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

April 9th 2010, Paris
Hackito Ergo Sum 2010

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

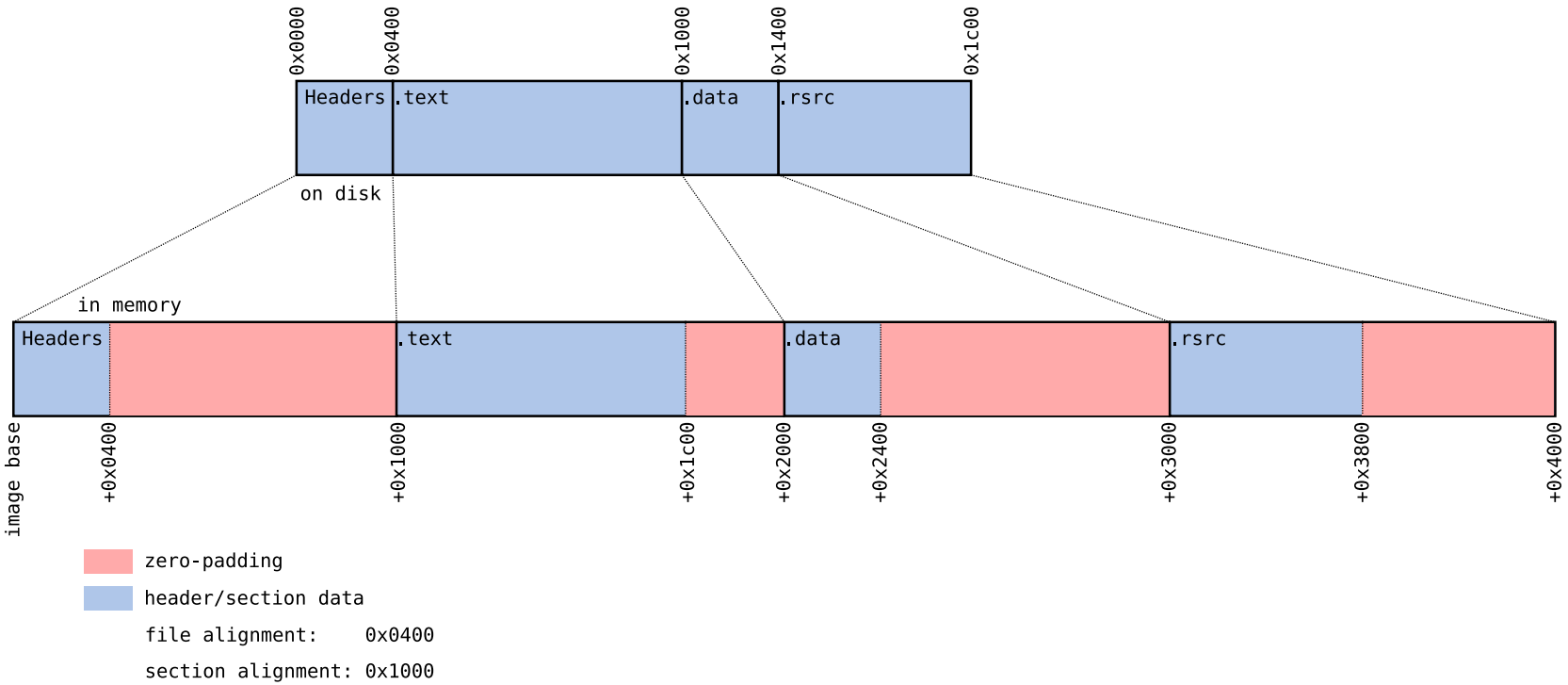
- ★ Founded 2004 in Aachen, Germany
- ★ Specialisation exclusively on penetration tests
- ★ Worldwide realisation of penetration tests
- ★ Research in the IT security field

