Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Ivor Kollár

Forensic RAM dump image analyser

Department of Software Engineering

Supervisor: Mgr. Viliam Holub, Ph.D., Department of Software Engineering, Faculty of Mathematical and Physics
Study program: Computer Science

2009

I would like to thank Martin Mareš and Pavel Kaňkovský for help with understanding i386 architecture and Linux kernel structures.

Contents

Title: Forensic RAM dump image analyser
Author: Ivor Kollár
Department: Department of Software Engineering, Faculty of Mathematical and Physics
Supervisor: Mgr. Viliam Holub, Ph.D.
Supervisor's e-mail address: holub@dsrg.mff.cuni.cz

Abstract: While different techniques are used for physical memory dumping, most of them provide a hard-to-analyse image of raw data. The aim of the work is to develop an automatic analyser of physical memory dumps retrieving contained information in a user-friendly form. The analyser is supposed to simplify automatic data extraction and should be used by forensic experts. Among expected features are multiple target architecture/OS support, target architecture/OS guessing, automated password/crypto keys collecting, process listing, and module/driver listing.
Keywords: forensic, analyse, pentest, RAM, dump, key, memory, recovery

Chapter 1

Introduction

1.1 Problem overview

Suppose, that there exist a method for retrieving content of whole physical RAM of running computer. Output of this method are Mega/Gigabytes of raw data. The most simple approach when trying to analyse this data is to use string-like utilities, to use pattern matching. This method is relatively reliable, but also slow with bigger images. Using pattern matching for some tasks is quite hard, for example listing running processes of OS (operation system). If exact version of OS is publicly known, the patterns can be made to match this exact version, and program can use special pattern for each version of OS. However this cannot be used, when analysing for example self compiled Linux kernel. In this work, some alternative approaches are being studied and implemented.

1.2 Assumed usage

Foriana is designed to perform fast analysis of the dump of physical RAM, and extract interesting information. As most common use-cases are assumed forensic analysis and penetration testing.

1.2.1 Forensic analysis

Forensic analysis tries to extract as much information as possible from object of interest. In computer science, forensic analysis is usually categorized as "in vivo", which can be translated as analysis of running and responding system and ”post mortem”, which can be translated as analysis of previously gathered data, that does not change. Assumed usage of Foriana is somewhere between this two categories: Retrieving RAM content is typically performed in vivo, while analysis itself is post mortem.

Content of RAM is an important piece of information, when trying to analyse previous activity on the machine. RAM can contain pieces of processes, deleted files, user sessions, crypto keys... With advancing information security awareness and wide deployment of encrypted filesystems, retrieving decryption keys is becoming a ”key” task. RAM is often the only place, where keys are kept.

1.2.2 Penetration testing

The goal of penetration testing is to simulate potential attacker and gain as much control of the target system as possible. After the test, all weaknesses exploited during test can be fixed, resulting in a more secure system.

With physical access to target computer, few methods for direct manipulation with RAM are well known (discussed later). Let’s assume, that we master one of this methods, and have Read/Write access to RAM (physical memory). If operating system of target is not exactly known, one may need some semi-automatic tool, that determines target OS, version, and output critical addresses, that can be parsed to memory modifying scripts. (To kill screensaver, turn off firewall, spawn portshell...)

Since memory can change without warning, while analysing image, analysis has to be completed quickly.

1.3 Physical RAM

RAM (Random-access memory) is a form of computer data storage. Today, it takes the form of integrated circuits that allow stored data to be accessed in any order (i.e., at random). The word random thus refers to the fact that any piece of data can be returned in a constant time. In this article under the word RAM is understood volatile type of memory, where the information is lost after the power is switched off. As explained in [8], time between power loss and data loss can be long enough to capture content of memory. Term ”Physical RAM” is used, because ”RAM” is used very often in computer science, and usually refer to linear mapped memory.

Chapter 2

Existing documents and tools

2.1 Image extraction

Several methods for retrieving content of RAM have been described. If possible, investigator should prefer hardware methods, because any manipulation with the target system can lead to information loss. Hardware methods also often allow to bypass OS security mechanisms like locked screensaver, etc.

2.1.1 OS independent

There are two main methods requiring physical access known to the public. First one based upon DMA (Direct Memory Access), and second based upon hardware properties of commonly used RAM modules. DMA access can be used, because i386 architecture trust every device connected to PCI BUS. Fireware (or IEEE 1394) is special case of such device, because device connected over firewire cable is still on PCI BUS of computer. This is a design feature.

- Method 1: Special card

  "Feature" of DMA access over PCI bus can be exploited with special devices, developed to help police investigators. Several types can be found on market today, but usually they are sold only to military/police agencies or officers. Cardbus, PCI and AGP connectors was seen on market. Advantages of these devices are professional support and reliability. They can be used by technically inexperienced people. On the other hand, they are expensive, and their availability is limited. Because dumping takes some time, there can appear race conditions in gathered memory. Also, theoretical defence against this type of attack was described in [11], allowing target system to crash before RAM is copied, or even fake the results.

