Peeking into Pandora's Bochs – Instrumenting a Full System Emulator to Analyse Malicious Software

Lutz Böhne (lutz.boehne@redteam-pentesting.de)
RedTeam Pentesting GmbH
http://www.redteam-pentesting.de

October, 28th, Luxembourg
hack.lu 2009

About Myself

- Lutz Böhne
- Graduated in 2008 from RWTH Aachen University
- Now employed by RedTeam Pentesting GmbH
- Talk will cover some work I did for my Diploma Thesis

About RedTeam Pentesting

- Founded 2004 in Aachen, Germany
- Specialisation exclusively on penetration tests
- Worldwide realisation of penetration tests
- Research in the IT security field

Motivation

Motivation

- malware is an ever-increasing threat
- example: Symantec generated more than 1.6 million new malware signatures in 2008[1], a 165% increase over 2007
- automated analysis of malware a necessity due to large number of samples
- also: malware often runtime-packed
- lack of free and open source analysis tools

[1] http://www.symantec.com/business/theme.jsp?themeid=threatreport

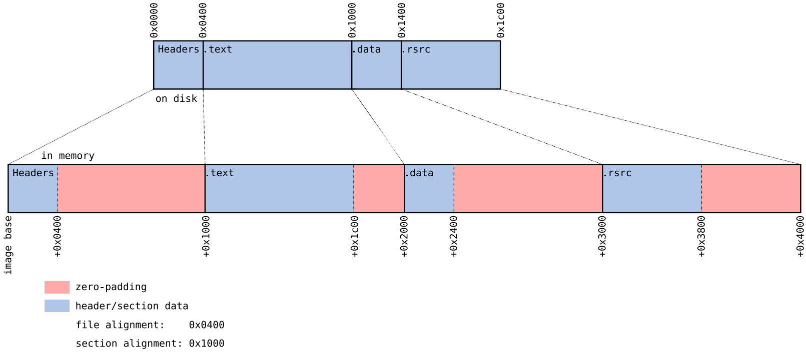

Figure: PE binaries - on disk and in memory

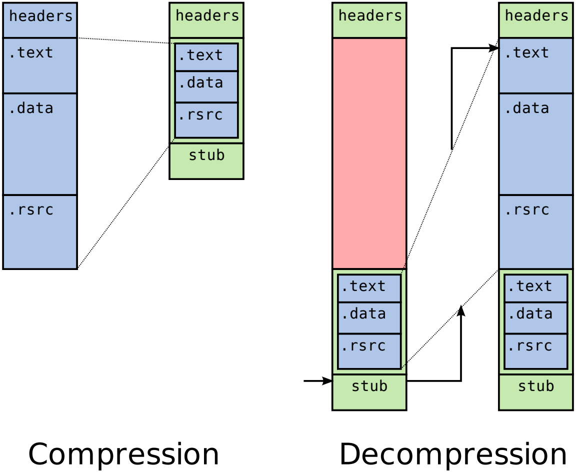

Figure: How runtime packers work

Runtime Packers - Compression

When packing a binary,

- the original code and data are packed or encrypted
- a small stub to unpack or decrypt the original code and data is added
- the entrypoint is set to the stub's first instruction
- often, the original import information is removed

Runtime Packers - Decompression

When executing a runtime-packed binary,

- first, the stub is executed to decompress or decrypt the original code and data
- second, the stub performs some tasks normally carried out by the PE loader, such as import resolution
- finally, the stub transfers control to the original code, for example by jumping to the so-called Original Entry Point (OEP)

Analysing runtime-packed executables

Static analysis

- code that is unpacked at runtime is typically not visible to static analysis methods
- static analysis of the unpacking stub is sometimes hampered by anti-disassembly techniques

Dynamic analysis

- some runtime-packers employ anti-debugging techniques to hamper dynamic anlysis

Weaknesses of typical runtime packers

- CPUs can only execute "plain text" code
- that code is "generated" at runtime by the unpacking stub and is at some point visible in memory
- typical approach: monitor execution of the unpacking stub and dump process memory whenever new code is being executed
- several projects deal with automated unpacking, but tools or source code are rarely released to the public.

Implementing an automated unpacker

Bochs

Pandora's Bochs is based on *Bochs*[2]

- FOSS PC Emulator
- written in C++
- built-in debugger
- supports instrumentation

[2] `http://bochs.sourceforge.net`

Pandora's Bochs

Pandora's Bochs originally designed as an automatic unpacker.
Challenges:

- unobtrusiveness
- awareness of guest-OS semantics
- OEP detection
- termination
- reconstruction of valid PE files

Instrumentation

Bochs can instrument certain events, for example

- ★ modification of the CR3 (Page Directory Base) register
- ★ memory accesses (writes)
- ★ execution of branch instructions

→ ideal for monitoring the unpacking process

Boch's Instrumentation Facilities

Bochs has many macros with inscrutable names. One might even go as far as to say that Bochs is macro infested.

