.rule -o output files/
SCAN RESULTS:
    * File files/libavl.so.2 (0 errors):
        => bsdelf
    * File files/cat (0 errors):
        => bsdelf
    ...
```

According to this report two files in the ”files/” folder matched our ”bsdelf” rule. This information is also available in a machine-readable format in the output directory:

```
$ cat output/match.json
{
    "matches": {
```

[1] Since Molly has a number of built-in rules, you might see some more matches. To exclude them use ”-p config.standardrules=false”.

```
        "files/cat": [ "bsdelf" ],
        "files/libavl.so.2": [ "bsdelf"]
    }
    ...
}
```

Furthermore, a detailed report is created for files with at least one match. These reports will mirror the original file name and have a ”_molly.json” suffix:

```
cat output/files/libavl.so.2_molly.json
{
    "depth": 0,
    "filename": "files/libavl.so.2",
    "information": {
        "checksum": "fa8b0b087b035e0937fe9358a93b20d502203b24def5b8316e717b0fdd648b43"
    },
    "matches": [
        {
            "Name": "bsdelf",
            "Vars": {
                "magic": "ELF",
                "osabi": 9,
            }
        }
    ],
    ...
}
```

Molly also keeps track of file hierarchies. For example if the above file was initially stored in files/container.zip, the following would appear instead:

```
$ cat output/files/container.zip_/libavl.so.2_molly.json
{
    "depth": 1,
    "parent": "files/container.zip",
    "filename": "output/files/container.zip_/libavl.so.2",
    ...
}
```

2.1 Performing actions

You may want to automatically perform an action once a match has been found. From the command line this is done using ”-on-rule” parameter as demonstrated below:

```
$ molly -R bsdelf.rule -o output \
    -on-rule "bsdelf:echo Found {filename} with {filesize} bytes" files
...
RULE bsdelf on files/libavl.so.2: Found files/libavl.so.2 with 14896 bytes
RULE bsdelf on files/cat: Found files/cat with 23648 bytes
...
```

It is also possible to use operators that perform an action from within the rule. This method has the benefit of having access to internal Molly data such as rule variables.

For example, assume we want to look at ELF headers for each detected ELF file. This is done with the *system* operator [2].

```
rule bsdelf {
    ...
    system("objdump -h %s", $filename);
}
```

Note that when successful this rule will not generate any output. A quick fix would be to capture *stdout* and use a second action operator to show it:

```
    var output = system("objdump -h %s", $filename);
    printf("ELF HEADERS:\n%s\n", output);
```

Molly provides a rich set of action operators including complex data analyzers and extractors. Refer to section 5 for a comprehensive list.

3 Command-line and API usage

The command-line format is:

```
Molly [OPTIONS] files
```

Where *files* are zero or more files (or directories) to be analyzed. Options are

Option	Description
-R rule file	Rule file (or directory) to use
-r rule text	In-line rule to use
-o dir	output directory (default ”output”)
-p param=value	set Molly parameter
-on-rule rule:command	command to run when a rule match is found
-on-tag tag:command	command to run when a tag match is found
-version	Show version number and exit
-h	Help information
-H	Extended help information

Available parameters are:

Name	Default value	Description
config.maxdepth	12	max scan depth
config.standardrules	true	load standard rule library
perm.create-file	true	allow Molly to create new files
perm.execute	false	allow Molly to execute external tools
option.verbose	false	be verbose

Molly comes with a small set of standard rules which can be excluded by setting *config.standardrules* to *false*.

The *-on-rule* and *-on-tag* parameters allows execution of external commands when a rule or tag match is seen:

[2] Since *system* is a potentially dangerous operator, it must be enabled with the command.line option ”-p perm.execute=true”

```
$ touch file1
$ molly -r "rule any{ }" -on-rule "any:ls -l {filename}" file1
RULE any on file1: -rw-rw-r-- 1 mh mh 6 mar 0 13:55 file1s
$ echo hello > file1
$ molly -r "rule any (tag = \"text\") { }" -on-tag "text: cat {filename}" file1
TAG text on file1: hello
```

Note that *{filename}* is an environment variable (see Section 4.1).

3.1 Using the library

The command-line tool is one of the two methods for using Molly. The other is embedding Molly in your own Go application, a simple example is shown below:

```
package main

import (
    "log"
    "bitbucket.org/vahidi/molly"
)

func main() {
    m := molly.New()
    m.Config.OutDir = "/tmp/Molly"

    if err := molly.LoadRules(m, "myrule.rule"); err != nil {
        log.Fatal(err)
    }

    if err := molly.ScanFiles(m, "unknownfile.bin"); err != nil {
        log.Fatal(err)
    }

    report := molly.ExtractReport(m)
    for _, f := range report.Files {
        log.Printf("%s, %d bytes. %d errors, %d matches\n",
            f.Filename, f.Filesize, len(f.Errors), len(f.Matches))
    }
}
```

An important reason for using Molly in this fashion is the ability to add custom functionality.

4 Writing complex rules

The rule format has a few interesting traits for more advanced users. These are discussed in the following.

4.1 Environment variables

Environment variables hold additional data from the environment including the target file. In a rule, such variables have a ”$” prefix:

```
rule biggofile {
    if $filesize > 4096;
    if $ext == ".go";
    printf("%s is one big Go file...\n", $filename);
}
```

To avoid confusion with shell environment variables curly brackets are instead used on the command-line:

```
$ molly -r 'rule cfiles { if $ext == ".c"; }' -on-rule "cfiles:gcc {filename} -o {newfile:
     compiled.o}" src/
```

The following environment variables are available with rules:

Variable	Type	Description
filename	string	Name of the currently processed file, e.g. */dir/file.c”*
shortname	string	Short file name, e.g. *”file.c”*
basename	string	Base file name, e.g. *”file”*
ext	string	File extension, e.g. *”.c”*
dirname	string	File directory, e.g. *”/dir”*
filesize	int64	File size
parent	string	Name of parent file (or ”” if root)
depth	int	Depth of file in extraction tree, 0 if root
num_matches	int	number of matches for this file so far
num_errors	int	number of errors encountered for this file so far
num_logs	int	number of logs generated for this file so far
newfile[:suggestedname]	string	produce new file (see note)
newdir[:suggestedname]	string	produce new directory (see note)

Note that newfile/newdir are only available on the command-line only. Furthermore, the files and folder created with these are fed back into Molly for analysis.

4.2 Metadata

Additional configuration parameters in rules are presented as metadata:

Metadata	Type	Default value	Usage	Description
tag	string		rules	associate rule with a tag
bigendian	boolean	true	rules / operators	specify endianness
pass	number 0-2	0	rules	scanner pass

Metadata must always be a static value. In the example below, *bigendian* and *tag* are metadata.

```
rule example (tag ="test", bigendian = false ) {
    var a = Long(0); // little endian, by rule metadata
    var b = Long(0, bigendian = true); // bigendian, overides rule
}
```

The rules are checked in one of three passes (pass 0 is the default). This allows rules to take action based on previous matches. For example the following rule will invoke binwalk as a last resort if no matches have been found by molly so far:

```
rule unknown (pass = 1) {
    // NOTE: condition to detect endless-loop in binwalk omitted for brevity!

    if $num_matches == 0; // nothing found by previous rules
    var d = dir("from_binwalk"); // store results here
    system("binwalk -C %s -d 4 -e -M %s", d, $filename);
}
```

4.3 Hierarchical rules

Defining related rules as a hierarchy can lead to shorter and simpler rule definitions while minimizing chance of mistakes and at the same time improving performance. In the following example multiple ELF architectures are covered by use of rule hierarchies:

```
rule ELF {
    var magic = String(0, 4);
    var class = Byte(4);
    var data = Byte(5);
    var version = Byte(6);
    var osabi = Byte(7);

    if magic== "\x7FELF" && (version == 1) &&
        (class == 1 || class == 2) && (data == 1 || data == 2);
}

rule ELF_be (bigendian = true) : ELF {
    if data == 2;
}

rule ELF_le (bigendian = false) : ELF {
    var machine = Short(18);
    if data == 1;
}

rule ELF_x86 : ELF_le {
    if machine == 0x0003;
}

rule ELF_arm64 : ELF_le {
    if machine == 0x00B7;
}
```

Metadata is also inherited, hence in this example both ELF_x86 and ELF_arm64 are little-endians.

5 Operators

Operators are functions that can be called within rules. The most common operators are the primitive operators for reading data. Other operators mainly operate on existing variables or the whole file.

Operator			Description
		Primitive operations	
uint8	**Byte**	(offset int)	read 1 byte from offset
uint16	**Short**	(offset int)	read 2 bytes
uint32	**Long**	(offset int)	read 4 bytes
uint64	**Quad**	(offset int)	read 8 bytes
string	**String**	(offset, size int)	read a byte octet from given offset
string	**StringZ**	(offset, maxsize int)	read a zero-terminated string with given max size
		String operations	
bool	**stricmp**	(string, string)	string compare, ignore case
bool	**strstr**	(string, string)	find string in text
bool	**strcasestr**	(string, string)	same as strstr but case-insensitive
bool	**strsuffix**	(string, string)	check if text ends with some string
bool	**strprefix**	(string, string)	check if text starts with some string
int	**strlen**	(string)	string length
int64	**strtol**	(string)	convert string to number
string	**strupper**	(string)	upper-case string
string	**strlower**	(string)	lower-case string
		Formatting	
string	**printf**	(string, ...any)	standard printf to stdout (Go syntax)
string	**sprintf**	(string, ...any)	standard printf to string (Go syntax)
		New Input	
string	**dir**	(string)	create a new directory
string	**file**	(string)	create a new file
		Miscellaneous	
[]uint8	**checksum**	(type string, ...uint64)	Checksum file or slice
int	**len**	(any)	Return length of item
string	**epoch2time**	(int64)	Convert UNIX epoch to a date string
bool	**has**	(type string, string)	target has the following data or metadata
		Actions	
string	**system**	(command string, ...any)	execute shell commands
string	**analyze**	(format, file string, ...any)	Perform analysis on file
string	**extract**	(format, file string, ...uint64)	Extract file or file slice

Shell commands

The *system* operator executes a shell command. It uses the same formatting syntax as printf/sprintf:

```
rule squashfs {
    ...
    var dir = dir("unpacked_stuff");
    system("unsquashfs -n -no -f -d %s %s", dir, $filename);
}
```

Executing shell commands is regarded a dangerous operation and should be avoided. A better solution is to enhance Molly with your own operators, analyzers and extractors. To protect the user from harm, the system action is disabled and must be enabled using the parameter ”-p perm.execute=true”.

5.0.1 Custom operators

To simplest way to extend Molly is to add a custom operator. These can then be called from rules to perform complicated tasks. For example, the following code implements AES code block decryption:

```
import (
    "fmt"
    "crypto/aes"
    "bitbucket.org/vahidi/molly/types"
    "bitbucket.org/vahidi/molly/operators"
    "bitbucket.org/vahidi/molly"
)

// Signature must match func(e *types.Env, args ...interface{}) (interface{}, error)
// Molly will attempt to convert between similar types but this may not always work
func decrypt(e *types.Env, key, data []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    bs := block.BlockSize()
    if len(data) % bs != 0 {
        return nil, fmt.Errorf("AES data must be padded to block size %d", bs)
    }

    dst := make([]byte, len(data))
    for i := 0; i < len(data); i += bs {
        block.Decrypt(dst[i:i+bs], data[i:i+bs])
    }
    return dst, nil
}

func main() {
    operators.Register("decrypt", decrypt); // must happen before loading rules
    m := molly.New()
    ...
}
```

After registration the new operator can be used in rules:

```
rule example {
    var key = String(0, 32);             // AES-256
    var ciphertext = String(32, 32);     // two blocks
    var cleartext = decrypt(key, ciphertext);
    ...
}
```

In addition, Molly provides API to add custom checksum functions, analyzers and extractors.

Checksums

The *checksum* operator computes a checksum or hash over a range of bytes. For example