- Method 2: Firewire

  This is special case of previous method. The difference is, that firewire cable can be easily purchased by anyone, same as portable notebook with firewire connector. Adam Boileau (alias "metlstorm") wrote a python tool for dumping physical RAM via firewire, `1394memimage` [13]. Later, he also released tool for taking control over computer running Windows operating system, `winlockpwn` [14]. `1394memimage` is free, and an excellent for capturing RAM of a target computer, which can be analysed afterwards.

  This is probably the easiest and the cheapest method to use. To dump some operating systems, such as MS Windows, device header have to be faked, but this is also a simple operation. Please note, that target system can be easily crashed by reading not mapped memory, or specially mapped memory. 1394memimage tries to cover such situations, but is not always successful. Because this method is somewhat slower, compared to special card, there can be more race conditions in gathered memory.

- Method 3: "cold boot" attack

  From forensic point, this is most valuable and reliable approach, but may be difficult to perform in real situations. RAM is quickly physically frozen, which makes information in RAM persistent for longer period of time even without electric power. More information can be found in Cold Boot Attacks paper [8]. Advantages of this method are perfect RAM image (no race conditions), difficult defence and universal approach to all investigated computers. On the other hand investigator needs sophisticated equipment. If not copied quick enough, RAM can contains a lot of random errors.

2.1.2 Windows

To obtain memory image of running Windows OS, `memdd` [15] can be used, or other specialized windows tools . To perform dump of OS memory, system user privileges are required. Alternatively, system hibernation file can be used, if available and user has used hibernation function during his work.

2.1.3 Linux

On running Linux system, the easiest way to collect memory dump is to execute command

```
#dd if=/dev/mem of=outputfile
```

with root privileges. This is "quick and dirty" method, because `/dev/mem` will not read more than 1GB of data. In some systems, such as Redhat/Fedora distributions, implementation of `/dev/mem` device is limited even more, allowing access only to first 1MB of RAM. On the other hand, dd command is present in almost every Linux distribution, and can be used when sophisticated tools are not available to perform quick dump.

A more sophisticated solution would be to write special dumping module which when loaded would create virtual character device similar to `/dev/mem`, but with no limitations.

Similar to the windows platform, depending on system configuration, swap partition can contain rests of hibernated RAM. This is the worst quality image, but may be the only available.

2.2 Image analysis

2.2.1 Windows

Andreas Schuster wrote `ptfinder` for finding processes in memory images of Windows OS. `ptfinder` is based on basic pattern matching. Tool can be found at `http://computer.forensikblog.de/files/ptfinder/`.

`The Volatility Framework` is "completely open collection of tools, implemented in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory (RAM) samples" [7]. At time of writing this article, Linux support was not implemented, or at least not publicly released. Framework is still under development and have the best perspective from available project.

There is also plenty of small "one-time" scripts and programs for windows memory analysis, but without being general enough, and without further support, so the are not mentioned in the text.

2.2.2 Linux

Mariusz Burdach wrote three simple pattern matching based tools: `procenum`, `pfenum` and `taskenum`. They are all included inside `idetect` toolkit. Toolkit can be found at `http://forensic.seccure.net/tools/idetect.tar.gz`

However, it seems that these tools are not working on 2.6 kernels, at least the author of this article was not able use them successfully. `Idetect` toolkit is probably no more maintained.

In 2009 product `SecondLook` from pikewerks company was introduced. Program is commercial, not freely available, so exact capabilities remain unknown. From screenshots it is evident, that parsing of the source code, debugger and GUI are included.

Chapter 3

Methods for finding structures of partially known OS

3.1 Problem definition

Let's have RAM image of some not exactly known operating system. Under "not exactly known" we understood system, where main branch is known, for example Linux, or Windows, but exact version not. System structures between small releases can change, but should not change completely. Such case would be considered to be a different main branch. Let's also assume, that we do not necessarily have source code for exact version of operating system, and that there can be waste amount of different small versions of OS.

In real world, such situation can be recognized in Linux kernels: Everyone can compile his own kernel with custom patches. Development of new versions is relatively fast, so there are many different versions in use.

3.2 Architecture specific structures

First of all, we should focus on architecture specific structures. Structures such as PaGe Directory (PGD) can be found using pattern matching, and because the number of different architectures is quite limited, it is possible to write appropriate filter for each architecture. Mastering linear-physical address translation is a big advantage, if not necessity for later analyse of image. It allows us to follow linear pointers inside image almost safely.

3.3 Pattern matching

Pattern matching can be successfully used for finding architecture specific structures, but the use of pattern matching for later analysis is limited. We can search for certain information, like process or module name, that should be present in target system, but we cannot search for structures themselves, because at the time of writing our program, we do not know, how will they look like. Therefore we need to look for alternative method of finding system structures, one of them is described below. To be complete, pattern matching can be used again later, once determination process is completed, and we have patterns of analysed structures.

