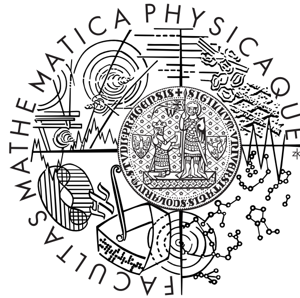
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Ivor Kollár

Forensic RAM dump image analyser

Department of Software Engineering

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

2010

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

I hereby declare that I have written this thesis without any help from others and without the use of documents and aids other than those stated above and that I have mentioned all used sources and that I have cited them correctly according to established academic citation rules.

In Prague 04.08.2010 Ivor Kollár

Contents

1 Introduction **6**
- 1.1 Problem overview . . . 6
- 1.2 Assumed usage . . . 6
  - 1.2.1 Forensic analysis . . . 6
  - 1.2.2 Penetration testing . . . 7
- 1.3 Physical RAM . . . 7

2 Existing documents and tools **8**
- 2.1 Image extraction . . . 8
  - 2.1.1 OS independent . . . 8
  - 2.1.2 MS Windows . . . 10
  - 2.1.3 Linux kernel . . . 10
- 2.2 Image analysis . . . 11
  - 2.2.1 MS Windows . . . 11
  - 2.2.2 Linux . . . 12
  - 2.2.3 Detection of cryptographic keys . . . 13

3 Detection of cryptographic keys and passwords **14**
- 3.1 Problem definition . . . 14
- 3.2 Frozen cache . . . 14
- 3.3 Finding keys . . . 15
- 3.4 Real patterns . . . 19

4 Methods for finding structures of partially known OS **22**
- 4.1 Problem definition . . . 22
- 4.2 Architecture specific structures . . . 22
- 4.3 Pattern matching . . . 23
- 4.4 “The Longest-Nearest” algorithm . . . 23
- 4.5 Dynamic pattern generation . . . 26

4.5.1 Generated patterns 27
4.5.2 Pattern optimization 28
4.5.3 Real results 28
4.6 SLABS 29
4.7 Parsing source code 30

5 Foriana 31
5.1 Internal Structure 32
5.2 Architecture support 33
5.2.1 ARM 34
5.2.2 x86_64 34
5.2.3 i386 35
5.3 Operating systems support 36
5.3.1 Determination process 36
5.3.2 Finding processes 37
5.3.3 Unknown system 37
5.3.4 Linux 38
5.3.5 BSD 42
5.3.6 Listing information 42

6 Foriana - user manual 44
6.1 Installation 44
6.2 fmem module 45
6.3 Analysing image 46
6.4 Results verification 48

7 Evaluation 49
7.1 Testing images 49
7.2 Obtained results 49
7.2.1 Linux 49
7.2.2 MS Windows 53
7.2.3 BSD 55
7.2.4 Other systems 56
7.3 Conclusion 57

Literature 59

Title: Forensic RAM dump image analyser
Author: Ivor Kollár
Department: Department of Software Engineering, Faculty of Mathematical and Physics
Supervisor: RNDr. 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

Název práce: Forenzní analýza obrazu paměti RAM
Autor: Ivor Kollár
Katedra (ústav): Katedra softwarového inženýrství, Matematicko-fyzikální fakulta
Vedoucí diplomové práce: RNDr. Viliam Holub, Ph.D.
e-mail vedoucího: holub@dsrg.mff.cuni.cz

Abstrakt: Pro získávání obrazů operační paměti počítače existují různé techniky, většina z nich ale poskytuje syrový obraz dat, který je obtížné analyzovat. Cílem téhle práce je vyvinout program pro automatickou analýzu obrazů paměti poskytující obsažené informace v uživatelsky přívětivé formě. Program by měl zjednodušit automatické dolování dat a předpokládaným uživatelem jsou forenzní znalci. Předpokládané technické schopnosti jsou podpora vícero architektur a operačních systémů, hádaní cílové architektury a operačního systému, automatické získávaní hesel a šifrovacích klíčů, vypsání seznamu procesů, modulů, ovladačů.
Klíčová slova: forenzní, analýza, RAM, paměť, klíč

Chapter 1

Introduction

1.1 Problem overview

Suppose that there exists a method for retrieving a content of the whole physical RAM of a running computer. Such method produces megabytes or even gigabytes of raw data. The simplest 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 hard, for example listing running processes of the operating system (OS). When the 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

The goal of this work is to design a program Foriana capable of fast analysis of a dump of a physical RAM, and extraction of the interesting information. The most common use-cases are forensic analysis and penetration testing.

1.2.1 Forensic analysis

Forensic analysis extracts as much information as possible from the object of interest. In computer science, forensic analysis is usually categorized as "in vivo" and "post mortem". The term "in vivo" can be translated as analysis

of running and responding system and “post mortem” can be translated as analysis of previously gathered data that does not change. The assumed usage of Foriana is somewhere between these two categories: Retrieving the content of a RAM is typically performed “in vivo”, while analysis itself is “post mortem”.

The content of RAM is an important piece of information, when trying to analyse previous activity on the machine. RAM can contain for instance pieces of processes, deleted files, user’s sessions, cryptographic keys. With advancing information security awareness and wide deployment of encrypted filesystems, retrieving decryption keys is becoming a “key” task. In secure system, RAM is often the only place where keys are kept.

1.2.2 Penetration testing

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

Since memory can change without warning, image analysis has to be completed as fast as possible.

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 work we understood under the word RAM a volatile type of memory, where the information is lost after the power is switched off. As explained in [9], time between power loss and data loss can be long enough to capture the 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 the content of RAM have been described. If possible, an investigator should prefer hardware methods, because any manipulation with the target system can lead to information loss. Hardware methods also often allow bypassing of OS security mechanisms like locked screensaver, etc.

2.1.1 OS independent

There are two main publicly known methods requiring physical access to the hardware. The first one is based upon DMA (Direct Memory Access) and second one is 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 a 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 and intelligence investigators. Several types are available on the market today but usually with a restriction to military/police agencies or officers. Cardbus, PCI and AGP connectors were seen on the market. The advantages of these devices are professional support and reliability. They can be used by technically

inexperienced users. 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 [13], allowing target system to crash before RAM is copied, or even fake the results.

- Method 2: Firewire

  Firewire attack is a special case of the previously mentioned method. The difference is in its availability. Firewire cable can be easily purchased as well as a portable notebook with a firewire connector. Adam Boileau wrote a tool for dumping physical RAM via firewire, `1394memimage` [16]. Later, he also released a tool for taking control over the computer running a MS Windows operating system, `winlockpwn` [17]. `1394memimage` is free and 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 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

  RAM is quickly physically frozen, which makes information in RAM persistent for longer period of time even without an electric power. From a forensic point, this is the most valuable and reliable approach but may be difficult to perform in real situations. More information can be found in Cold Boot Attacks paper [9].

  Advantages of this method are perfect RAM image (no race conditions), difficult defence and universal approach to all investigated computers. On the other hand the investigator needs sophisticated equipment. If not copied quick enough, RAM can contain a lot of random errors.

- Method 4: "hot boot" attack

"Hot boot" method is similar to "cold boot" attack, but instead of freezing RAM, the computer is restarted and booted from investigator's medium (flash disk, CD-ROM, ...). Depending on BIOS manufacturer and version, several scenarios may occur.

If booting from alternative medium is enabled, special forensic software will be loaded and starts dumping the RAM. If "lucky" enough, BIOS did not clear the RAM content during the boot process (RAM initialization) and investigator will obtain a full memory dump. Unfortunately, some BIOSes do clear RAM content during the boot process.

This problem can be solved by using special computer with an open case dedicated to this task. When removed and inserted into other computer quick enough, the most of RAMs will keep their content. According to [9], "quick enough" seems to be up to 30 seconds with quality degrading over time. With little training, RAM can be exchanged in a few seconds, making "hot boot" attack real for practical use. Disadvantages are variable quality of the memory dump and the need of a dedicated equipment for retrieving.

There are few tools available for coldboot and hotboot method of memory imaging, `bios_memimage` and `efi_memimage`, both freely available [1].

2.1.2 MS Windows

To obtain memory image of running MS Windows, `memdd` [18] can be used, or other specialized MS Windows tools, such as `windd` [22] (often referenced as win32dd and win64dd) or commercial `WinEn` from EnCase Forensic toolkit. To perform dump of OS memory, system user privileges are required. For the practical usage, the `memdd` was found the most trouble free. Alternatively, system hibernation file can be used, if available and user has used hibernation function during his work. There are a lot of different tools, mostly commercial, not listed in this work. The full list can be found on [25].

2.1.3 Linux kernel

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

[1] http://citp.princeton.edu/memory/code/

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

with root privileges. This is a "quick and dirty" method because `/dev/mem` will not read more than 1 GiB of data. In some systems, such as Redhat/Fedora distributions, the implementation of `/dev/mem` device is limited even more, allowing access only to first 1 MiB 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 a quick dump.

Alternatively, on older Linux kernels, mmap() trickery could be used. According to experiments done while completing this work, it is not possible anymore.

A more sophisticated solution would be to write special dumping module which would create a virtual character device similar to `/dev/mem` but without limitations.