Motivation

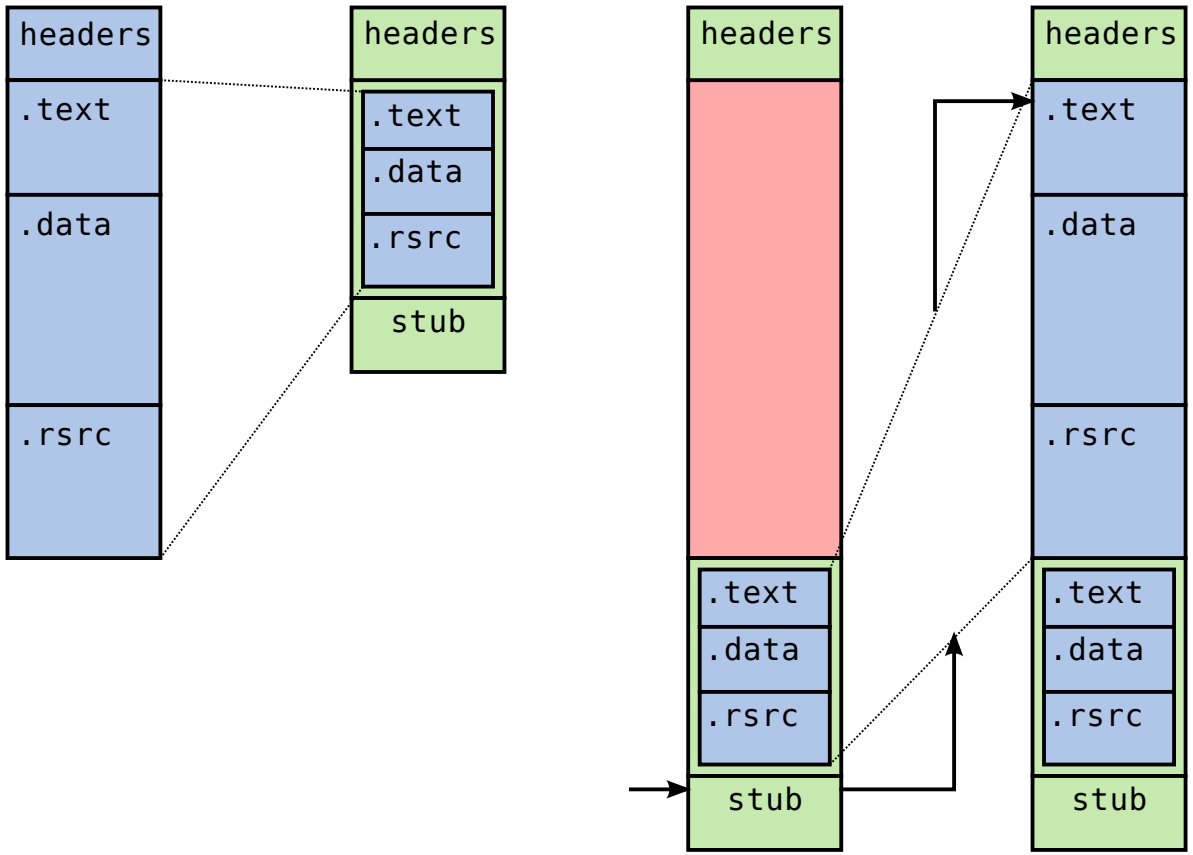
Malware

- ★ malware is an ever-increasing threat
- ★ high number of samples → automated analysis desirable
- ★ lack of free and open source analysis tools
- ★ malware is often runtime-packed

PE Files


0x0000
0x0400
0x1000
0x1400
0x1c00
Headers
.text
.data
.rsrc
on disk
in memory
Headers
.text
.data
.rsrc
image base
+0x0400
+0x1000
+0x1c00
+0x2000
+0x2400
+0x3000
+0x3800
+0x4000
zero-padding
header/section data
file alignment: 0x0400
section alignment: 0x1000

Runtime Packers



Compression Decompression

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

Pandora's Bochs

Pandora's Bochs is an automated unpacker based on *Bochs*[1]
Challenges:

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

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

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:
  - monitor virtual address space changes
  - determine current process
  - switch fine-grained instrumentation on or off
- fine-grained instrumentation:
  - record memory writes
  - monitor branches
    $\rightarrow$ check whether the branch target is modified memory

Instrumentation

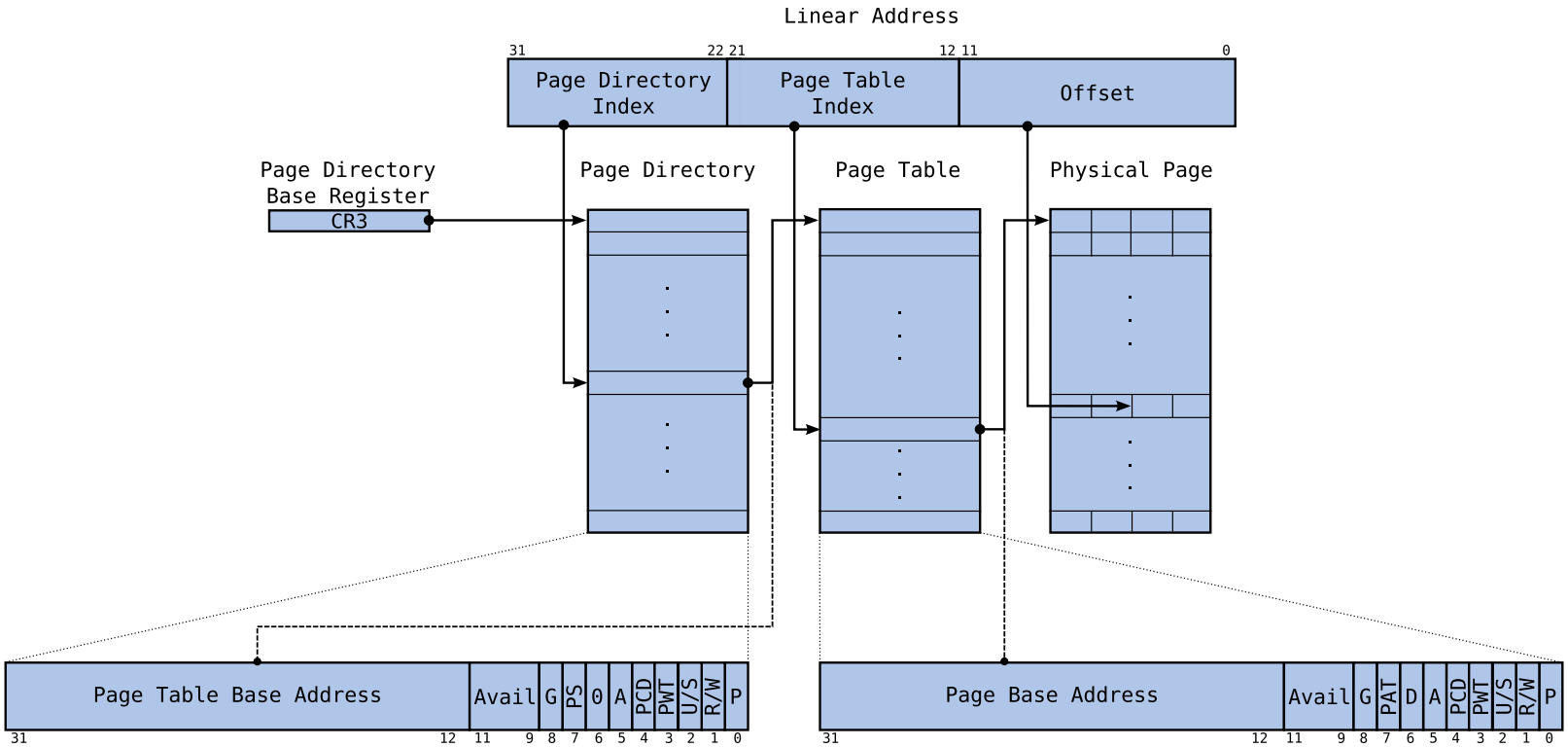
Instrument at two different levels of granularity:

- coarse-grained instrumentation:
  - **monitor virtual address space changes**
  - determine current process
  - switch fine-grained instrumentation on or off
- fine-grained instrumentation:
  - record memory writes
  - monitor branches
    $\rightarrow$ check whether the branch target is modified memory

Paging on the x86 architecture

- ★ Modern operating systems provide *each* process with its *own* 4-GB virtual address space
- ★ x86 memory management unit uses page directories and page tables 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

Paging on the x86 architecture

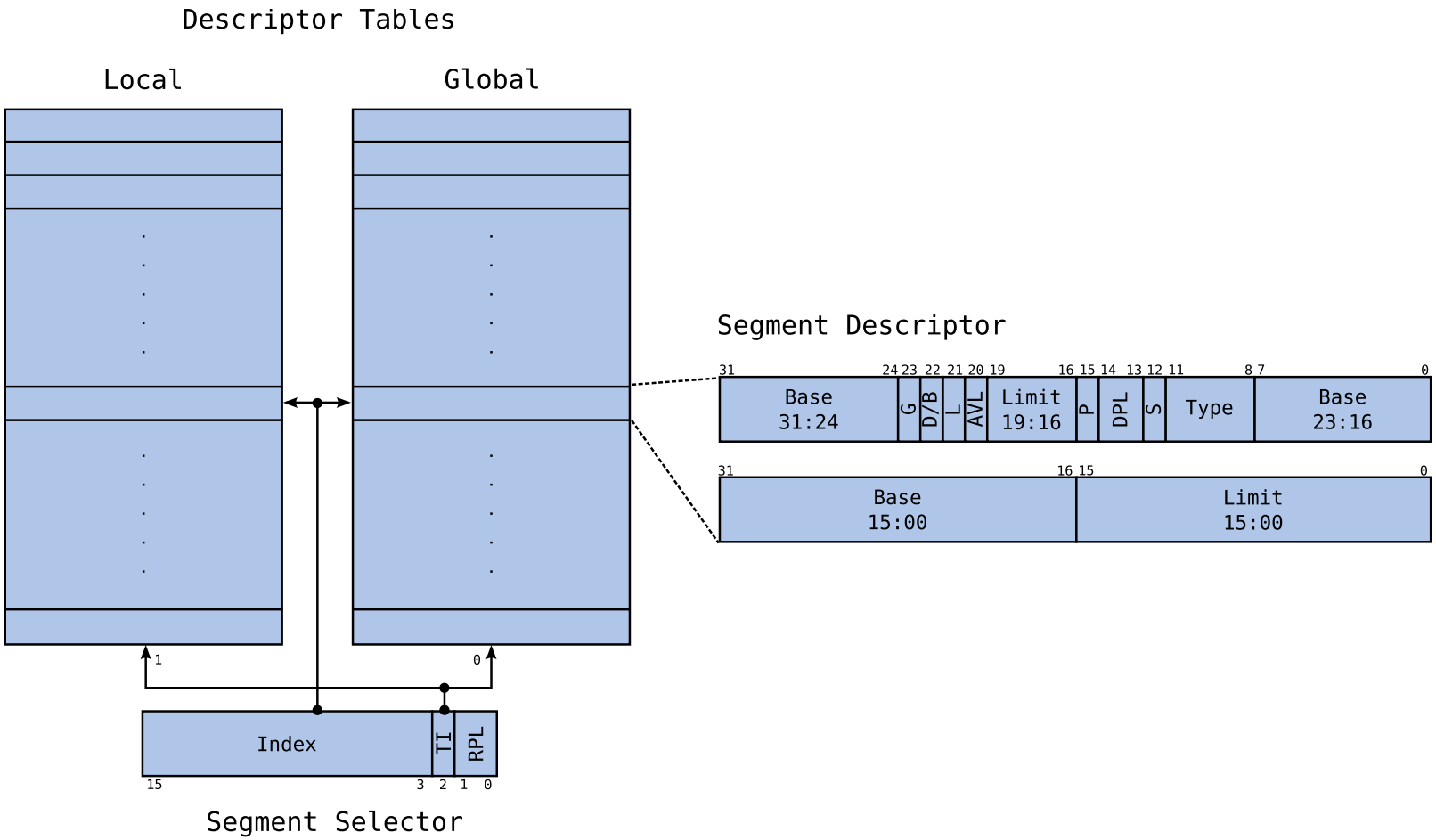


Instrumentation

Instrument at two different levels of granularity:

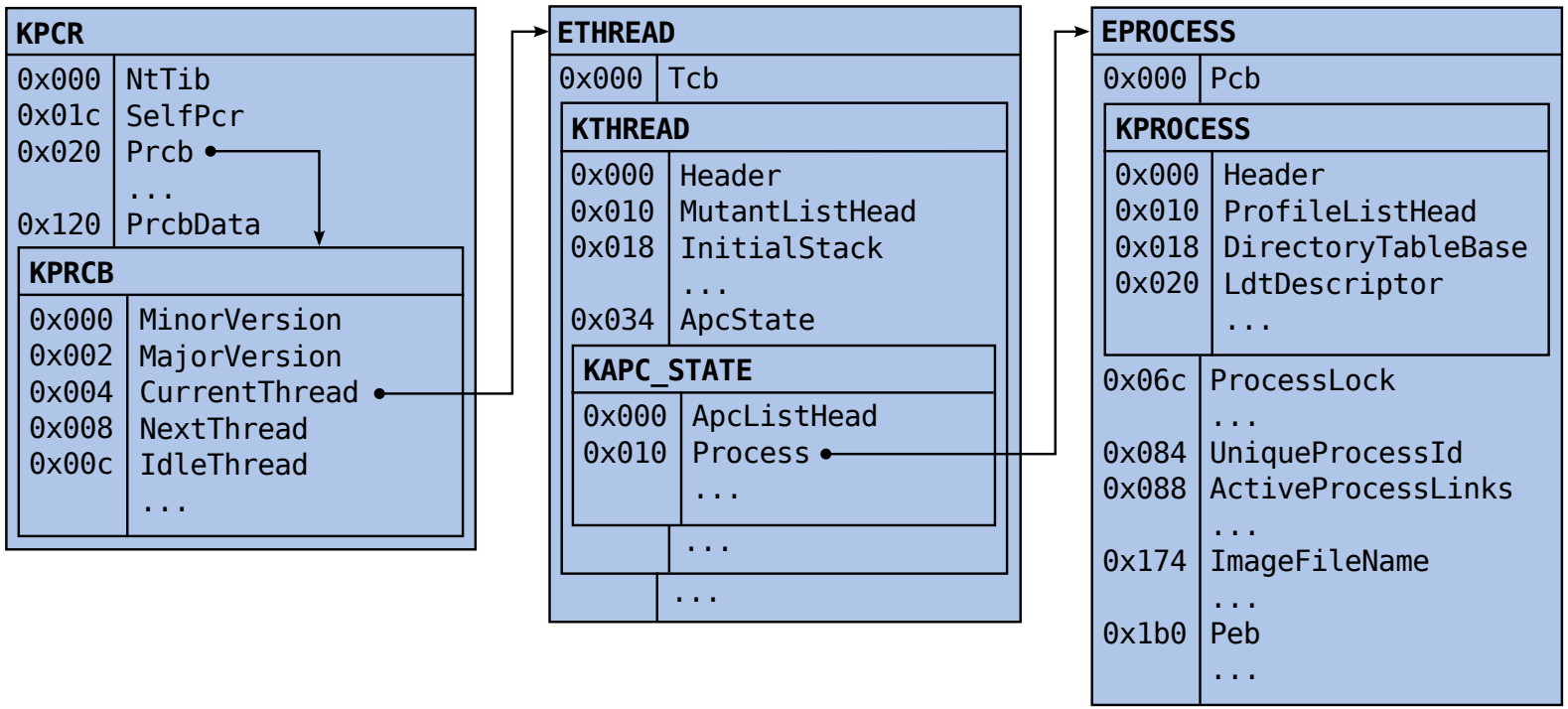
- coarse-grained instrumentation:
  - monitor virtual address space changes
  - **determine current process**
  - switch fine-grained instrumentation on or off
- fine-grained instrumentation:
  - record memory writes
  - monitor branches
    $\rightarrow$ check whether the branch target is modified memory