3.4 "The Longest-Nearest" algorithm

This algorithm was developed while looking for process list on Linux kernel, but seems to be generic enough, to work in other operating systems and architectures and with different type of structures in list.

Algorithm prerequisites:

- 1. Structures we are looking for are in double linked list. This is the most common structure, than cannot be simplified (too much).
- 2. We know "name" of at least 1 member of list. By "name" we mean any string long enough to be fulltext searched without too many false positives.
- 3. We are able to follow the pointers found in memory/dump. On i386 architecture this means that we have mapping from linear to physical memory.
- 4. Algorithm result is not guaranteed, to provide reasonable results, no part of list can be completely separated from other (at least one pointer must point to this sublist of structures)

Algorithm description:

- 1. Full-text search for name of one of list members. This search is looped until we find what we are looking for, or reach end of dump/memory.

- 2. Each time name is found, make check if this is one of the structures from list we are looking for. Check is done by following algorithm:

  Look constant1 bytes forward/backwards from found strings, and test if there is `list_head` structure. This is very simple structure, containing only 2 pointers: One to previous member of list, second to next member of list. Or reversed, but this is not important.

  Testing if we are looking at `list_head` is simple: Follow pointer we are at, and test if the address pointer points to is another pointer, that points back. The same check is done for 1 pointer under. If both pointers satisfy this condition, we are really looking on `list_head`, at least at syntactically correct one. This is visualised in figure 3.1.

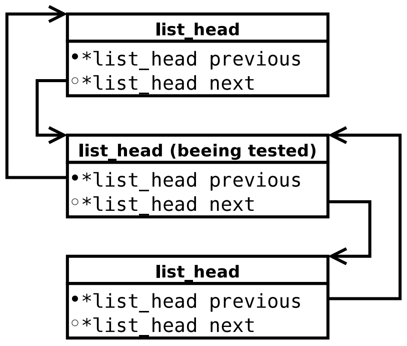

Figure 3.1: Testing list_head

  In same way we try at most constant2 `list_heads`. For each `list_head` compute its length. The longest of `list_head` found, should be `list_head` we are looking for. Checking of length is needed to prevent false positives. There can be (and usually are) many `list_heads` inside structure, for example lists of children or threads in Linux `task_struct`.

Simplified visualisation is in figure 3.2.

Another two "inputs" to the algorithm are constant1 and constant2, these have to be determined by hand for concrete applications. For real world operating systems, values around 100 for constant1 and 10 for constant2 seem produce enough reliable results.

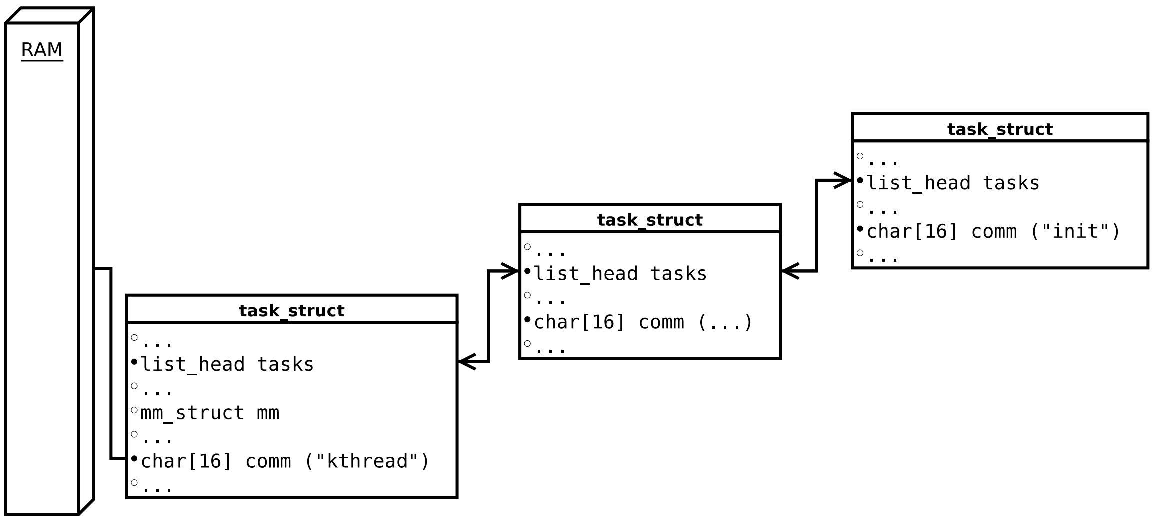

Figure 3.2: The Longest-Nearest

3.5 Generating dynamic patterns

Using ”The Longest-Nearest” algorithm allows analysis of image at logical level, identifying structures through relations between them. This is great for general overview of the image content, but investigator still have to manually inspect found structures and make patterns, which can be used for pattern-matching. Pattern matching cannot be omitted, as finished processes (files, modules) are no longer in any consistent data structure.