Such tool have been implemented in this work, `fmem`. After writing `fmem` tool we have realised that similar results can be achieved with `crash`, found in the Linux kernel source tree in file `linux/drivers/char/crash.c`. The reasons for overlooking the existence of this module is probably the fact that the module is included inside the `crash` utility [23].

Similar to the MS Windows platform, depending on system configuration, a swap partition can contain parts of hibernated RAM. This is the worst quality image but may be the only one available in some cases.

2.2 Image analysis

2.2.1 MS Windows

Andreas Schuster wrote `ptfinder` for finding processes in memory images of MS Windows. `ptfinder` is based on basic pattern matching. The tool can be found on the website[2].

The `Memparser` [3] from Chris Betz is a different implementation of a similar ideas.

`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" [8]. At the time of writing this article, Linux support was not implemented, or at

[2]http://computer.forensikblog.de/files/ptfinder/

[3]http://www.dfrws.org/2005/challenge/memparser.shtml

least not publicly released. Framework is still under development and have the best perspective from available projects.

There are also plenty of small "one-time" scripts and programs for MS Windows memory analysis but not general enough and without further support, so they 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. The Toolkit can be found on the website [4].

However, it seems that these tools are not working on 2.6 kernels, at least we were not able use them successfully. `Idetect` toolkit is probably not maintained any more.

An interesting project, `draugr`, was introduced in 2007, but only one release was made and the tool is probably not supported any more. According to its website[5], this tool should be able to list processes by both following the linked list and bruteforce search, resolve symbol addresses without system.map and has an integrated disassembler. All these functionalities are documented in demonstration videos but in practical tests this tool was not found working, at least by the author of this article.

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

The most sophisticated project seem to be the `crash` [6] utility, when patched as described in Linux Memory Forensic paper [24]. Prerequisites for successful work are a memory dump, `System.map` file and a kernel binary. Once these resources are available, crash is a powerful tool supporting number of different architectures and allowing user to list processes, modules, open files, mounted filesystems and many others.

---

[4] http://forensic.seccure.net/tools/idetect.tar.gz
[5] http://code.google.com/p/draugr/downloads/list
[6] http://people.redhat.com/anderson/

2.2.3 Detection of cryptographic keys

As shown in [9], detection of cryptographic keys can be done OS independent, even without the source code of analysed applications. Also, when looking for “key schedule” mechanism of cryptographic algorithm, usual protection that “wipes” key in memory can be bypassed. The key schedule structure contains enough information to reconstruct whole cryptographic key with very high success rate (over 95%) [9].

In previously cited work, authors implemented tools for detecting AES keys (`aeskeyfind`) and RSA keys (`rsakeyfind`).

During testing, both tools were found working relatively fine. On some tested images `aeskeyfind` found a lot of false positives (up to 10 per one real key) but also all correct keys. Presence of false positives is not really a problem, when an oracle which verify key validity is available (for example encrypted partition). Alternatively, `threshold` parameter can reduce the number of false positives.

The `rsakeyfind` utility is written using mapping file into virtual memory, which makes it unusable for analysis of big memory images on i386 architecture. On smaller images program seems to work correctly, and produces reasonable results with tolerable amount of false positives.

To make the list of tools complete, there is also `aesfix` capable of error-correction for AES key schedules. All three tools can be found at the website [7].

Another tool is `interrogate` [26]. Should be able to recover AES, RSA, WIN-RSA, serpent and twofish keys, and to reconstructs the virtual address space of process. In the tests, key finding took much longer and produced more false positives than `aeskeyfind` and `rsakeyfind` utilities. When attempting to reconstruct virtual address space of process, the program was generating segmentation fault.

More information about detection of cryptographic keys can be found in the next chapter.

[7] http://citp.princeton.edu/memory/code/

Chapter 3

Detection of cryptographic keys and passwords

3.1 Problem definition

A memory can contain various authentication tokens (cryptographic keys, hashes, one-time passwords, login names and passwords) various storages and services (instant messaging, mailboxes, shell accounts, . . .).

Tools for detection of cryptographic keys were discussed in section 2.2.3. This chapter describes possible defence, probabilities of the success in finding strings and some practical strings to look for.

3.2 Frozen cache

The defence against the attacks looking for key schedule (sometimes referenced as “round keys”) structure is up to the encryption application. Key schedule structures are not necessary in memory, but their computation each time they need to be used is a performance problem.

Even without key schedule structures, an attacker can try to attack the key itself. The attack would be less effective and practical but still possible.

Interesting concept of protecting cryptographic key is shown on [20]. At the time of writing this article, practical implementation is still not available.

In short, the idea is to store the key itself, the key schedule structure, a eventually all the critical information into CPU cache. Therefore, this information will be not in RAM, and cannot be dumped by available techniques.

This solution is possible but practical realisation is not trivial. Storing custom information in CPU cache implies performance problems, new problems on multiprocessor systems, and problems with the structure of code. Current operating system have not been designed for such requirements and a reasonably fast implementation would require significant changes.

More information and actual status can be found on the website [1].

Similar results could be achieved using TPM (Trusted Platform Module). It would be possible to bind event to keyboard shortcut or screensaver activation. This event would migrate keys to the cache or TPM module and clear the key schedule structures.

3.3 Finding keys

Because of the existence of tools mentioned in previous sections, these ideas were not re-implemented. Instead, `keyfind` tools were added to Foriana package preserving original copyrights.

Therefore other searching methods were explored. The most robust seems to be classic pattern matching. Classic pattern matching is not automatic, not ready for new situations, and often require operator's help for generating patterns. Speed can be a serious problem too, especially on bigger RAM images. Pattern matching also expects, that the pattern is stored in a continuous memory.

On the other hand, pattern matching can be easily performed using generally available Unix tools (strings, xdd, grep, . . .). Their usage is really simple and generally known.

Available tools and their interfaces are "state-of-the-art" and can be combined in many unexpected ways. Anything similar implemented into Foriana would be inevitably worse. For the purpose of pattern matching of binary strings with substitutions, `bgrep` utility was modified and included into Foriana package.

Key files have often standardized format that is stable for long time. Key files themselves are usually encrypted but they are interesting even though. Key files can be bruteforced and may be needed for verifying some of the found passwords.

When constructing patterns, two opposite requirements have to be considered: The longer the pattern is, the lower the chance of false positives.

[1] http://frozencache.blogspot.com/

The longer the pattern is, the higher the chance that pattern in memory is split and will be missed. Long patterns also increase the risk of miss when memory image is not perfect and contain errors.

So let's estimate the probability that string of length n will get split while using m byte pages in RAM of size M bytes and pattern placement is completely random.

Split placements:

$$(n-1) * (M/m-1) \approx n * M/m$$

All possible placements:

$$M - n \approx M$$

(as n is very small compared to M;)

So the probability of split will be:

$$p = ((n-1) * (M/m-1))/(M-n) \approx (n * M/m)/(M) = n/m$$

Quick and dirty guess is n/m. For the real world, if 4 KiB pages are used, string up length 40 will have (40 /4096 = 0.00977) less than 1% chance to get split. With 2 MiB or 4 MiB pages in use, this probability will be under 0.02% resp. 0.01% even for 400 bytes long pattern.

As we can see, with a perfect memory image, we don't have to take patterns length in consideration, if 2 MiB or bigger pages are used. Some problems arise with commonly used (in i386 architecture) 4 KiB pages. Up to the length about 20 bytes, chance to fail is less then 0.5%, which seems to be enough for the most of searched patterns. When looking for longer patterns (some e-mail addresses, names in unicode, etc) the truncation of pattern should be considered, even for the cost of false positives.

This estimation assumes a perfect image, which is uncommon for some extraction techniques. With the presence of error bits, situation changes dramatically. Even the presence of 1 % of incorrect bits will increase probability of a miss for 20 bytes long pattern approximately to:

Probability that 1 bit is flipped: 1%

Probability that in 20 bytes (160bits) no bit is flipped:

$$0.99^{160} \approx 20\%$$

This is not much.

Probability that in 20 bytes (160bits) exactly 1 bit is flipped:

$$((0.99^{159}) * 0.01 * 160) \approx 32\%$$

Probability that in 20 bytes (160bits) exactly 2 bits are flipped:

$$((0.99^{158}) * 0.01^2 * 160) \approx 25.99\%$$

(generally):

$$p(i) = (1-p)^i \cdot p^{n-i} \binom{n}{i}$$

where p(i) is probability that there are exactly i error bits in the pattern, p is probability of a bit flip and n is the number of bits in the pattern.

Thus probability for up to 3 flipped bits is

$$20.03\% + 32.37\% + 25.99\% + 13.74\% \approx 92.13\%$$

Similarly for 1 per mille probability of bit flip (p=0.001) and 20 bytes long string:

$$85.20\% + 13.65\% + 1.09\% + 0.06\% \approx 100.00\%$$

Note that this is not "real" probability because error bits are not distributed completely random, as can be seen in Figure 3.1.