Segmentation on the x86 architecture

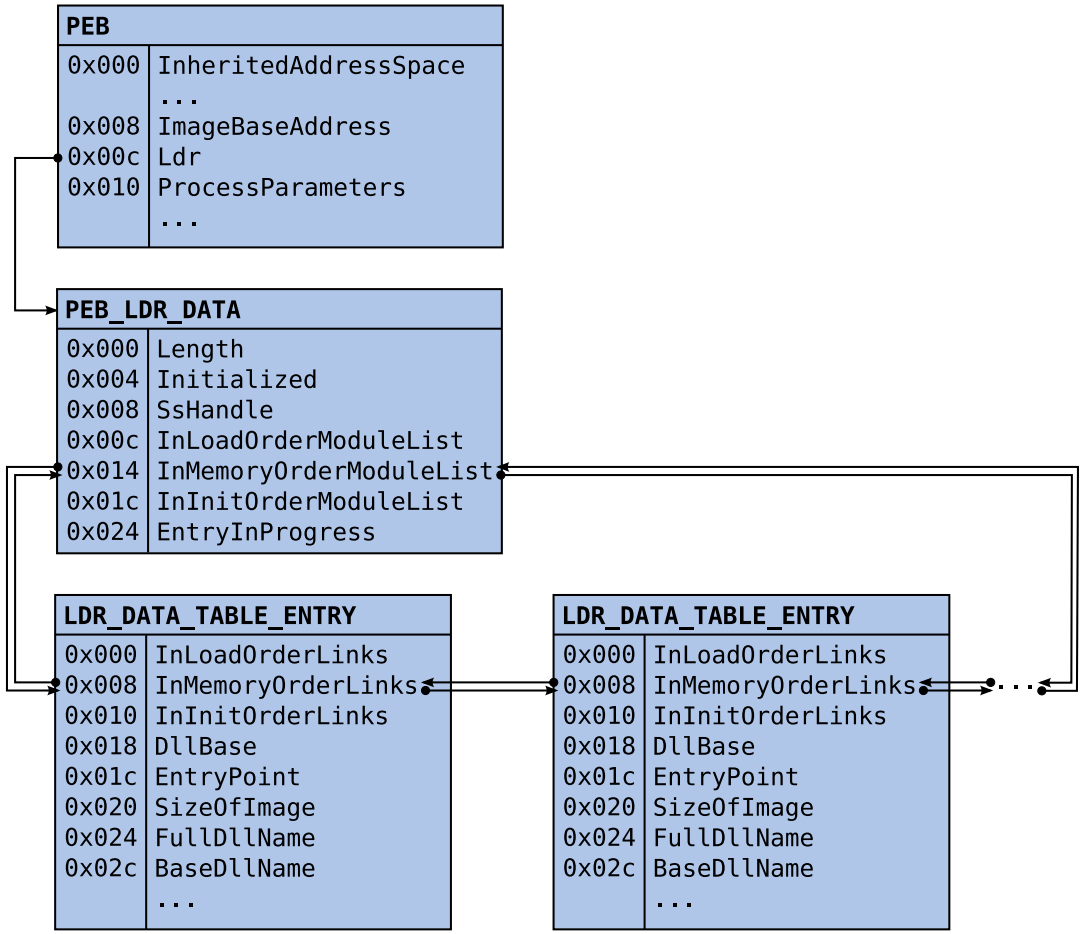


Identifying the current process in Windows (XP)

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



More information about the current process



Instrumentation

Instrument at two different levels of granularity:

- coarse-grained instrumentation:
  - monitor virtual address space changes
  - determine current process
  - **switch fine-grained instrumentation on or off**
- fine-grained instrumentation:
  - record memory writes
  - monitor branches
    $\rightarrow$ check whether the branch target is modified memory

Instrumentation

Instrument at two different levels of granularity:

- coarse-grained instrumentation:
  - monitor virtual address space changes
  - determine current process
  - switch fine-grained instrumentation on or off
- **fine-grained instrumentation:**
  - record memory writes
  - monitor branches
    $\rightarrow$ check whether the branch target is modified memory

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[2].
  - ★ 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

[2] 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

Will more layers get unpacked?

→ undecidable in the general case

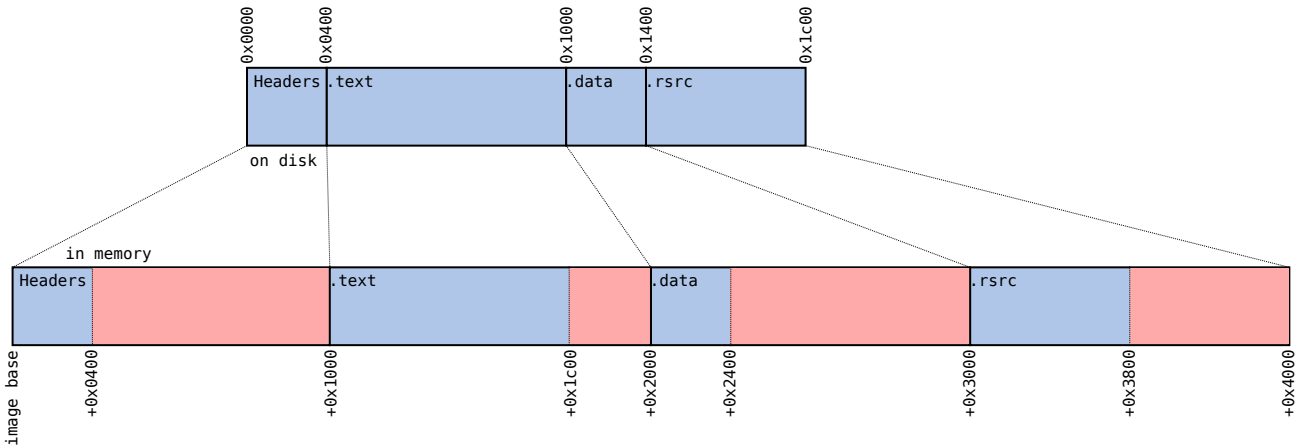
→ when to stop unpacking?