Whole process can be automated, if we allow certain level of errors in patterns. This condition can look simple, but may turn pattern-matching into non-trivial task, if we want to preserve high speed of search.

Automated patter generation can be done in these steps:

- Use ”The Longest-Nearest” algorithm (or any else) to determine offsets. Then get list of structures for which patterns should be generated.
- At the beginning pattern is represented by first structure in list.
- Proceed whole list replacing all bytes in pattern that are not same in all structures with wildcard/regexp for any character.
- At the end, pattern that match all the structures in list is produced.

Error tolerance has to be added, as some bit/bytes present in all structures of list, for example `state` (-1 unrunnable, 0 runnable, ¿0 stopped in 2.6 kernel) will be different in ”dead” processes. And automat is not not able to discover, which bytes are important and which not.

3.6 Parsing source code

During Foriana development, an alternative approach to problem was tried. Linux kernel have publicly available source code. If exact version of source code and configuration options used for kernel compilation are known, source code can be parsed to fit structures in memory. This would allow to skip almost whole determination process (after finding VPT), a provide much better results. After considering this solution, I decided that Foriana can be used for linear to physical memory translation and rest can be performed by standard tools (compiler and debugger). Writing these tools again would have no sense, same as writing GUI for them.

Chapter 4

Foriana

Nowadays, almost all tools use classic pattern-matching approach. They remember signatures of certain system structures, usually for each version of system a slightly different signature. Then scan whole image looking for signatures, declaring result of this operation as process(module, file descriptors, ..) list.

This is perfect method when looking for ”lost” information, like parts of deleted files, rootkit hiding somewhere inside the memory, etc... Probably the best choice for forensic expert.

But using this approach is relatively slow, can produce many false positives, (for example if user on computer investigated was reading about kernel structures;) and mainly, they are extremely dependent on version of operating system.

This can (and is) working well for MS Windows, because there are few versions of the operating system, and all are known to public. (Someone may object, that these systems are widely used, but internal structures are not public. In reality, vendors EULAs are ignored, and internal structures were reverse engineered)

On the other hand, in Linux/Unix world, there are many distributions, kernel is changing quite often, and people even compile kernel by themselves, so simple pattern-matching approach is not working very well.

Foriana tries to solve problem in more general way, do not insist on exact structure signatures, but finding a logical relationships, looking for lists of element defined more freely.

This can allow us to hope, that algorithm will not fail even on newer versions of operating systems, not known at moment of program compilation.

The exact process is described in next chapters.

4.1 Architecture and OS guessing

At this moment Foriana supports only i386 architecture, so program tries to process image as being i386. This is done by looking for VPT. If VPT is found (or is supplied by user) architecture is i386, otherwise processing fail.

In the future, guessing based probably on interrupt vectors will be introduced.

Operation system type and version guessing is a black magic, and results should be inspected manually. Used method is searching for strings like "linux-2.6.22" or "Linux version 2.6.21.5" in first pages of memory. However this is far from producing reliable results. From obvious reasons, guessing will probably fail for "future" operating systems.

4.2 Linux kernel structures

`task_struct` is main structure for every process. This structure is quite big, around 1 Kbyte in year 2009. Information interesting for logical analysis are shown in figure 4.1.

On the other hand in `module` structure are just few interesting information, as can bee seen on figure 4.2.

4.3 Finding VPT

Virtual page table is the root of structure used for translation from linear to physical address.

This translation is heavily architecture dependent. In Linux, address of VPT is exported as `pg_swapper_dir` or `swapper_pg_dir` symbol, and can be found in "System.map" file.

For i386 architecture, the following pseudo-algorithm is used:

- Start search at 0x00300000
- Check every 4KB aligned position
- Look for block of zeroes of length 0xc00.

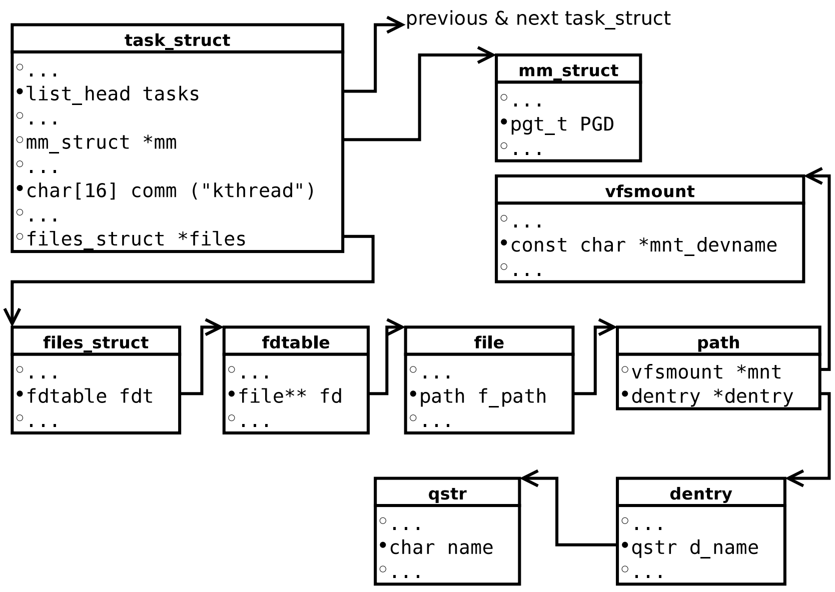

Figure 4.1: Kernel structures 1 (task_struct related)