- Bochs Developers Guide

Bochs's Instrumentation Facilities

Implemented as a set of macros that are used throughout the emulator source code, for example:

- `BX_INSTR_TLB_CNTRL(cpu_id, what, new_cr3)`
- `BX_INSTR_CNEAR_BRANCH_TAKEN(cpu_id, new_eip)`
  `BX_INSTR_CNEAR_BRANCH_NOT_TAKEN(cpu_id)`
  `BX_INSTR_UCNEAR_BRANCH(cpu_id, what, new_eip)`
  `BX_INSTR_FAR_BRANCH(cpu_id, what, new_cs, new_eip)`
- `BX_INSTR_LIN_ACCESS(cpu_id, lin, phy, len, rw)`

Instrumentation

I prefer Python to C++, therefore wrote a Python interface:

- Bochs is linked against the Python interpreter library
- Bochs provides its own “module” that allows anything running within the Python interpreter to query emulator state (for example memory, registers)
- at emulation startup, a module written in Python is imported
- instrumentation macros essentially call a set of functions exported by the Python module

Instrumentation

Instrument at two different levels of granularity:

- coarse-grained instrumentation: whenever the CR3 register is modified, determine whether the current process is of interest. Turn fine-grained instrumentation on or off accordingly.
- fine-grained instrumentation: if the current process is monitored,
  - record memory writes
  - monitor branches
    → check whether the branch target is modified memory

All processes and their corresponding PE images are logged to a database. So are (optionally) branches and writes.

Identifying Processes on x86

- Modern operating systems provide *each* process with its *own* 4-GB virtual address space
- x86 memory management unit uses page directories and page tables ("two-level paging") to translate virtual to physical memory addresses
- page directory base register (CR3) contains physical address of active page directory
  → active page directory identifies active virtual address space
  → every process identified by unique CR3 value

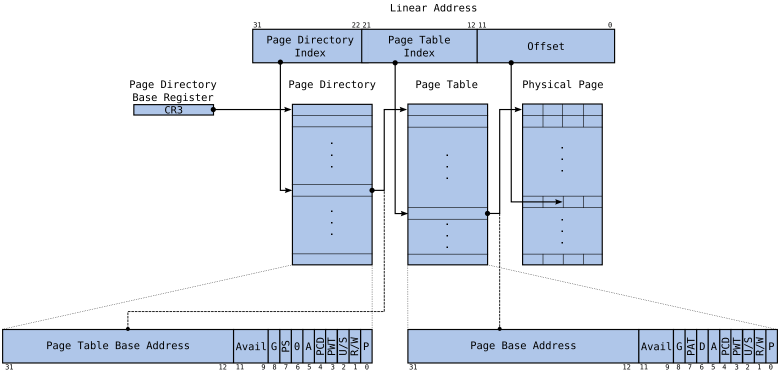

Figure: Paging on the x86 architecture

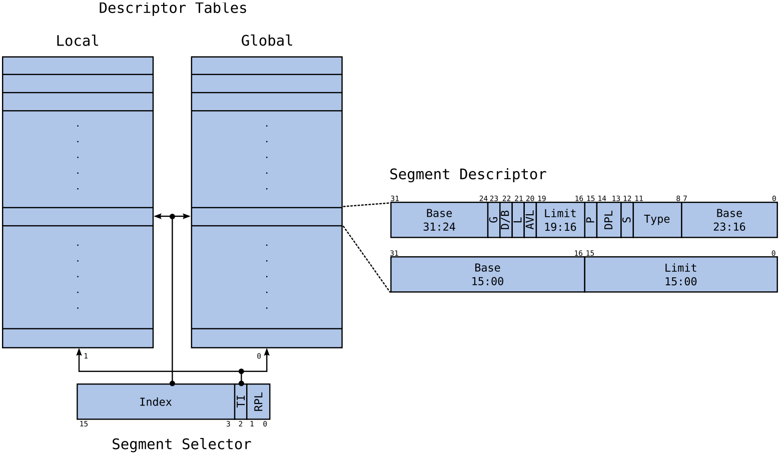

Figure: Segmentation on the x86 architecture

At fs:0 (segment descriptor 0x30) in kernel-mode:

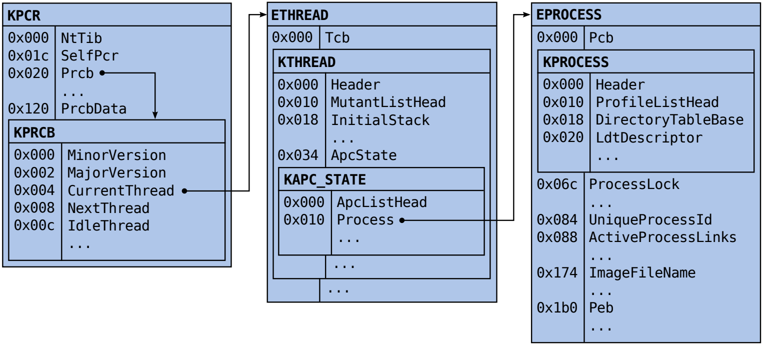

Figure: Identifying the current process in Windows (XP)

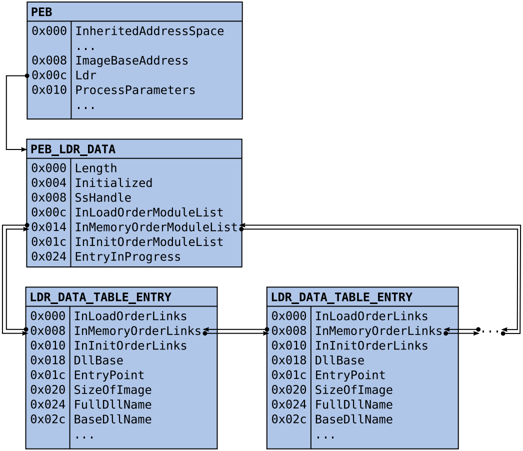

Figure: More information about the current process

Memory Dumps

Whenever a branch targets memory that was previously written to by the same process, that memory region is dumped to a database

- region to dump identified by VAD tree[3].
  - data structure in kernel space
  - contains information about a processes' virtual address space
    → stack, heap, memory-mapped files
- need to continue execution, in case there is more to unpack
  → memory around the current branch target is marked clean

[3] See Brendan Dolan-Gavitt. The VAD tree: A process-eye view of physical memory. *Digital Investigation*, Volume 4, Supplement 1:62–64, September 2007.

OEP Detection

Branches to modified memory regions are OEP candidates

Limitations:

- only the *first* branch to such a memory region
- only branch targets within the original process image
- last candidate is the most likely → when to stop monitoring?

Termination

It is undecideable whether new code will be unpacked
→ when to stop unpacking?

- Fixed timeout can guarantee termination
- Before that timeout, track "innovation". A process shows innovation, if
  - there are many memory writes per unique branch target
  - new DLLs appear in the process image
  - modified memory is executed
  - an API function not called before is called
  - stop emulation after a configurable number of task switches where no monitored process showed innovation

Reconstructing a valid PE file from a memory image

- copy original headers to the end of the file and zero-pad them
- make "PE Signature Offset" point to the copied headers
- set "Entry Point" to the detected OEP
- set "File Alignment" to "Section Alignment" and correct all section headers
- append new section header for a new section named `.pandora` that contains the copied headers
- reconstruct Imports

Import Reconstruction

Import Address Table (IAT):

- on-disk: describes which library functions to resolve
- normally filled by the PE loader with of addresses of library functions
- in packed executables, typically filled by the unpacker stub

Reconstruction:

- find all branches from within the process image to a DLL
- disassemble the branch instruction
  $\rightarrow$ operands of indirect jumps are potentially within an IAT
- inspect potential IAT, and try to resolve symbols
  $\rightarrow$ reconstruct IAT and corresponding headers

API Call Tracing

API Call tracing yields information about a malware sample's behaviour

- branch instructions are instrumented anyway
  $\rightarrow$ little overhead to check if branch target is an API function
- need to know API function prototype to determine stack layout for API call arguments

GCC-XML[4]

There is one open-source C++ parser, the C++ front-end to GCC, which is currently able to deal with the language in its entirety. The purpose of the GCC-XML extension is to generate an XML description of a C++ program from GCC's internal representation. Since XML is easy to parse, other development tools will be able to work with C++ programs without the burden of a complicated C++ parser.