- ★ termination guaranteed by (configurable) time limit
- ★ until then: determine “innovation” of a process
- ★ stop emulation when no monitored process showed innovation for a certain amount of time

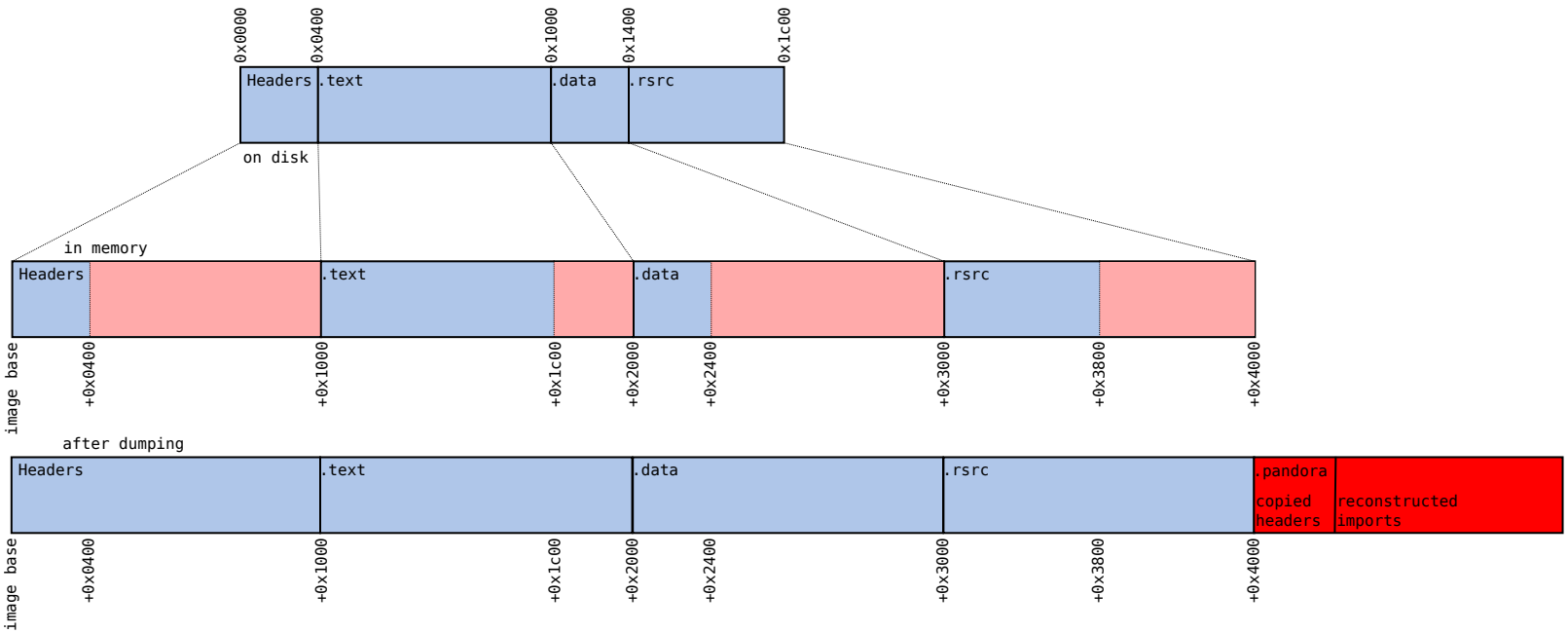
Reconstructing a valid PE file from a memory image

- ★ 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
- ★ adjust ‘File Alignment” and correct all section headers
- ★ append new section `.pandora`
- ★ reconstruct Imports

Reconstruction



Reconstruction



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 reconstructed at runtime by the unpacker stub

Reconstruction:

- ★ find indirect branches from within the process image to a DLL
- ★ indirect operands are potentially within an IAT
- ★ check potential IAT for validity and reconstruct

API Call Tracing

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

- ★ branch instructions are instrumented anyway
  → 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[3]

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.

[3] `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[4] 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
- ★ outputs arguments (and return value) once on call and once on return

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

Results

Results - Criteria

Performance was evaluated on a set of synthetic samples and on unknown malware. 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

- ★ Packed Notepad (68kB) und Wget (732kB)
- ★ 30 different runtime packers, using their *default* configuration
- ★ 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

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
- ★ analysis indicates most of them could be unpacked correctly

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
- ★ 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?