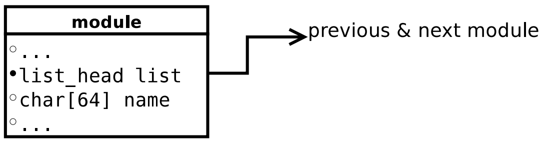

Figure 4.2: Kernel structures 2 (module related)

- If you find such a block (4KB), skip first 0xc00 bytes (zeroes), and then try following test:
- If first 3 pointers masked by 0x6f equals each other, than this should be virtual page table. This heuristic based on set flags is definitely not perfect test, even if not failing on known configurations. I believe, that with dumps from enough different configurations, better heuristic can be developed. But this can be done after when first dump where Foriana fail is found.

4.4 Using VPT

Translation from linear to physical address is realised by function `read_linear()`. Function was written according to Intel [6] and AMD [4] hardware specifications. As i386 hardware specification is old and because of backward compatibility quite complicated some rare cases are not implemented, and can be added later.

Pseudo-algorithm:

- Inputs are PGD, pointer to linear address and size of data to read.
- Find correct PGD (Page Directory): Default (0) is kernel PGD, otherwise supplied PGD (for each process)
- From information in PGD and linear address derive paging mode used. (4MB or 4KB page size, ...)
- Process whole VPT tree structure, according to specification.
- As requested block size can be bigger then used page size, data may be stored in multiple pages. Recursively as described in previous steps complete whole requested linear block from pages in memory dump.

Described function permits following linear pointers inside dump, and solves memory fragmentation.

4.5 Determination process

Before determination of processes and modules can begin, first phase (finding VPT) have to be complete. This determination phase have to be completed only once. After exact distances valid for analysed kernel are known, they are saved into structure `my_image` and can be used directly.

Pseudo-algorithm used while looking for system structures:

4.6 Finding processes

Pseudo-algorithm for finding and describing process structure (`task_struct`) as implemented in Foriana:

- 1: Find string LINUX_MAGIC_PROCESS in memory dump (via fulltext search). By default, this is "kthread", which was present in all analysed memory images. (Sometimes this process is called "kthreadd", but this is not problem for fulltext search, when terminating zero is not used). Process "init" was not used, event through "init" should be first process of all Linux systems by definition. Reason is, that string "init" is simply too common, and creates too many false positives.

- 2: Check, if found string is inside task_struct structure. String should represent "comm" variable. This verification is made by counting head_list structures, found maximally TS_MAGIC_MAX_TRIES bytes before comm string.

  head_list as defined in Linux kernel source code is structure, that contains pointers to previous and next head_list structures in list. If structure contains pointer to itself, the list contain only one head_list. See figure 3.1 for visualisation.

  This was the first implemented method. Produces relatively good results, but sometimes error occur. Therefore another cross-checking algorithm had to be implemented into step 2.

  Using two constants TS_MAGIC_HEADLIST_FRONT_NUMBER and TS_MAGIC_HEADLIST_FRONT_SPREAD, size of SPREAD lists around FRONT_NUMBER is compared, and the longest list is considered to be task_struct list. (The list of all processes.)

- 3: Finally distance between comm variable and head_list is saved into my_image structure. This value will be later used each time some function need it. Also whole my_image structure is saved to cache file by default.

4.7 Finding modules

Please note, that module structures are not included in memory image obtained by "soft" methods, like using program dd. In Linux kernel, module structures are in part of memory called "high memory" (or simply "highmem") that is not mapped into linear address space. To obtain memory with module structures, "hard" method must be used, for example firewire DMA access, or coldboot method.

Initially, the same algorithm as for processes was used (as described in previous chapter). Later, because beginning of `module` structure is much more simple, algorithm was simplified, and nearest `list_head` before module name is chosen. In this case, method is reliable enough, and speed was improved. Paradoxically, this method is even more reliable because is really simple, and is more resistant to image inconsistencies.

4.8 Finding open files

Looking for open files (or file descriptors) using previously described method is quite difficult. From `task_struct` to list of open files for this `task_struct`, 5 pointers have to 5 different structures have to be processed, each with unknown offsets used. This is a lot of guessing at one time, and can be very time consuming, or even impossible. I have tried to implement this feature into Foriana, but it is not working until now. However, the source code for this functionality is commented out and present in file `arch/i386/functions.c`

Whole situation is illustrated in figure 4.1.