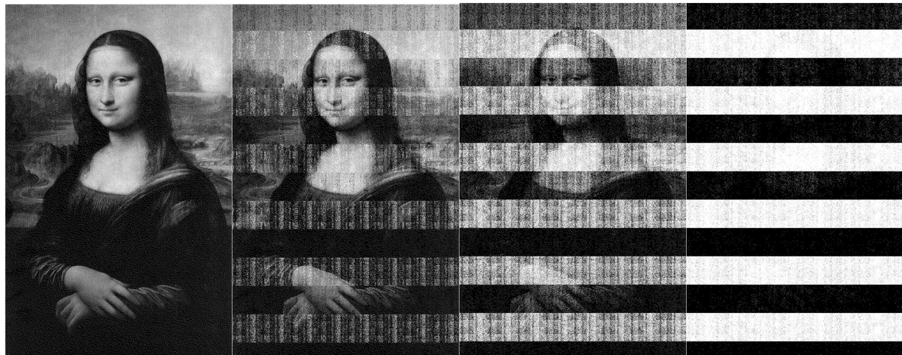


Figure 3.1: Error distribution in real memory [9]

To be able to search for strings in an image with errors, the utility that computes Hamming distance for each pattern need to be used.

With pattern and max. error bits specified, deterministic Turing machine can be generated, containing all possible patterns derived from original pattern with error bits. Then the real pattern can be matched in dump file using (for example) Aho-Corasick or Bitap algorithm.

Let's quickly estimate the size of such Turing machine (approximate amount of possible derivation).

For example string of length 20, and 1% of incorrect bit. 20 bytes = 160 bit, so let's say that 2 or 1 bit is reversed.

That is

$$\binom{20}{2} + \binom{20}{1} + 1 = 190 + 20 + 1 = 211$$

As number of error bit grows, number of possible patterns grows exponentially. For 3 bits, this would be

$$1140 + 190 + 20 + 1 = 1351$$

Generally:

$$n = \sum_{i=1}^{r} \binom{l}{i}$$

where n is number of possible patterns, r number of flipped bits and l length of pattern in bits. (similarly for whole chars).

Thus, we are able to correct only relatively low amount of error bits this way.

Similar tool as described here is already implemented: The `agrep` utility [21] (and GPL implementation, `tre-agrep`), but error tolerance works only on character level, not bit level. According to Figure 3.1, this may be even a optimization, because error bits seems to stick together.

In experiments, `agrep` utility was only twice slower when comparing fuzzy (Levenshtein distance up to 8) and strict searching. Using Levenshtein distance instead of Hamming distance is overkill and may even produce some false positives, but agrep implementation of Bitap algorithm seems to be suitable for this situation. Compared to “strings -t x — grep”, agrep was multiple times faster.

3.4 Real patterns

Below is a list of some useful patterns to look for.

Private ssh keys:

- OpenSSH format of RSA and DSA keys. Can be protected by password. Pattern: "`SA PRIVATE KEY-----`".
- RFC 4716 SSH Public Key File Format. Often referenced as "ssh2", and used in commercial ssh software. Pattern: "`---- BEGIN SSH`".
- Putty format of private ssh keys. Pattern: "`PuTTY-User-Key-File`".

OTR (Off-The-Reccord) private keys:

- Gaim/Pidgin format of OTR private keys. OTR can be used as user-to-user encryption over any untrusted IM protocol. Pattern: "`(private-key`".

File `/etc/shadow`:

- Each `/etc/shadow` file should contain root account, so the pattern is "`root:$1$`". "`$1$`" represents beginning of FreeBSD MD5 hash used in the most of Linux distributions. Other Unix systems use different hashes: Openbsd uses blowfish ("`root:$2a$`"), commercial Unixes often use DES ("root:", 13 alphanumerical characters, ":"). Shadow/Passwd file found in memory contain list of users and password hashes. John the Ripper can be used for password cracking.

Various plaintext passwords:

- Subversion format of svn password. Pattern: "`password.V `". Please note that "." is in reality 0x0A, so you need to use "strings -n 2 ..." (minimal length of string) when looking for pattern, or do not use matching based on printable strings.
- Pidgin/Gaim stores account information in XML file, so login information for all protocols (Jabber/ICQ/MSN/...) can be found with this pattern: "`<password>`". This pattern also work for "PSI" client but passwords are not stored in plaintext, and need to be de-obfucsated after discovery.

- MSSQL server password, (tested on Express Studio 2005). Generally, patterns “`password =`” and “`pass =`” works for SQL connections, often can be found in source code. Because parts of (even remotely) edited files can persist in memory, interesting information can be found sometimes.

- Viminfo file, can contain last edited/opened files, possible names of password files, and last searched strings. Pattern: “`This viminfo file was`”

- A lot of important information can be found in memory of terminal emulation applications, such as xterm, mlterm, gnome-terminal, etc. It is important to note that program executed from terminal have no control over terminal memory, and plaintext information (passwords) will remain in the terminal memory with high probability. Usually, when scroll buffer gets full, such information should be overwritten. But sometimes memory can be reallocated, terminal closed before data gets overwritten, or some other situation occurs and data can stay in memory for a long time. Name of shell profile script seem to be a good start and works for all available terminals. Produces a lot of false positives but information that can be found this way are usually worth effort [11]. Pattern “`ast login`” usually appears after successful login to Unix/Linux server. If password authentication was used, plaintext password can be stored nearby. Other patterns: “`.profile`”, “`.sh_profile`”, “`.bash_profile`”

- Memory remembered password in Firefox was found with binary data around. Exact pattern was not discovered, but it was between patterns “`chrome://communicator/locale/wallet/`” and `P.a.s.s.w.o.r.d M.a.n.a.g.e.r` (The second pattern is in UTF-16.) These patterns were extracted from the Linux version of Firefox. Manual extraction will probably be required.

- Skype login credential was found inside a binary structure. Exact pattern was not discovered but password was preceded by patterns “`P.e.o.p.l.e. .O.n.l.i.n.e`” (text in UTF-16) and “`/home/xxx/.Skype/yyy`” where xxx is Linux username of logged user and yyy is Skype username. Actively running Linux version of Skype program was examined. Manual extraction will probably be required.

This list is definitely not exhausting, and contains only patterns for programs available to us. More work needs to be done in this area.

Using patterns is very simple, for example

```
$ strings -t x image.dd | grep 'PuTTY-User-Key'
360b000 PuTTY-User-Key-File-2: ssh-rsa
...
1998acc0 PuTTY-User-Key-File-
```

and then use favourite hex viewer to verify and extract key/password.

Alternatively

```
$ agrep 'PuTTY-User-Key' image.dd
...
```

or

```
$ ./bgrep 50755454592D5573... image.dd
...
```

Practical problems with patterns usually arises: UTF-16 encoding (characters are longer than one byte), diacritics in general, and a lot of false positives.

All described patterns are included in the shell script `pwdfind.sh` from Foriana package.

Chapter 4

Methods for finding structures of partially known OS

4.1 Problem definition

Let's have a RAM image of some not exactly known operating system. By "not exactly known" we mean a system where major version is known, for example Linux or MS Windows, but exact version is 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 the source code for exact version of operating system and that there can be vast amount of different small versions of the 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.

4.2 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 the image. It allows us to follow linear pointers inside the image almost

safely.

4.3 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 that should be present in target system, like process or module name. 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 patterns of analysed structures are available.

4.4 "The Longest-Nearest" algorithm

This algorithm was developed while looking for the 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 that cannot be simplified (too much).
- 2. We know the "name" of at least 1 member of the list. By "name" we mean any string long enough to be full-text searched without too many false positives.
- 3. We are able to follow the pointers found in memory/dump. On most of the architectures this means that we can provide mapping from linear to physical memory.
- 4. Algorithm result is not guaranteed. To provide reasonable results no part of the list can be completely separated from other parts (at least one pointer must point to this sublist of structures)

Algorithm description:

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

- 2. Each time the name is found, make check if this is one of the structures from the 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 in reverse order, but this is not important.

  Testing if we are looking at `list_head` is simple: Follow the pointer we are at and test if the address pointer points to is another pointer that points back. The same check is done for one pointer below. If both pointers satisfy this condition, we are really looking on `list_head`, at least at syntactically correct one. This is visualised in Figure 4.1.

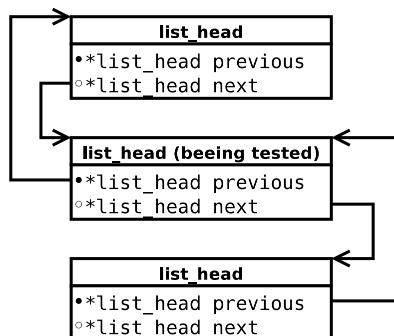
  

  Figure 4.1: Testing list_head

  In the same way we try maximally constant2 `list_heads`. For each `list_head` we compute its length. The longest of `list_head` found should be the `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 the structure, for example lists of children or threads in Linux `task_struct`.

Simplified visualisation is in Figure 4.2.

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.