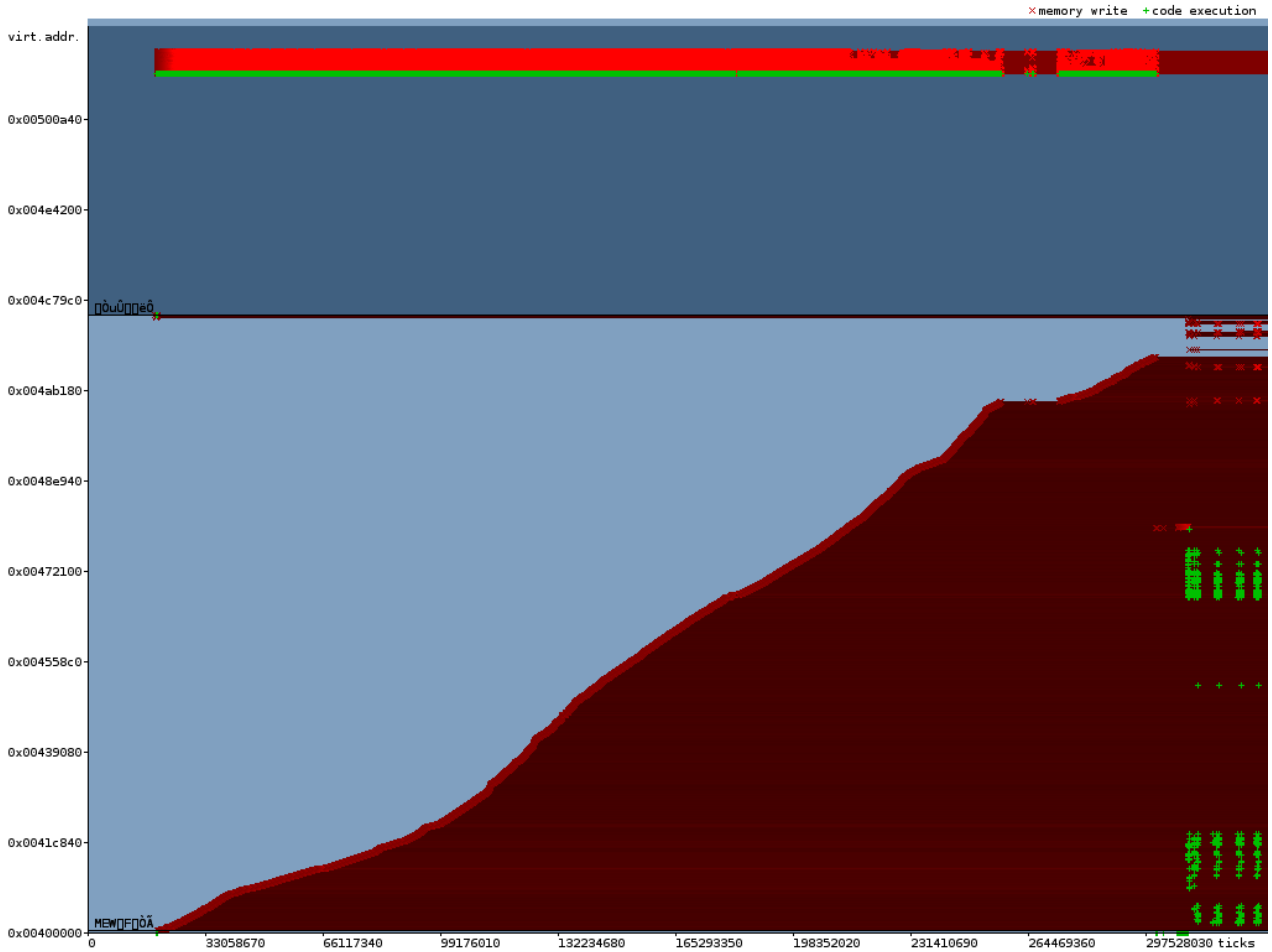


Figure: Unpacking MEW11SE 1.2

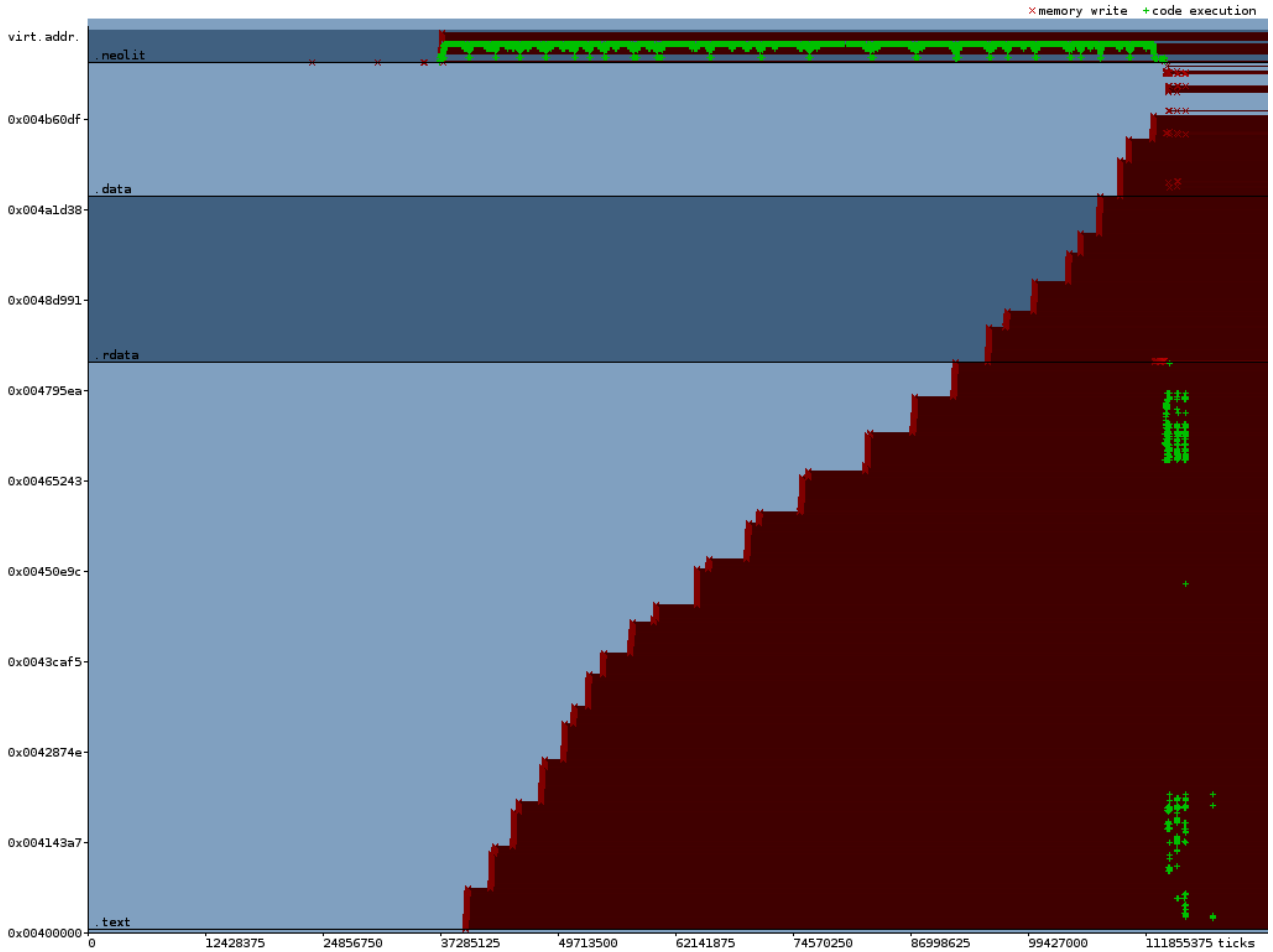