[4] `http://www.gccxml.org`

GCC-XML Output

```
<Function id="_9749" name="GetProcAddress" returns="_9622"
    context="_1" location="f2:2610" file="f2" line="2610"
    extern="1" attributes="dllimport __stdcall__">
  <Argument name="hModule" type="_8702" ... />
  <Argument name="lpProcName" type="_6677" ... />
</Function>

<Typedef id="_6677" name="LPCSTR" type="_2864" ... />
<PointerType id="_2864" type="_294c" size="32" align="32"/>
<CvQualifiedType id="_294c" type="_294" const="1"/>
<Typedef id="_294" name="CHAR" type="_293" ... />
<FundamentalType id="_293" name="char" size="8" align="8"/>
```

pygccxml[5] to the rescue

Using pygccxml, we can use GCC-XML's output from python, to

- query functions by name
- inspect function prototypes
- determine the stack layout for function calls

Current implementation

- handles character strings and integers
- doesn't know anything about input and output parameters
- doesn't handle return values
- has basic support for handling stolen bytes

[5] http://www.language-binding.net/pygccxml/pygccxml.html

Stolen Bytes

A method employed by some executable protectors. Basic idea:

- copy first N instructions of an API function to someplace else
- append a jump to the (N+1)th instruction
- modify import information to call the copied bytes

Basic countermeasures:

- if a branch target is not an exported symbol, use the one with the next-smallest address
- disassemble instruction stream from there to the branch target
- keep track of and adjust for instructions that modify ESP

Results

Results - Criteria

- Unpacking Time
- OEP detection
- Does the unpacked code match the original code (`.text` section)
- Could a valid and executable PE image be reconstructed

Results - Synthetic Samples

- Generated by packing two different binaries, Notepad (68kB) und Wget (732kB)
- 30 different runtime packers, using their *default* configuration
- Only 20 packed Notepad samples would execute

Results - Synthetic Samples

- hidden code could be extracted from almost all samples
- OEP detected correctly for 80% of all samples
- valid, *executable* PE images could be reconstructed for 58% of all samples
- major obstacle to reconstruction: modification of the original code by a packer
- unpacking times from several minutes to an hour or more
  $\rightarrow$ could be somewhat improved by logging less extensively

Malware Samples

- 409 samples, collected over the course of one month by the RWTH Aachen Honeynet
- 379 known malware (ClamAV), 239 runtime-packed(PEiD)
- 361 started execution and 343 executed modified memory
- average run time was 7 minutes and 21 seconds
- Dr. Whatson started in 152 cases
- analysis indicates most of them could be unpacked correctly
- need to do more real-world testing

Conclusion

Conclusion and Future Work

- plain unpacking seems to work fairly well, appears to be largely immune to anti-debugging techniques
- API call tracing not heavily tested
  - results so far look promising

  → future work: track return values, output parameters
- major obstacle to reconstruction of valid PE images:
  - executable protectors that modify the original code
  - examples: stolen bytes, API call/entry point obfuscation

  → need better, interactive tools?
- emulation speed is subpar, some compatibility issues

  → use different emulator/virtualizer?

  → profile and optimise instrumentation code

Additional Information

- My Thesis is available at https://0x0badc0.de/PandorasBochs.pdf
- Git repository:
  - mirrors the Bochs CVS repository
  - Pandora's Bochs commited into a branch pandoras_bochs
  - moving target, used more as a version-controlled backup
  - clone from git://0x0badc0.de/home/repo/git/bochs
  - Gitweb at https://0x0badc0.de/gitweb?p=bochs/.git
- Slides will be made available at http://www.redteam-pentesting.de

Questions?