Open files (resp. file descriptors of open files) for each process can be accessed through `files_struct`. But to access filenames themselves five pointer dereferences (`fdtable, file, path, dentry, qstr`) are required. This represent a problem, as exact offsets of structures mentioned above are unknown. Determination process can be simplified by fixing structure offsets up to `path` structure. Path structure contains only 2 pointers to structures: `vfsmount` and `dentry`. `vfsmount` contains char `mnt_devname`, name of a device the file is stored on. It is relatively safe to assume that this name starts with string '/dev/'. Using described optimization, determination process can be completed in two steps, reducing complexity of guessing by one level.

This can look as minor improvement, but is important to realize, that guessing without fixed point have complexity exponential with base of the constant chosen for each step. Five levels of search are touching limits of practical usability.

Opposite to listing of processes and modules, it is almost sure that listing of open files will work only for 2.6 branch of Linux kernel.

4.9 Listing modules

In process of determination module structures, address of one of modules was stored into cache. So take this address, and using this stored offsets between `list_head` and module name, listing of `list_head` is relatively straightforward. Only risk is jumping into endless loop. To prevent this, check for revisiting initial module is implemented. Even if checking for first module, endless loop can occur (for example 1,2,3,4,5,4,5,4,5,4,5,...), so if counter increased with each module reach some limit, loop is detected an listing is terminated. For Linux, this limit is 65535, because there cannot be more modules on standard Linux box. If needed, limit can be easily changed trough header file.

4.10 Listing processes

Listing of processes is very similar to listing modules. Once offsets and one of precesses are known, whole `list_head` is processed and `comm` string printed. The same protection against falling into infinite loop is applied.

For Linux, loop limit is 65535, because there cannot be more processes on standard Linux box. If needed, limit can be easily changed trough header file.

Chapter 5

Foriana - user manual

5.1 Installation

Installation from source code:

```
$ wget https://hysteria.sk/~niekt0/foriana/foriana_current.tgz
$ tar -xvjf foriana\_current.tgz
$ cd foriana_xxx
$ ./configure
$ make
```

Verify, that program was build successfully.

```
$ ./src/foriana
Syntax error.

foriana 0.5.0 by niekt0@hysteria.sk
Usage: foriana [options] RAM_image
Options:
  -a/--all : try to dump everything available. Generally not good idea.
...
```

Eventually, install Foriana on your system.

```
$ sudo make install
```

5.2 Analysing image

To list program options, type

```
$ foriana --help
```

If you feel lucky enough, try just

```
$ foriana image
```

Foriana will try to use all compiled in functionalities. Please note, that this is not optimal way for every image. For example, if you do not have dump of highmem, Foriana will spend a lot of time searching for module structure, that is not inside. Also, memory can contain inconsistencies, which may give you strange results. Like structure being falsely detected, and thus listing pseudo-random strings.

Better approach is to analyse image step by step. First, try to determine offsets in structures.

```
$ foriana -d image
```

If you have dump without highmem, you can save time by trying

```
$ foriana --skip-determine-modules -d image
```

Now, if successful, Foriana should create file named .image (where image is name of analysed image) that contains information about analysed image. So slow determination have to be done only once. From now, you can try to list information.

```
$ foriana --list-processes image
```

or just

```
$ foriana image
```

Foriana automatically checks for .image in current directory, and use stored information if available. The easiest way to store gathered results is to use standard shell pipes, for example

```
$ foriana image > date-image
```

5.3 Results verification

Result verification have to be done manually, by creating pattern for structure of interest, and then using fulltext search for that pattern.

Structures addresses can be found in Foriana output after image analysis. For example

```
...
 process address: c18f5098, name: migration/0
 process address: c18fbad8, name: ksoftirqd/0
...
```

Possible automation of this process is described in section 3.5.

Chapter 6

Conclusion

6.1 Testing images

Program was tested on 10-20 RAM images of Linux systems of various quality. Some of images were obtained by ”soft” methods, some via firewire, few were incomplete or damaged.

6.2 Obtained results

Successful run looks similar to following example. Determination process:

```
$ foriana -d  x60-32bit-1G-kde

Analyzing file x60-32bit-1G-kde
Dumped RAM size is 1015 MB.
Looking for cache_file: ".x60-32bit-1G-kde" ... no valid cache found (err 1).
VPT_address found at: 0x927000
        linux_determine: Looking for process "kthread".
Found valid process at 0x1915bbc.
        i386_lin_find_userspace_process: Looking for userspace process "init".
Found userspace process "init" at 0xc18f5a98
Found userspace PGD at 0xf7bfc000 and determined distances: mm_pgd: 36, hl_mm:
Determination process of task_struct complete.
        linux_determine_module: Looking for module "yenta_socket".
        linux_determine_module: Starting at address 0x30000000.
        i386_lin_verify_module: Found module struct at negative offset 8
```

```
Found valid module at 0x3d5b888c.
Determination process of module structure complete.
Saving determined information to cache file.
Clean exit;).
```