The algorithm is not resistant to memory errors. It requires working translation between linear and physical addresses. Page directory entry and

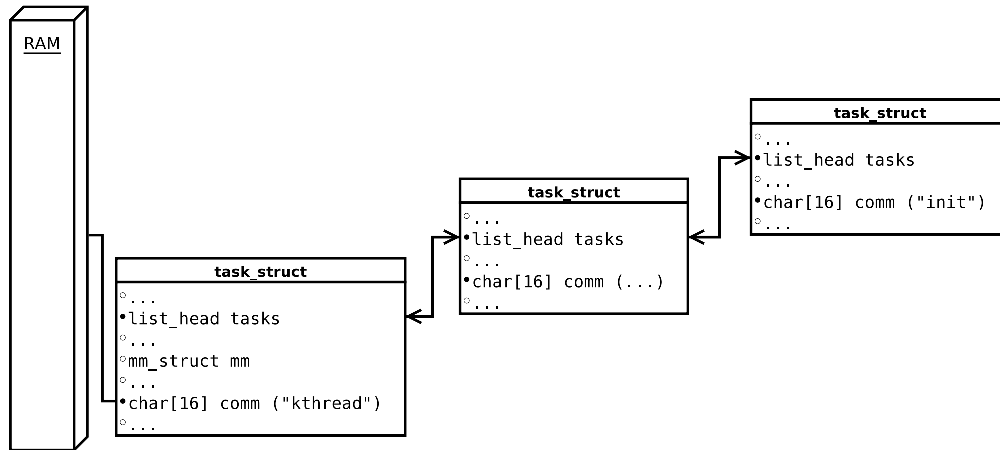


Figure 4.2: The Longest-Nearest

page table entries represent a single point of failure if corrupted. On the other hand, thanks to the mechanism of copying page tables, values used by kernelspace are copied into almost all page directories. They may contain old values, but the most of pages will translate correctly. The task of choosing alternative page directory is left on a user.

Also, pointers in kernel structures should not be too damaged. If so, logical relation will be damaged too and the determination of kernel structures will fail.

Described algorithm (and its modifications) can be used on almost any structure containing `list_head`. In other words, every structure in some double linked list can be determined and listed. Double linked list are widely used in all available operating systems, usually because they can be easily modified, easily applied on non trivial object, it is easy to process the whole list, ...

Linux kernel source code was analysed and it was observed that almost all important kernel structures are in some double linked list.

In certain situations, it may be difficult to reach the right list. For example when multiple levels of structures are used (this is the case of listing open files per process). Another problematic situation is when there are multiple double linked lists of a similar size and close to each other.

4.5 Dynamic pattern generation

Use of the "The Longest-Nearest" algorithm allows an analysis of an image at logical level, identifying structures through relations between them. This is great for general overview of the image content, but an 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.

The whole process can be automated if we allow certain level of errors in the 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 pattern generation can be done in these steps:

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

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 Linux kernel) will be different in "dead" processes. Automat is not able to discover, which bytes are important and which are not.

In theory, any granularity can be used for comparing patterns. In practical test, single bit granularity was found out too confusing, fixing even not important parts of patterns, and too difficult to match, once pattern is generated. Integer (4 or 8 bytes) granularity showed to be too big, omitting important pieces of information. Bytes granularity seems to be the best choice.

4.5.1 Generated patterns

For some types of the structures this method is excellent. The example of such structure is the BSD `proc` structure.

In the following examples, unknown bytes are represented by `??`, non-zero bytes are represented by `NZ` and other chars are values in hexadecimal representation.

The pattern created from a real BSD system analysed in subsection 7.2.3:

```
----- PROCESS PATTERN START -----
????????????NZNZ??NZNZNZNZNZNZNZd602c6c000000b000000000000000000
04000000??NZNZc100NZNZNZ0000000000NZNZNZ00NZNZc100000000????????
????0000????????????????NZ??NZNZ??0000000000000000??NZNZ????0010
01000000????000000000000??NZd3c1????????NZNZNZNZ????????????????
????????????????c902c6c0000043010000000000000000040000000????????
...
0000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000cefaefbe6b350c00
----- PROCESS PATTERN END -----
```

Such pattern have a few unique characteristics and it can be successfully matched even with a high error ratio.

Unfortunately, not all structures are suitable for this type of pattern generation. Some types of the structures may change too much or contain too common values, for example zeroes.

The example of such structure is the Linux `module` structure. Pattern created from a real Linux system:

```
----- MODULE PATTERN START -----
??????????????????????????????????????????????????????0000000000
??000000????????????????0000000000000000000000000000000000000000
...
????????????????????????????NZNZ??000000??NZ????????????NZ????00
000000000000000000000000NZ??????NZNZNZ????????????00000000000000
00000000??????????000000????????????????
----- MODULE PATTERN END -----
```

As can be seen, there are only a few characteristics available in the pattern. In addition, zeroes are quite common in the memory. The substitution check must be performed each time the zeroes are found, so matching such pattern is time consuming.

4.5.2 Pattern optimization

The problem of matching the pattern above is the complexity of such operation. Even when using optimal algorithm, an extended check must be performed each time the first matching value is found. Because of that, scanning the memory for the patterns starting with unknown values takes significantly longer time.

To reduce this problem, the pattern can be optimized before it is used. The obvious optimization is removing all unknown and non-zero values from the start of the pattern.

For example pattern `????NZNZ??NZNZNZd602c6c...` is optimized to `d602c6c...`

Situation gets complicated with the substitutions. With each added substitution (or "error"), the complexity of the search grows exponentially.

A simple but effective workaround used in Foriana package is the implementation of the parameter `--static` into the `bgrep` utility. The parameter `--static` allows treating first n bytes as immutable. This makes the actual use of the substitution much more rare event, for the cost that the first n bytes are not mutated and some structures may not be found. This "hack" is a compromise between practical usability and theoretical requirements.

4.5.3 Real results

		correct/false matches				
Structure	In List	0 err	5 err	20 err	40 err	80 err
BSD process	33	33/0	33/0	33/0	49/3	57/11
Linux process	34	34/0	35/0	48/9	60/20	N/A
Linux module	51	51/0	53/1020	N/A	N/A	N/A
Windows process	34	36/34	N/A	N/A	N/A	N/A

Table 4.1: Matching the generated patterns

The table 4.1 shows the result of the experiments with using the patterns of various structures. The results are only approximate. For proper results, all systems should run in exactly same conditions and with the same number of finished processes. 128 MiB and 256 Mib images of i386 architecture were used for the test. When the result is marked as N/A, matching the pattern took too much time or too many false positives were produced. All of the experiments were made with optimized pattern, as described earlier.

Dynamic patterns implemented in Foriana are effective for analysing the BSD process structure. The method can be used on Linux processes with a partial success but it seems to be ineffective for finding Linux modules and Windows processes.

We believe that further improvements of the results are possible. More effective search function would allow for the searches with more substitutions. More sophisticated pattern optimization may reduce the search time. Current implementation does not know the start and the end of an analyzed structure. Setting operating system specific constants may enlarge patterns and increase the chances for better characteristics. Creating patterns with single bit granularity may improve the results, too.

4.6 SLABS

SLABS are an interesting feature of some operating systems. Currently implemented in Solaris, Linux and others, SLABS provide a memory allocation mechanism intended to prevent memory fragmentation and thus increase performance.

Basic idea behind SLABS is that the most common objects need same amount of memory each time they are allocated. After some time these objects are deallocated, but it is evident that the same type of object will be used soon, so it may be more efficient to leave memory reserved for object of this type only.

Once a single structure is found, using SLABS may provide interesting information such as exact size of structure and addresses of other structures of same type. Investigator may get even structures of processes, that are almost completely destroyed.

Idea of SLABs was not implemented in this work, but may increase reliability of determination process in future.

Lately, a paper [27] investigating this idea was published. Focused on Linux SLAB and SLUB allocators, paper provide promising results. The amount of gathered information is extensive, and information are analysed in a systematic way.

4.7 Parsing source code

During Foriana development, an alternative approach to problem was tried. Linux kernel have publicly available source code. If the exact version of the source code and configuration options used for kernel compilation are known, the source code can be parsed to produce objects with known offsets. These objects can be then to find real structures in memory. This would allow skipping almost whole determination process (after finding VPT), and provide much better results. One of problems with parsing source code is that same code can produce really different binary code when various configuration options and compile optimizations are used.

Another method of analysis using kernel binary and system map is implemented sufficiently in patched `crash` utility [1].

After considering these facts, we decided to try a different approach based upon logical relations between kernel structures.

[1] http://volatilesystems.blogspot.com/2008/07/linux-memory-analysis-one-of-major.html

Chapter 5

Foriana

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

This is a perfect method when looking for “lost” information, like parts of deleted files, rootkit hiding somewhere inside the memory, ...

On the other hand, this approach have few disadvantages: It is relatively slow, can produce many false positives, (for example if user on computer investigated was reading about kernel structures) and mainly,it is extremely dependent on version of operating system.

Pattern-matching is working well for MS Windows, because there are few versions of the operating system and all are known to the 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 are reverse engineered.)

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

Foriana tries to solve this problem in more general way and do not insist on exact structure signatures. Approach is based on logical relations and on looking for lists of elements defined by some characteristics.

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

The exact process is described in next chapters.

5.1 Internal Structure

Foriana supports multiple architectures and operating systems. Mechanism for adding new architecture/system was made as simple as possible.

The architectures are stored in folder `src/arch/`, each one having its own subdirectory. Each architecture is identified by structure `arch_struct`.

```
struct arch_struct{
        // architecture name (string)
        char architecture[ARCH_NAME_LEN];

        // pointers to architecture specific functions
        read_linear_t read_linear;
        VPT_test_t VPT_test;
        user_VPT_test_t user_VPT_test;
        find_VPT_t find_VPT;
        arch_init_t init;
        arch_guess_t guess;

        // sizes used by architecture
        int pointer_size;
        int integer_size;
```

With following functions implemented here, rest of code is completely architecture independent.