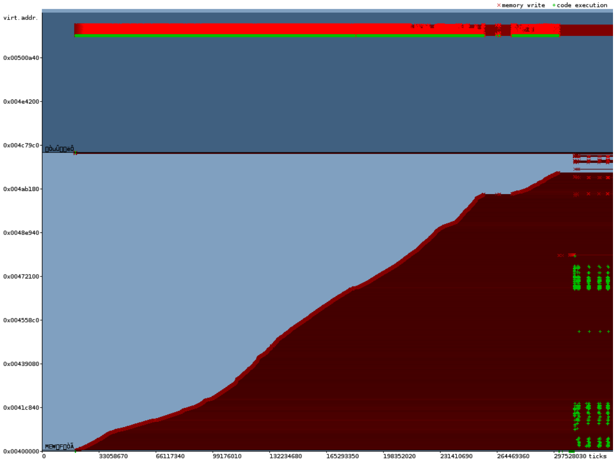

Figure: Unpacking MEW11SE 1.2

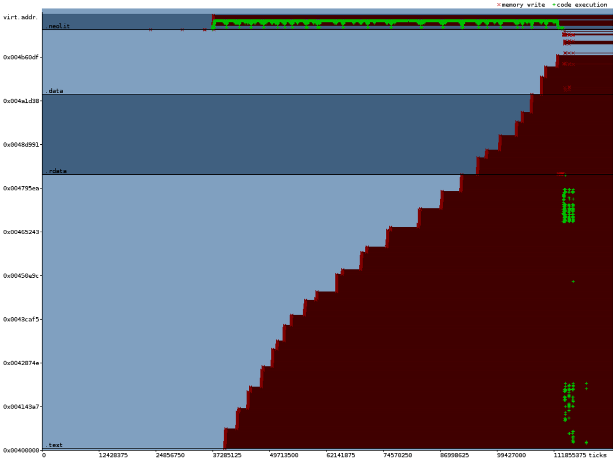

Figure: Unpacking Neolite 2.0

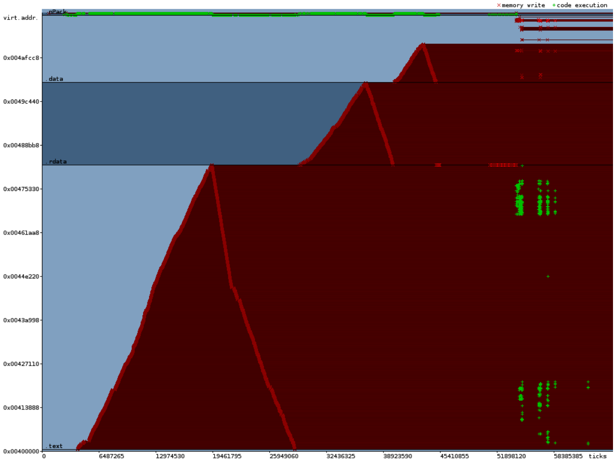

Figure: Unpacking nPack 1.1300beta

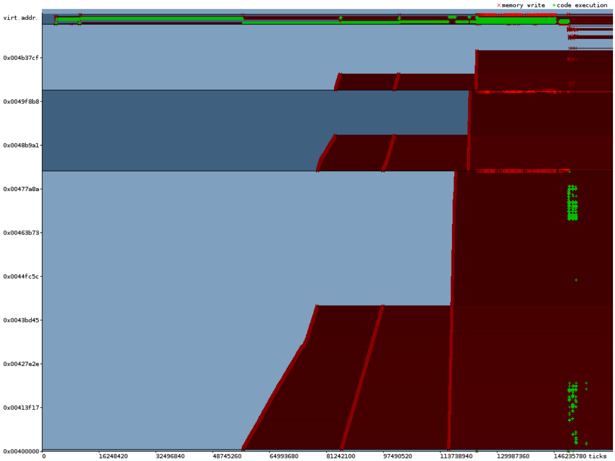

Figure: Unpacking PESpin 1.304

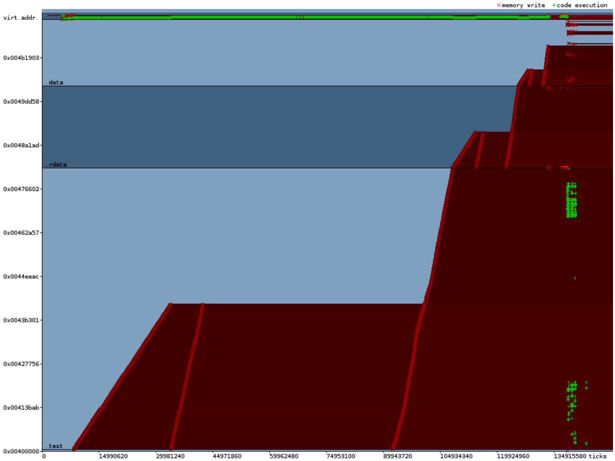

Figure: Unpacking tELock 0.98

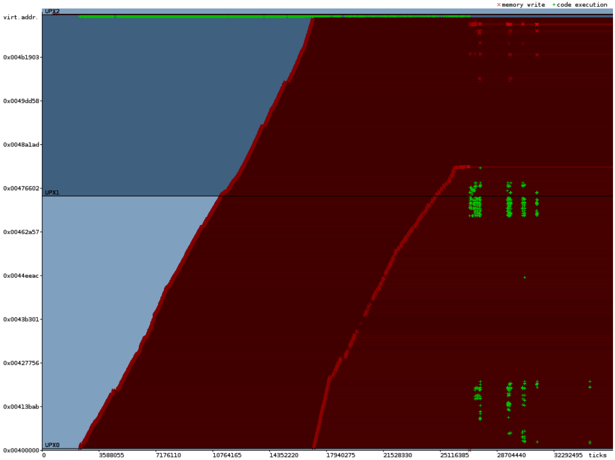

Figure: Unpacking UPX 3.01