Listing process:

```
$ foriana --list-processes  x60-32bit-1G-kde-noclean
Analyzing file x60-32bit-1G-kde-noclean
Dumped RAM size is 1015 MB.
Looking for cache_file: ".x60-32bit-1G-kde-noclean" ... ok.
Listing processes:
(linear addresses of kernel, usually substitute 0xC0000000 to get position in
        process address: c18f5098, name: migration/0
        process address: c18fbad8, name: ksoftirqd/0
        process address: c18fb5d8, name: migration/1
        process address: c18fb0d8, name: ksoftirqd/1
        process address: c1910b18, name: events/0
        process address: c1910618, name: events/1
        process address: c1910118, name: khelper
        process address: c1915a98, name: kthread
        process address: dfc1fa98, name: kblockd/0
        process address: dfc1f598, name: kblockd/1
        process address: dfc1f098, name: kacpid
        process address: c19b00d8, name: ata/0
        process address: c19b4b18, name: ata/1
        process address: c19b4618, name: ata_aux
        process address: c19b4118, name: ksuspend_usbd
        process address: c19b5098, name: khubd
        process address: c19b75d8, name: kseriod
        process address: c19bf118, name: khpsbpkt
        process address: c19b5598, name: knodemgrd_0
        process address: c19b5a98, name: pdflush
        process address: c19b05d8, name: pdflush
        process address: c19b0ad8, name: kswapd0
        process address: c1966098, name: aio/0
        process address: c1966598, name: aio/1
        process address: c1966a98, name: jfsIO
        process address: c1965118, name: jfsCommit
        process address: c1965618, name: jfsCommit
        process address: c194db18, name: jfsSync
        process address: c194a0d8, name: xfslogd/0
        process address: c194a5d8, name: xfslogd/1
        process address: c194aad8, name: xfsdatad/0
        process address: c1927098, name: xfsdatad/1
        process address: c19bead8, name: ocfs2_wq
        process address: c194f598, name: user_dlm
        process address: dffc3a98, name: scsi_tgtd/0
        process address: dffc6118, name: scsi_tgtd/1
        process address: dffce618, name: scsi_eh_2
        process address: dffcca98, name: scsi_eh_3
        process address: dffce118, name: scsi_eh_4
        process address: dffc3598, name: scsi_eh_5
        process address: dfee3b18, name: exec-osm/0
        process address: dffc05d8, name: exec-osm/1
        process address: dffbdb18, name: block-osm/0
        process address: dffc00d8, name: block-osm/1
        process address: dfed2098, name: scsi_eh_6
        process address: dfee0ad8, name: usb-storage
        process address: dfee3618, name: kcryptd/0
        process address: dff97098, name: kcryptd/1
        process address: dffceb18, name: ksnapd
        process address: dffc0ad8, name: kmirrord
        process address: dffbd118, name: kirqd
        process address: dff93ad8, name: aufsd
        process address: dffc6b18, name: aufsd
        process address: dffaeb18, name: aufsd
        process address: f7d7e618, name: aufsd
        process address: c1927598, name: loop1
        process address: c19b7ad8, name: loop2
        process address: c1954b18, name: loop3
        process address: f7d16598, name: loop4
        process address: dffae118, name: loop5
        process address: dffb6118, name: loop6
        process address: c1965b18, name: loop7
        process address: c19c8118, name: loop8
        process address: dffb1a98, name: loop9
        process address: c19be5d8, name: loop10
        process address: c19b70d8, name: loop11
        process address: c194fa98, name: loop12
        process address: c1915098, name: loop13
        process address: dffcc598, name: loop14
        process address: f7db0b18, name: loop15
        process address: c1926118, name: loop16
        process address: dffaf5d8, name: loop17
        process address: c1959598, name: loop18
        process address: c19be0d8, name: loop19
        process address: f7d855d8, name: loop20
        process address: c1959a98, name: loop21
        process address: dffa75d8, name: loop22
        process address: dffb9ad8, name: loop23
        process address: f7d7e118, name: loop24
        process address: dffa8a98, name: loop25
        process address: c19bfb18, name: loop26
        process address: f7d81098, name: loop27
        process address: c19c8b18, name: loop28
        process address: dffbca98, name: loop29
        process address: dffa7ad8, name: loop30
        process address: dfee00d8, name: loop31
        process address: f72bcb18, name: loop32
        process address: dffcaad8, name: loop33
        process address: dffaf0d8, name: loop34
        process address: dfed2a98, name: udevd
        process address: f69470d8, name: pccardd
        process address: f6b2e0d8, name: iwl3945/0
        process address: f698ea98, name: iwl3945/1
        process address: f69e4618, name: iwl3945
        process address: f694fb18, name: kmmcd
        process address: f697f118, name: kpsmoused
        process address: f6b380d8, name: mount.ntfs-3g
        process address: f6b81ad8, name: rc.M
        process address: dffca0d8, name: syslogd
        process address: f7d685d8, name: klogd
        process address: f6b29598, name: acpid
        process address: dffb1098, name: dbus-daemon
        process address: f6bcb118, name: hald
        process address: f6b35098, name: hald-runner
        process address: f6947ad8, name: hald-addon-keyb
        process address: f7d7d0d8, name: hald-addon-acpi
        process address: f7d16a98, name: hald-addon-keyb
        process address: f6cb8098, name: hald-addon-keyb
        process address: f7d12618, name: hald-addon-keyb
        process address: dffbd618, name: hald-addon-stor
        process address: dffb90d8, name: crond
        process address: f7d85ad8, name: gpm
        process address: f7d79098, name: bash
        process address: f69c50d8, name: startx
        process address: f6b92b18, name: xinit
        process address: dfed2598, name: X
        process address: f694f118, name: xinitrc
        process address: dff97a98, name: startkde
        process address: f6aa00d8, name: start_kdeinit
        process address: c1926618, name: kdeinit
        process address: f72bc618, name: dcopserver
        process address: f69c5ad8, name: klauncher
        process address: f7db0118, name: kded
        process address: f694f618, name: gam_server
        process address: f72bc118, name: kwrapper
        process address: f69a80d8, name: ksmserver
        process address: c19bd598, name: kdesktop
        process address: f697f618, name: kicker
        process address: dff97598, name: kio_file
        process address: f6ab4118, name: kxkb
        process address: dffb95d8, name: artsd
        process address: f7d7eb18, name: kaccess
        process address: dfee3118, name: krandrtray
        process address: f69a85d8, name: kmix
        process address: f69b1618, name: startcompiz.sh
        process address: dffb1598, name: fusion-icon
        process address: f697fb18, name: knotify
        process address: dffa8098, name: compiz
        process address: dffb6618, name: dbus-launch
        process address: f6b3c118, name: dbus-daemon
```

```
        process address: f6b3cb18, name: kde-window-deco
        process address: f69c55d8, name: konsole
        process address: f6cb8a98, name: sh
        process address: c084a4a8, name: swapper
        process address: c18f5a98, name: init
Listing processes done.
Listed 145 processes (hopefully).
Clean exit;).
```