- `init` This is initialization function that set function pointers and other settings.
- `guess` Function that tries to guess if analysed architecture match architecture represented by structure.
- `VPT_test` Function to test if provided address contain root of VPT structure.
- `user_VPT_test` Function to test if provided linear address contain root of VPT structure of some userspace process.

- `read_linear` Function for reading from linear/virtual addresses. Walk trough virtual page tables is performed using provided VPT address. This permits following linear pointers inside image, and solves memory fragmentation.

To integrate new architecture into foriana, it must be added into `Makefile.am` and registered in function `register_all_arch()`.

Operating system support is made in similar way. They are stored in folder `src/os/`, each one having its own subdirectory. Systems are identified by structure `os_struct`, which is bigger than `arch_struct` because OS support is more complicated. Systems must implement set of hooks (these of course can be empty) that are called from generic functions. Generic function tries to be completely independent from OS, but sometimes this is not possible.

Each operating system must implement following hooks:

- `hook_proc_determine` Hook called after first process string is found.
- `hook_proc_determine2` Hook called after process structure is analysed.
- `hook_verify_task_struct` Hook that perform verification of the process structure.
- `hook_verify_task_struct2` Hook call after verification. Needed for setting OS type.
- `hook_list_processes` Hook to list only implemented features.

The list of hooks is not definitive, and may be extended in the future.

To integrate new OS into foriana, it must be added into `Makefile.am` and registered in function `register_all_os()`.

5.2 Architecture support

Each architecture implements function `guess` that should return 0 if image looks like it was running on the same architecture. Guess functions are called one by one, and when first one succeed, the architecture is "guessed".

Currently, all implemented guess functions are based on interrupt vectors This pure heuristics because interrupt vectors does not have to be at the start of the memory, even though they usually are.

5.2.1 ARM

Functions were written according to ARM [7] hardware specifications.

For VPT_test function, the following pseudo-algorithm is used:

- The address of a top level page table must be 16 KiB aligned. This is required by architecture specification.
- For each entry in the table, bits 0 and 1 determine the page type. If the bits are set to 01, "course" page is used. If the bits are set to 10, "section" (or supersection) page is used.
- If course page is used, also bits 2, 3 and 4 should be zero. If section page is used, at least bit 19 should be zero. If subpages are enabled, even more bits should be zero.
- During the whole check errors and valid records are counted. If more then `ARM_VPT_MAGIC_ERR_FATAL` errors are found, the check is terminated immediately. There must be at least `ARM_VPT_MIN_REC` records in the table. The ratio of records and errors must be less then `ARM_VPT_MAGIC_ERR_RATIO`

Practical tests proved that ratio condition is a very good filter. ARM architecture have enough "forced bits" so the results are reliable, with only a few false positives.

5.2.2 x86_64

Functions were written according to Intel [6] and AMD [4] hardware specifications.

For VPT_test function, the following pseudo-algorithm is used:

- The address of a top level page table must be 4 KiB aligned. This is required by architecture specification.
- In x86_64 in all three modes (4 KB, 2 MB and 1 GB) bits number 7 and 8 in PML4E (Page Map Level 4 Entry) are marked as MBZ (must be zero). This is forced by the specification.
- Each not empty entry is checked if it "looks like" a valid PDPT (Page Directory Pointer Table). In x86_64 for 4 KB and 2 MB modes bits 7 and 8 in PDPE are marked as MBZ (must be zero). For 1GB mode, bit number 7 must be set to 1. This is quite weak but useful.

- During the whole check errors and valid records are counted. If more then `X86_64_VPT_MAGIC_ERR_FATAL` errors are found, the check is terminated immediately. There must be at least `X86_64_VPT_MIN_REC` records in the table. The ratio of records and errors must be less then `X86_64_VPT_MAGIC_ERR_RATIO`

Practical tests proved that ratio condition is a very good filter. X86_64 architecture have few "forced bits" but results are still acceptable.

5.2.3 i386

Functions were written according to Intel [6] and AMD [4] hardware specifications. PAE support is not implemented, as there is no evidence of used PAE in the image itself. Information about use of PAE is stored in processor registers that are not part of RAM image. Thanks to modularity, PAE support can be added later as i386-PAE architecture.

For VPT_test function, the following pseudo-algorithm is used:

- The address of a top level page table must be 4 KiB aligned. This is required by architecture specification.
- Test block of zeroes of length 3 KiB. This block can contain `I386_VPT_MAGIC_ZEROES_ERR` non zero values. Reason for this step is that the entries in first 3 KiB map linear addresses bellow 0xC0000000. This address is used on multiple systems (Linux, BSD) as a barrier between kernelspace and userspace addresses. Thus for system PGD first the 3 KiB should be empty. Observation is that this heuristic works for MS Windows systems too.
- If successful, skip the first 0xc00 bytes, and then try following test:
- Try first `I386_VPT_MAGIC_ENTRIES_CHECK` pointers, and check if all neighbour pointers masked by 0x6b equals each other. Tolerate `I386_VPT_MAGIC_ENTRIES_ERR` errors. If the test is true, this should be the page directory. Mask 0x6b represent bits IGNore, Acessed, Page-level WriteThrough, Read/Write and Present. At first look this heuristic does not make much sense. It is based on an assumption that if processor can set some bits in any way (IGN for example), it will set them the same way for all records. Also pages in kernelspace should (or could) be mapped mostly in the same way, with the same options.

Introduced heuristic based on the masked entries is definitely not a perfect test. During the development a lot of tests were made. The algorithm was changed several times and finally for values: `I386_VPT_MAGIC_ZEROES_ERR` = 40,
`I386_VPT_MAGIC_ENTRIES_CHECK` = 5, `I386_VPT_MAGIC_ENTRIES_ERR` = 1 produces correct results for all available testing images (over 30). As the algorithm looks weird, some alternative methods based on heuristics used for ARM and x86_64 architectures were tried. These methods were producing too much false positives because there are only very few "forced bits" in i386 specification.

An interesting situation was observed:

Often VPT of some userspace process is found instead of the kernel VPT. This could/should produce an incorrect results, but userspace VPT (resp. PDE) are cloned from the initial kernel PDE. Userspace processes cannot change the kernel memory mapped over 0xC0000000 (in Linux) so all information about kernel mapping stay intact. Therefore even if some other PDE is found, Foriana will highly probably produce the correct results. All the test images were analysed correctly when the second found VPT was used instead of the first one.

Translation from linear to physical address is implemented according to Intel [6] and AMD [4] hardware specifications.

`guess` function is based upon the structure of IDT. IDT is usually located at the beginning of the memory at the address 0x0. Unfortunately, this can be false. IDT can be on any different position and thus heuristics based upon IDT analysis can fail.

5.3 Operating systems support

5.3.1 Determination process

Before the determination of processes and modules can begin, first phase (architecture type and VPT address) have to be successful. 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.

In the next subsection, application of the"Longest-Nearest" algorithm is described. Algorithm work in generic way for any process structures. For supported OS, only differences are mentioned. This "Longest-Nearest" al-

gorithm should work even for completely unknown OS, when prerequisites listed in section 4.4 are satisfied.

5.3.2 Finding processes

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

- 1: Find the string `MAGIC_PROCESS` in the memory dump (via full-text search).
- 2: Check, if the found string is inside process structure. The string should represent the name variable. This verification is made by counting `head_list` structures, found maximally `TS_MAGIC_MAX_TRIES` bytes before the name string.

  `head_list` as defined in Linux kernel source code is a structure that contains pointers to previous and next `head_list` structures in the list. If structure contains a pointer to itself, the list contain only one `head_list`. See Figure 4.1 for visualisation.

  This was the first implemented method. It produces relatively good results but is not very reliable. 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`, the size of `SPREAD` lists around `FRONT_NUMBER` is compared, and the longest list is considered to be the list of process structures. (The list of all processes.)
- 3: Finally, the distance between the name variable and `head_list` is saved into `my_image` structure. This value will be later used each time some function needs it. Also, the whole `my_image` structure is saved to the cache file by default.

Supported operating systems:

5.3.3 Unknown system

Special type of operating system is named "Unknown". Whole guessing of operating system is implemented only here.

In the first implementation, searching for strings like “linux-2.6.22” or “Linux version 2.6.21.5” in the first pages of memory was used. However, this was far from producing reliable results. In the current version of the Foriana, the detection of an operating system is performed after the process structure is determined. This may look strange but it produces reliable results.

Heuristics for OS detection:

- If non-cyclic `list_heads` are used, the system is identified as BSD. All other supported systems use cyclic `list_heads`.
- If process `kthread` (or `kthreadd`) is found, the system is identified as Linux.
- If process `svchost.exe` is found, the system is identified as MS Windows.

Certainly, other heuristics for operating system detection can be implemented.

When looking for one of the processes, its name must be known.

By default, this is “`kthread`”, which was present in the most of analysed Linux memory images. (Sometimes this process is called “`kthreadd`”, but this is not problem for full-text search, when terminating zero is not used). Process “`init`” was not used, event through “`init`” should be the first process on all Linux systems by definition. Because string “`init`” is simply too common and creates too many false positives, search time is also significantly longer.

5.3.4 Linux

Linux kernel structures

In Linux, the address of VPT is exported as `pg_swapper_dir` or `swapper_pg_dir` symbol and can be found in `System.map` file if available.

`task_struct` is the main structure for every process. This structure is quite big, around 1 Kbyte in year 2010. Information important for logical analysis is shown in Figure 5.1.