```
rule example {
    ...
    checksum("sha256", 0, 1024); /* SHA-256 for the first KB */
}
```

It currently supports the following types: sha256, sha128, sha1, md5, crc32, crc32-ieee, crc32-castagnoli, crc32-koopman, crc64, crc64-iso, crc64-ecma. These can be further extended via the checksum API:

```
import (
    "crypto/md5"
    "hash"
    ...
)

func mysuperhash() hash.Hash {
    return md5.New
}

func main() {
    operators.RegisterChecksumFunction("mysuperhash", mysuperhash)
    m := molly.New()
    ...
}
```

Analyzers

The *analyze* operator performs some type of analysis on the current file. For example

```
rule DalvikDex (bigendian = false) {
    ...
    analyze("dex", "my-dex-analysis");
}
```

The outcome of this analysis will be found in the generated report:

```
$ output/classes.dex_molly.json
...
   "information": {
      "checksum": "128dc3a70d699aee229fe596de4471ca0ed88d4dc4871c4b60ca70e793a687bb",
      "my-dex-analysis": {
         "callmap-class": {
            "a.a.a.a.a": [ "java.nio.charset.Charset.forName"],
            ...
}
```

Currently the following analyzers are supported:

Type	Description
strings	String extraction
version	Version extraction from strings
histogram	Generate byte histogram
elf	ELF analyzer
dex	Android DEX analyzer

Additional analyzers can be added using the API:

```
// Signature should be func(string, io.ReadSeeker, ...interface{}) (interface{}, error)
// By convention, analyzers return a dictionary but can technically return anything.
func entropy(filename string, r io.ReadSeeker, blocksize int) (interface{}, error) {
    var ret []float64

    ... /* compute Shannon's entropy on blocks of <blocksize> bytes */

    return map[string]interface{} { "entropy": ret }, nil
}

func main() {
    operators.AnalyzerRegister("entropy", entropy)
    m := molly.New()
    ...
}
```

The new operator can be used in rules:

```
rule example {
    ...
    analyze("entropy", "myentryopyanalisys", 1024);
}
```

Extractors

The *extract* operator extracts data from the target file. For example

```
rule jffs2 (tag = "filesystem", bigendian = false) {
    ...
   extract("jffs2", "jffs2");
}
```

The currently supported formats are binary, tar, MBR, cramfs, JFFS2, zip, gz, CPIO and uImage.

The binary extractor can also operate on file *slices*. In this context a file *slice* is a subset of a file and is defined by the pair $(offset, length)$.

For example, the following will extract bytes 10, 11, 20 and 21:

```
    extract("binary", "mydata", 10, 2, 20, 2);
```

Support for new formats can be added using the extractor API:

```
func unpacker(e *types.Env, prefix string) (string, error) {
    inputfile := e.GetFile()
    r := e.Reader
    w, _, err := e.Create(prefix + inputfile + "_unpacked)
    // using e.Create() will ensure that the resulting file has a sane name and is fed
    // back into molly for analysis later

    ... /* unpacking from r to w using some custom algorithm */

    return "", nil
}

func main() {
    operators.ExtractorRegister("unpacker", unpacker)
    m := molly.New()
    ...
}
```

References

[1] Victor M. Alvarez. Yara, malware identification and classification, 2008.

[2] D.A. Andriesse, J.M. Slowinska, and H.J. Bos. *Compiler-Agnostic Function Detection in Binaries*, pages 177–189. Institute of Electrical and Electronics Engineers, Inc., 6 2017.

[3] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *The Network and Distributed System Security Symposium (NDSS 2016)*, 02 2016.

[4] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, New York, NY, USA, 2016. ACM.

[5] Craig Heffner. The binwalk firmware analysis tool, 2010.

[6] Tomasz Kojm. Clamav, an open source antivirus engine for detecting trojans, viruses, malware & other malicious threats.

[7] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 709–724, Washington, DC, USA, 2015. IEEE Computer Society.

[8] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17:365–375, 1974.

[9] Christos Zoulas. file-4.00 is now available, 2003.