During testing, all not damaged images were analysed correctly.

Damaged/incomplete images, produced various results: Program execution stopped after determination process failed, program stopped after runtime inconsistency detected, listing empty members of list, and with maliciously damaged file even potentially infinite loop was reached. (Actually, Foriana sanitizes such cases, and loop is terminated after 65 000 repeats, because there cannot be more processes/modules on current Linux system)

In real world, determination process takes from few seconds, up to tens of seconds for 3GB images. Exact time depends on position of structures we are looking for inside memory dump, that can be theoretically random. Once determination process is complete, listing and reading linear addresses is done in fragments of seconds.

6.3 Conclusion

Analyzing memory dump at logical level proved to be functional complement to classic pattern-matching methods. Some of the ideas were implemented in program Foriana. Program produces fast and reliable results, but amount of gathered information and supported architectures are limited. Some new ideas were developed, described and are waiting for implementation in future work.

Bibliography

[1] Burdach Mariusz *Digital forensics of the physical memory*, Warsaw, 1995.

[2] Andreas Schuster *Searching for processes and threads in Microsoft Windows memory dumps*, Digital Investigation 3S (2006) pp 10–16. ISSN 1742-2876, DOI: 10.1016/j.diin.2006.06.010.

[3] Jorge Mario Urrea *An Analysis of Linux RAM Forensics*, Naval Postgraduate School Monterey, (2006).

[4] *AMD64 Architecture Programmer's Manual Volume 2:System Programming*, Advanced Micro Devices Inc., (2007).

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, Intel Corporation, (2008).

[6] *TLBs, Paging-Structure Caches, and Their Invalidation*, Intel Corporation, (2007).

[7] Aaron Walters, Nick L. Petroni *Volatools: Integrating Volatile Memory into the Digital Investigation Process*, Komoku, (2007).

[8] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, Edward W. Felten *Lest We Remember: Cold Boot Attacks on Encryption Keys*, Princeton University, (2008).

[9] Gabriela Limon Garcia *Forensic physical memory analysis: an overview of tools and techniques*, Helsinki University of Technology, (2007).

[10] David R. Piegdon *Hacking in physically addressable memory*, RWTH Aachen University, (2007).

[11] Joanna Rutkowska *Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools*, COSEINC, 2007.

[12] Adam Boileau *Hit By A Bus: Physical Access Attacks with Firewire* , "Ruxcon 2k6", 2006.

[13] Adam Boileau *1394memimage*, `http://storm.net.nz/static/files/pythonraw1394-1.0.tar.gz`, 2006.

[14] Adam Boileau *winlockpwn*, `http://storm.net.nz/static/files/winlockpwn`, 2008.

[15] ManTech International *MemDD*, `http://sourceforge.net/project/showfiles.php?group_id=228865`, 2008.

[16] Wikipedia *Various articles* , `http://www.wikipedia.org`, 2009.