On the other hand the `module` structure contains just few interesting information, as can be seen on Figure 5.2.

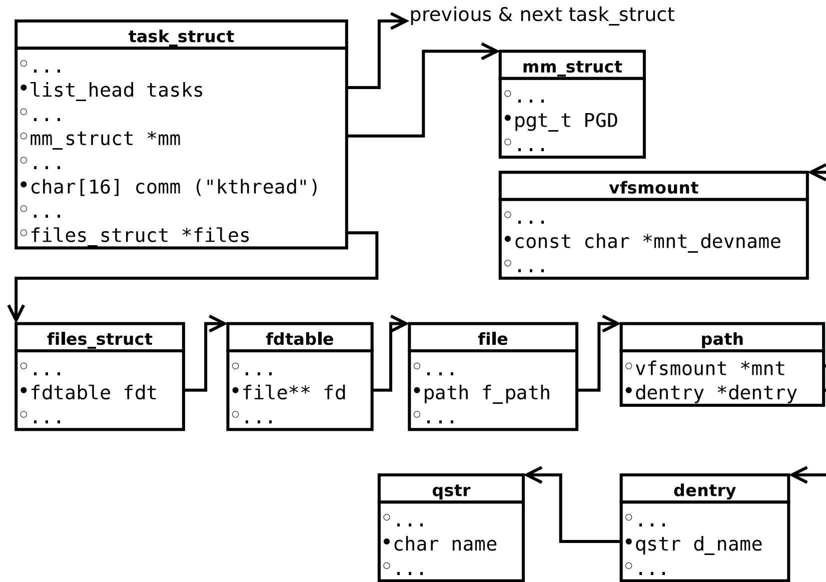


Figure 5.1: Kernel structures 1 (task_struct related)

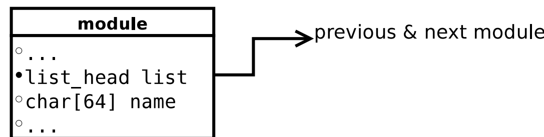


Figure 5.2: Kernel structures 2 (module related)

Userspace process

For Linux systems, at least one userspace process needs to be found. Such process is usually the "init" process that should be present on all Linux systems. Direct search for "init" is slow because this string is just too common. For each occurrence the verification needs to be performed. A side effect is a higher number false detections.

To improve the speed two step search is used: In the first step some, kernel space process is found. In the second step, the whole process list is searched until the process "init" is found.

Using described optimization significantly reduced the time needed for finding the process structure. Exact speed improvement depends on the position of the process structure in the dump file. The bigger part of the dump

file needs to be processed, the bigger speed improvement.

Finding VMA

After Linux userspace process is found, further determination can be performed. Firstly the distance to mm_struct needs to be found out.

```
struct mm_struct {
        struct vm_area_struct * mmap;  /* list of VMAs */
...
```

As can be seen, the first entry of `mm_struct` is `vm_area_struct`.

```
struct vm_area_struct {
  struct mm_struct * vm_mm;  // The address space we belong to.
  unsigned long vm_vm_area;  // Our vm_area address within vm_mm.
  unsigned long vm_mm_back;  // The first byte after our mm_back
                             // address within vm_mm.
 // linked list of VM areas per task, sorted by address
 struct vm_area_struct *vm_next;
```

The first pointer in `vm_area_struct` should point back to `mm_struct`, so verification of the right structure implies checking the linked list in a generic way.

If both pointers match, list size of `vm_area_struct` is counted, using address of the pointer `vm_next` and the offset from beginning of structure. The right one `vm_area_struct` should be in the longest list.

The described tests turned out to be sufficient, and offset to `mm_struct` structure can be determined.

Sometimes, user VMA list of user process can be too short. This happened few times during tests with init process. In such case, the determination will produce incorrect result, and different userspace process name should be used. This typically means just listing processes in the first step, picking one that should have a lot of VMAs (for example Xserver), and repeating the whole determination again.

Finding process PGD

Another step of determination is finding process PGD. PGD is required for reading the process virtual addresses, and for dumping process virtual address space into a file.

PGD should be stored inside the process mm_struct, so assuming that the offset to mm_struct is already known and that the function user_VPT_test is available, the search is straight-forward. Pointers inside mm_struct are tested, until one of them fulfills the conditions of the test. Of course, the determined offset is saved.

Finding modules

Please note that the module structures are not included in the memory image obtained by "soft" methods like using program dd and /dev/mem device. In Linux kernel, module structures are located in a memory area called "high memory" (or simply "highmem") that is not mapped into linear address space. To obtain the memory with module structures appropriate method must be used. For example firewire DMA access, coldboot method or "fmem" module.

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

Finding open files

Looking for open files (or file descriptors) using previously described method is quite difficult. From task_struct to the list of open files for this task_struct, 5 pointers pointing on 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. We tried to implement this feature into Foriana without success. It turned out to be too difficult to determine structure whose branching factor is so big. Source code have been removed from current release, and will be included once support is fully working. Heuristics based on SLAB listing might be more successful.

The whole situation is illustrated in Figure 5.1.

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 represents a problem, as the exact offsets of structures mentioned above

are unknown. The determination process can be simplified by fixing the structure offsets up to the `path` structure. Path structure contains only two pointers to structures: `vfsmount` and `dentry`. `vfsmount` contains char `mnt_devname`, the name of a device the file is stored on. It is relatively safe to assume that this name starts with the string "/dev/"". Using described optimization, the determination process can be completed in two steps, reducing the complexity of guessing by one level.

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

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

5.3.5 BSD

BSD is using non cyclic lists, and therefore few hooks have to be applied. Initial determination of process structure must be done using noncyclic version of verify_list function. Further for each tested list, the size of strings that would represent the names of processes need to be counted. This additional check is required, because there can occur (and often occurs) situation with multiple pointers pointing somewhere into the list. Without the check, the list would be identified correctly, but the determined offset would be false.

Once the process structure is determined, another hook must be done. Because the list is non-cyclic, the pointer to one of the list members needs to be set to the first list entry. Otherwise, the generic process listing function could not be used later.

5.3.6 Listing information

Listing modules

In the process of determination module structures, the address of one of the modules was stored into the cache. Same for offset between `list_head` and module name. Starting at the stored address and using the stored offset, the listing of `list_head` is relatively straight-forward. The only risk is getting into an endless loop.

To prevent loop, check for revisiting initial module is implemented. Even when checking first module occurrence, the endless loop can still occur (for

example sequence 1,2,3,4,5,4,5,4,5,4,5, ...).

With each printed module counter is increased. When counter reach some limit, a loop is detected and the listing is terminated.

For Linux, this limit is 65535 because there cannot be more modules on a standard Linux box. If needed, limit can be changed trough the header file.

Listing processes

The listing of processes is very similar to listing of modules. Once the offsets and one of the processes are read from cache, the whole `list_head` is processed and the name string is printed. The same protection against falling into infinite an loop is applied.

The loop limit is set to 65535 because there cannot be more processes on a standard Linux or MS Windows box. If needed, the limit can be changed trough the header file.

If the analysed system supports it, the memory mapped areas are printed for each process.

Chapter 6

Foriana - user manual

6.1 Installation

Installation from the source code:

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

If some problems with automatically generated configure file occur, command `autoreconf` should solve them.
Verify, that program was build successfully.

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

foriana 1.0.0 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
```

6.2 fmem module

`fmem` can be downloaded separately, but is also part of the `Foriana` package.

To dump the memory from a system the module needs to be compiled. Compilation on the target system is the best choice if possible. If not, the customization of Makefile and cross compilation must be done.

To compile the module on target system:

```
$ cd fmem-xxx
$ make
make -C /lib/modules/‘uname -r‘/build SUBDIRS=‘pwd‘ modules
...
  LD [M]  fmem.ko
```

Once the module is created, it can be inserted. Inserting the module will spawn device `/dev/fmem`.

```
# ./run.sh
Module: insmod fmem.ko a1=0xc0117844 : OK
Device: /dev/fmem
----Memory areas: -----
reg00: base=0x000000000 (    0MB), size= 2048MB, write-back
reg01: base=0x080000000 ( 2048MB), size= 1024MB, write-back
reg02: base=0x0bf700000 ( 3063MB), size=    1MB, uncachable
reg03: base=0x0bf800000 ( 3064MB), size=    8MB, uncachable
reg04: base=0x0d0000000 ( 3328MB), size=  256MB, write-combining
-----------------------
!!! Don’t forget add "count=" to dd !!!
```

With device `/dev/fmem` created, it is possible to perform the memory dump using your favourite tools. Usually the `dd` command that can be found on almost every Unix system is used.

For example:

```
# dd if=/dev/fmem of=dump1.dd bs=1MB count=3584
3584+0 records in
3584+0 records out
3489660928 bytes (3584.0 MB) copied, xxx.xxxx s, xx.x MB/s
```

For forensic purposes, it is better to send dumped data directly over the network, or dump them to a removable medium. Using local filesystem can cause potential data loss.

For example:

```
# dd if=/dev/fmem bs=1MB count=3584 | nc -v xxx.xxx.xx.xx 2345
```

on the analysed machine and

```
$ nc -v -l 2345 > dump1.dd
```

on the machine where the image should be stored.

6.3 Analysing image

To list the program options, type

```
$ foriana --help
```

In the ideal scenario, following command should perform whole analysis.

```
$ foriana -a image
```

Foriana will try to use all compiled-in functionalities. Please note that this is not the optimal way for every image. For example, if you do not have a dump of highmem, Foriana will spend a lot of time searching module structure that is not there. The memory can also contain inconsistencies, causing foriana to give you strange results. For example the structures can be falsely detected and the listing will print out pseudo-random strings.

A better approach is to analyse the image step by step. First, try to determine the offsets in structures.

```
$ foriana -d image
```

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

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

Sometimes the determination can fail for various reasons. For example the heuristic for finding VPT can fail. In that case try to specify architecture manually and to list all virtual page root tables in the image. Alternatively, you can get VPT address from some other source, such as System.map for Linux systems.

```
$ foriana --find-pgd --arch i386 image
```

VPT root can be specified manually. To increase the chance of a success even more, operating system can be specified too.

```
$ foriana --pgd address --arch i386 --system linux -d image
```

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

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

or just

```
$ foriana image
```

Foriana automatically checks for .image in the current directory, and use stored information if available.

Problem of this approach is that it lists only processes that were active during the dump, not processes that were terminated before. To list all the processes, you can generate a process pattern and then use `bgrep` utility included in the package to match all occurrences of the pattern in the dump.

```
$ foriana --list-processes --generate-process-pattern image
```

Because finished processes can be slightly different, it is good to allow few errors. When increasing number of errors, the speed is decreased very fast (exponentially). One "hack" for solving problem of speed is using `--solid` parameter, that treat first few bytes as immutable.

```
$ bgrep --solid 3 --err 10 process_pattern image
```

Previous steps are automated in the script `allproc.sh` included in the foriana package.

```
$ ./allproc.sh image
```

The easiest way to store gathered results is to use standard shell pipes. For example:

```
$ foriana --list-process --list-modules image > date-image
```

Once the process list is known, you can for example save the memory used by one of processes. (If analysed OS is supported)

```
$ foriana -p address --process_mem dump1 image
```

With the process memory saved into the file, the examination of the process can continue on much small data set, resulting in less false positives.

6.4 Results verification

The result verification can be done manually, by creating the pattern for the structure of interest, and then using a full-text search for that pattern.

For forensic purposes, fuzzy search using only substitutions should be used, as described earlier.

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

```
...
 process address (virt/phys: 0xC11E5598/0x1151058) name: udevd
 process address (virt/phys: 0xC7F82B18/0x11E5598) name: sshd
...
```

Automation of this process is implemented in the shell script `allproc.sh`.

Chapter 7

Evaluation

7.1 Testing images

The `fmem` tool was tested on several machines, running Linux 2.6 kernel on i386 or x64 architecture. No major problems were found and the program seems to produce the correct results.

The `Foriana` was tested on about 50 RAM images of Linux, MS Windows and BSD systems of various versions, various quality and running on i386, x86_64 or ARM architecture. Some of the images were obtained by "soft" methods, some via firewire, few were incomplete or damaged.

7.2 Obtained results

7.2.1 Linux

Successful run looks similar to the following example. Complete determination without any additional supplied information:

```
$ foriana -d x60-32bit-1G-kde-noclean
Analyzing file x60-32bit-1G-kde-noclean
Trying cache_file: ".x60-32bit-1G-kde-noclean": N/A (1).
Using magic_user_process: kdesktop
Dumped RAM size is 1015 MB.
Guessing architecture: OK, i386.
VPT address found at: 0x927000
- proc_determine: Looking for process "kthread".
```

```
- bsd_list_head_size: Susp. struct at 0xC191066C, loop (35).
- bsd_list_head_size: Susp. struct at 0xF694FB6C, loop (86).
- bsd_list_head_size: Susp. struct at 0xC1910618, loop (144).
- verify_task_struct: List size 145
Found valid process at 0x1915BBC.
Determine list-name offset as 292
Proc_determine: Assuming os is linux.(process name)
- lin_find_userspace_process: Looking for userspace process "init".
Found userspace process "init" at (virt/phys) 0xC19BD598/0x19BD6BC
- lin_find_process_VMA: vm_area_struct at 0xC19BD5A0, size: 9
- lin_find_process_VMA: vm_area_struct at 0xF6987F44, size: 15
Found userspace PGD at 0xF765B000
Determined mm_struct pgd offset as 36
Determination process of task_struct complete.
- linux_determine_module: Looking for module "yenta_socket".
- linux_determine_module: Starting at address 0x30000000.
- lin_verify_module: Found module struct at negative offset 8
Found valid module at 0x3D5B888C.
Determination process of module structure complete.
Determine open files: TODO
Saving determined information to cache file.
Clean exit;).
```

Listing processes:

```
$ foriana --list-processes  x60-32bit-1G-kde-noclean
--create-process pattern
nalyzing file x60-32bit-1G-kde-noclean
Trying cache_file: ".x60-32bit-1G-kde-noclean": ok.
User specified OS: linux
Dumped RAM size is 1015 MB.
Listing processes:
- proc addr (v/p: 0xF697F618/0x19BD598) name: kicker
-- mapped areas:
-- 0x08048000-0x08052000, (40 KB)
-- 0x08052000-0x08053000, (4 KB)
-- 0x08053000-0x08228000, (1 MB)
-- 0xB62F4000-0xB636D000, (484 KB)
-- 0xB636D000-0xB63A9000, (240 KB)
```

```
...
- proc addr (v/p: 0xC18FBAD8/0x18F5098) name: ksoftirqd/0
-- mapped areas:
-- none (kernelspace process?)
...
- proc addr (v/p: 0xF7D79098/0x37D85AD8) name: bash
-- mapped areas:
-- 0x08048000-0x080E8000, (640 KB)
-- 0x080E8000-0x080ED000, (20 KB)
...
Listing processes done.
Listed 145 processes (hopefully).
Clean exit;).
```

When analysing the Linux image, Foriana is able to find the process structures, and for each process its VPT (Virtual Page Table) and VMA (Virtual Memory Areas). Once these information are known, it is possible to dump process memory into the file, or read the data from the process virtual address space. Kernel modules support works fine and generating patterns for processes and modules too. While processes and modules support works successfully with older kernels (2.4, and even 2.2 (!)), VPT and VMA support works only for 2.6 kernel versions.

Once the determination process is complete, listing and reading linear addresses is done in fragments of seconds.

When an address of the process is known, it is easy to find process PGD, or even to dump the process memory.

```
$ foriana -p f7d79098 x60-32bit-1G-kde-noclean \
--process_mem bash_mem
Analyzing file x60-32bit-1G-kde-noclean
Trying cache_file: ".x60-32bit-1G-kde-noclean": ok.
Dumped RAM size is 1015 MB.
Detailed information about process at 0xF7D79098.
Process linear address(in kernel context): 0xF7D79098
Process physical address(offset in dump): 0x37D79098
Process name(short): "bash"
Process PGD address(linear): 0xF6956000
Process PGD address(physical): 0x36956000
Previous process: 0xF69C50D8 (phys: 0x37D79098)
```

```
Next process: 0xF7D85AD8 (phys: 0x37D7909C)
Process mapped areas (userspace addresses):
-- 0x08048000-0x080E8000, (640 KB)
-- 0x080E8000-0x080ED000, (20 KB)
-- 0x080ED000-0x08109000, (112 KB)
-- 0xB7D86000-0xB7D8F000, (36 KB)
...
-- 0xB7F38000-0xB7F3A000, (8 KB)
-- 0xBF8DE000-0xBF8F3000, (84 KB)
Process memory saved to file: bash_mem.
Process open files: TODO;)
```

The file `bash_mem` should now contain all memory mapped by the process bash.

Foriana have a capability of reading process virtual addresses when the physical address of process PGD is provide [1].

```
$ foriana --pgd-read-linear 0x375da5a4 --read-linear 0xdff97510 \
 --size-read-linear 200 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.
Reading 200 bytes from linear address 0xdff97510.
Read 200 bytes from 0xdff97510 (physical address 0x1ff97510):
Hex dump & printable strings:
dff97510: 10 75 f9 df 10 75 f9 df 00 00 00 00  .u...u..........
dff97520: 00 00 00 00 00 00 00 00 00 00 00 00  ................
dff97530: 01 00 00 00 00 e0 fb e5 02 00 00 00  ............@ @.
dff97540: 00 00 00 00 ff ff ff ff 80 00 00 00  ............w...
dff97550: 78 00 00 00 77 00 00 00 00 01 10 00  x...w......... .
dff97560: 00 00 00 00 00 00 00 00 00 c9 33 24  ..........3$.K.I
dff97570: 36 00 00 00 00 4b a8 49 36 00 00 00  6....K.I6....H..
dff97580: 00 00 00 00 00 00 00 00 00 00 00 00  ................
dff97590: 02 00 00 00 01 00 00 00 18 41 ab f6  .........A......
dff975a0: a0 75 f9 df a0 75 f9 df a8 75 f9 df  .u...u...u...u..
dff975b0: c0 9a 94 f6 c0 9a 94 f6 14 34 85 c0  .........4......
dff975c0: 00 00 00 00 11 00 00 00 00 00 00 00  ................
dff975d0: 00 00 00 00 14 29 00 00              .....)..
dff975d8: End of hexdump.
```

7.2.2 MS Windows

Example analysis of MS Windows image:

```
$ foriana -d --determine-skip-modules --list-processes -i \
 dfrws2005-physical-memory1.dmp
Analyzing file dfrws2005-physical-memory1.dmp
Ignoring cache file.
Dumped RAM size is 126 MB.
Guessing architecture: OK, i386.
VPT address found at: 0x30000
```

[1] 4 bytes from each line were cut to fit in page width

```
- proc_determine: Looking for process "kthread".
- proc_determine: "kthread" not found.
- proc_determine: second try: "svchost.exe"
- verify_task_struct: List size 34
Found valid windows process at 0x5BECA3C.
Proc_determine: assuming os is windows. (process name)
Determination process of task_struct complete.
Determine open files: TODO
Saving determined information to cache file.
Listing processes:
Disabling VMA listing, not implemented for windows kernels.
- proc addr (v/p: 0xFF27E934/0x7784874) name: svchost.exe
- proc addr (v/p: 0xFF16E4B4/0x5BEC934) name: nc.exe
...
- proc addr (v/p: 0xFF0DAE54/0x58CBD34) name: dd.exe
- proc addr (v/p: 0x80480648/0x414DE54) name:
- proc addr (v/p: 0xFF144114/0x480648)  name: helix.exe
...
- proc addr (v/p: 0xFF1BABD4/0x6A77114) name: Explorer.Exe
- proc addr (v/p: 0xFF18B4F4/0x3E35BD4) name: Apoint.exe
- proc addr (v/p: 0x8046AE04/0x2B844F4) name:
- proc addr (v/p: 0xFF29D174/0x46AE04)  name: services.exe
...
- proc addr (v/p: 0xFF22F874/0x2138734) name: Avconsol.exe
Listing processes done.
Listed 34 processes (hopefully).
Clean exit;).
```

Note the processes without a name at 0x80480648 and 0x8046ae04. These empty names are intentional. Comparing with the official results of the DFRWS challenge (2008) it is clear that these are special “Idle” and “System” processes.

These results shows that the initial idea of using logical connections between OS structures was correct, and that our approach can be really OS independent and produce correct results.

In the current version, Foriana works correctly only on approximately one third of analysed MS Windows images. Older MS Windows versions seems to be better analysed than newer versions. As MS Windows does not have the source code available and does not publish relevant documentation,

debugging is difficult and time consuming. Thus MS Windows support is more in a form of a "proof of the concept".

7.2.3 BSD

Several BSD systems were analysed. In contrast to Linux and MS Windows, BSD kernels use non-cyclic lists for their main structures. Thanks to special type of lists used, Foriana can identify BSD kernel even if the system is not specified by the user.

Currently, BSD support in Foriana handles listing process and creating patterns.

In principle, module listing should work as well. However, kernel modules are rarely used in BSD. This have two immediate impacts: Firstly, it is impossible to find the module present in the most of the systems. Secondly, it is extremely difficult to determine the module structure, since there are a only few entries in the list. These are reasons why module listing for BSD was not implemented.

Example analysis of a BSD system with the use of automatically generated patterns. First, determining the image:

```
$ foriana -d --determine-skip-modules freebsd_128M_x86 \
 --system bsd
Analyzing file freebsd_128M_x86
Trying cache_file: ".freebsd_128M_x86": N/A (1).
Using magic_process: init
User specified OS: bsd
Dumped RAM size is 128 MB.
Guessing architecture: OK, i386.
VPT address found at: 0x101E000
- proc_determine: Looking for process "init".
- verify_task_struct: List size 33
Found valid process at 0x1B6E4AC.
Determine list-name offset as 1196
Determination process of task_struct complete.
Determine open files: TODO
Saving determined information to cache file.
Clean exit;).
```

As can be seen, a list with 33 processes was found.

Then, script `allproc.sh` automatically creates patterns and tries to match them with the bgrep utility.

```
$ ./allproc.sh freebsd_128M_x86 "-e 35 -s 2"
Generating process pattern: OK,
 saved to ".freebsd_128M_x86_process_pattern"
Finding all occurrences: OK,
 saved to ".freebsd_128M_x86_processes"
Using offset: 516
Structure addr: String addr: xxd format
(chars after 0x00 can reveal fork history)
d97ea0:  0d980a4: 6b65 726e 656c 0000 0000 0000 0000 kernel......
1944000: 1944204: 675f 646f 776e 0000 0000 0000 0000 g_down......
19447f8: 19449fc: 696e 7472 006c 0000 0000 0000 0000 intr.l......
1944aa0: 1944ca4: 6964 6c65 006c 0000 0000 0000 0000 idle.l......
28e1d48: 28e1f4c: 6464 006e 006c 0000 0000 0000 0000 dd.n.l......
...
29c0000: 29c0204: 6d76 006e 006c 0000 0000 0000 0000 mv.n.l......
29c02a8: 29c04ac: 6d76 006e 006c 0000 0000 0000 0000 mv.n.l......
29c0550: 29c0754: 6d76 006e 006c 0000 0000 0000 0000 mv.n.l......
29c07f8: 29c09fc: 6d76 006e 006c 0000 0000 0000 0000 mv.n.l......
29c0aa0: 29c0ca4: 6a6f 7400 006c 0000 0000 0000 0000 jot..l......
29c0d48: 29c0f4c: 7368 006e 006c 0000 0000 0000 0000 sh.n.l......
29c1000: 29c1204: 6174 7275 6e00 0000 0000 0000 0000 atrun.......
29c12a8: 29c14ac: 6673 636b 5f75 6673 0000 0000 0000 fsck_ufs....
29c1550: 29c1754: 6765 7474 7900 0000 0000 0000 0000 getty.......
Found 43 processes.
```

As we can see, additional 10 already dead processes were found. Experiments with the number of allowed errors could even increase the number of found processes, or what was left from them.

From the process name, it can be discovered that processes “mv”, “dd” and some other are probably children of the “kernel” process and of the “cron” process.

7.2.4 Other systems

No Solaris (resp. OpenSolaris) memory dump was available for testing but according to the source code:

- Processes are in linked list.
- Processes name is inside process structure.
- Processes list is long enough.
- At least one process name is known.

Therefore all prerequisites of the "Longest-Nearest" algorithm have been met.

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

In a real world, determination process takes from a few a seconds up to tens of seconds for images several GiB in size. The exact time depends on the position of the structures we are looking for in the memory dump. In theory, the position can be almost completely random. Time is also influenced by user specified options. Providing the type of the architecture, the address of VPT and the type of the operating system can sometimes save significant amount of time.

7.3 Conclusion

Analysing the memory dump at a logical level proved to be a functional complement to the classic pattern-matching methods. The combination of the developed "Longest-Nearest" algorithm, the dynamic pattern generation and an error tolerant pattern-matching of binary patterns proved to be powerful analytic tool. This combinations produces results fully comparable to other methods. While range of supported systems is much wider and adaptation on new system require less effort.

Described ideas were implemented in program called Foriana. The program produces fast and reliable results, but the amount of gathered information is limited. Some of the new ideas were developed in this work. Operating system independent analysis was achieved. The software is implemented in an extensible fashion to easily allow adding support for more

architectures and operating systems. Currently supported architectures are i386, x86_64 and ARM. Operating system support was practically tested for Linux, BSD and MS Windows. In theory any operating system should be supported up to the level of listing processes, even though it may require changing some constants from header files. Automatic detection of architecture is based upon IDT heuristics. Automatic detection of operating system is based upon properties of the found OS structures.

For the purpose of this work a tool for obtaining physical RAM images from Linux systems was developed as a side product. Detection of cryptographic key was implemented sophisticated enough in [9] and [26], that were not available when choosing theme for this work. Therefore, this part of work is limited to providing shell script with practical patterns for password extraction.

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] *ARM Architecture Reference Manual*, ARM Limited, 2005.

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

[9] 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.

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

[11] Sherri Davidoff, *Cleartext Passwords in Linux Memory*,Massachusetts Institute of Technology, 2008.

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

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

[14] *Wikipedia*, http://www.wikipedia.org, 2010.

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

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

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

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

[19] Chris Betz, *Memparser*, http://www.dfrws.org/2005/challenge/memparser.shtml, 2005.

[20] Jurgen Pabel, *Frozen Cache*, http://frozencache.blogspot.com/, 2009.

[21] Udi Manber, Sun Wu, *Approximate grep*, ftp://ftp.cs.arizona.edu/agrep/, 1992.

[22] Suiche Matthieu, *Windd*, http://windd.msuiche.net/, 2009.

[23] *Crash*, http://people.redhat.com/anderson/, 2010.

[24] Volatile systems team *Linux Memory Forensics*, http://volatilesystems.blogspot.com/2008/07/linux-memory-analysis-one-of-major.html, 2008.

[25] *ForensicWiki*, http://www.forensicswiki.org/wiki/Tools:Memory_Imaging, 2009.

[26] Carsten Maartmann-Moe, *Interrogate*, http://sourceforge.net/projects/interrogate/, 2009.

[27] Andrew Case, Lodovico Marziale, Cris Neckar, Golden G. Richard, IIIc,*Treasure and tragedy in kmem_cache mining for live forensics investigation*, http://www.dfrws.org/2010/proceedings/2010-304.pdf, 2010.