Figure: Unpacking Neolite 2.0

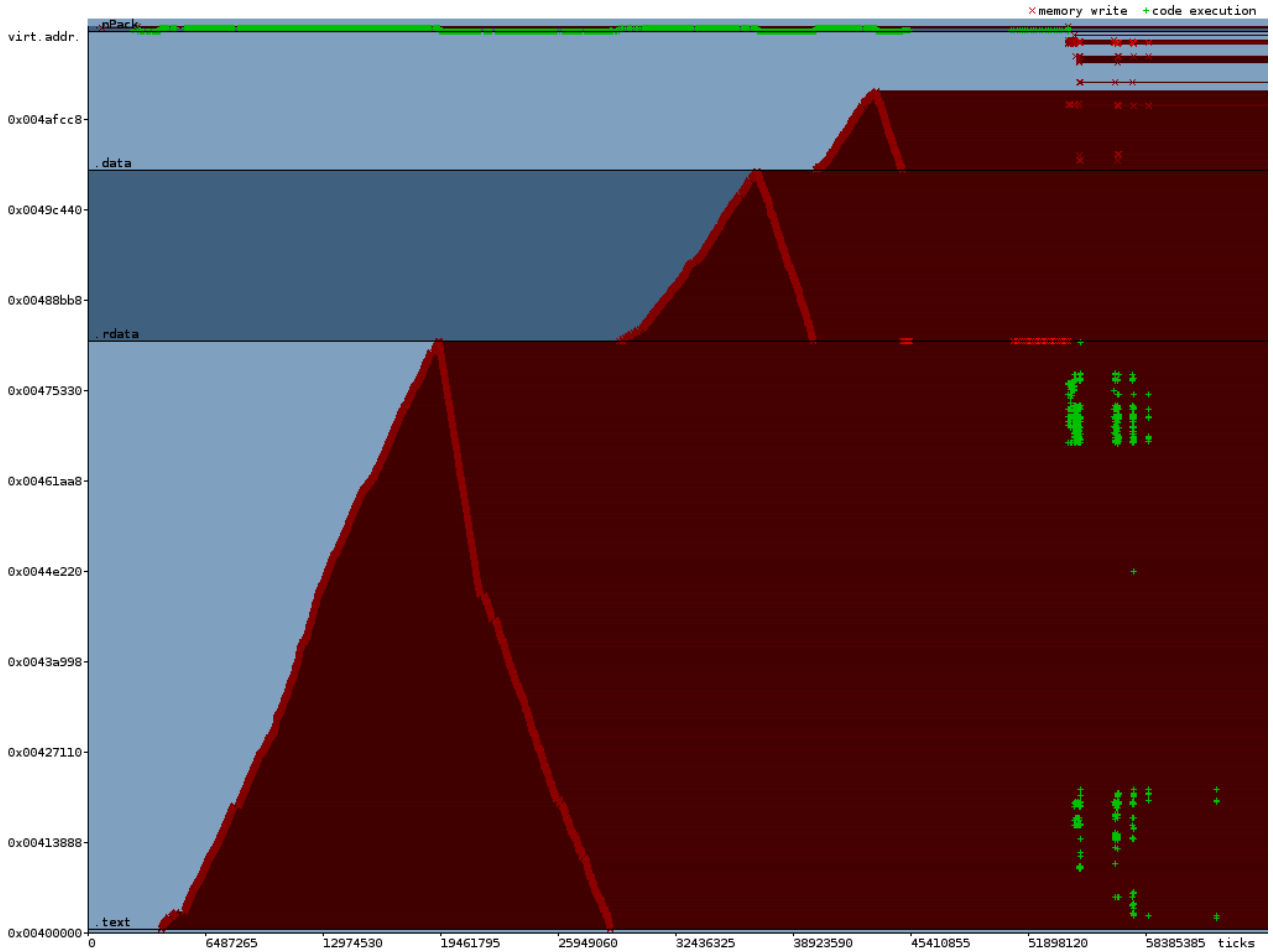


Figure: Unpacking nPack 1.1300beta

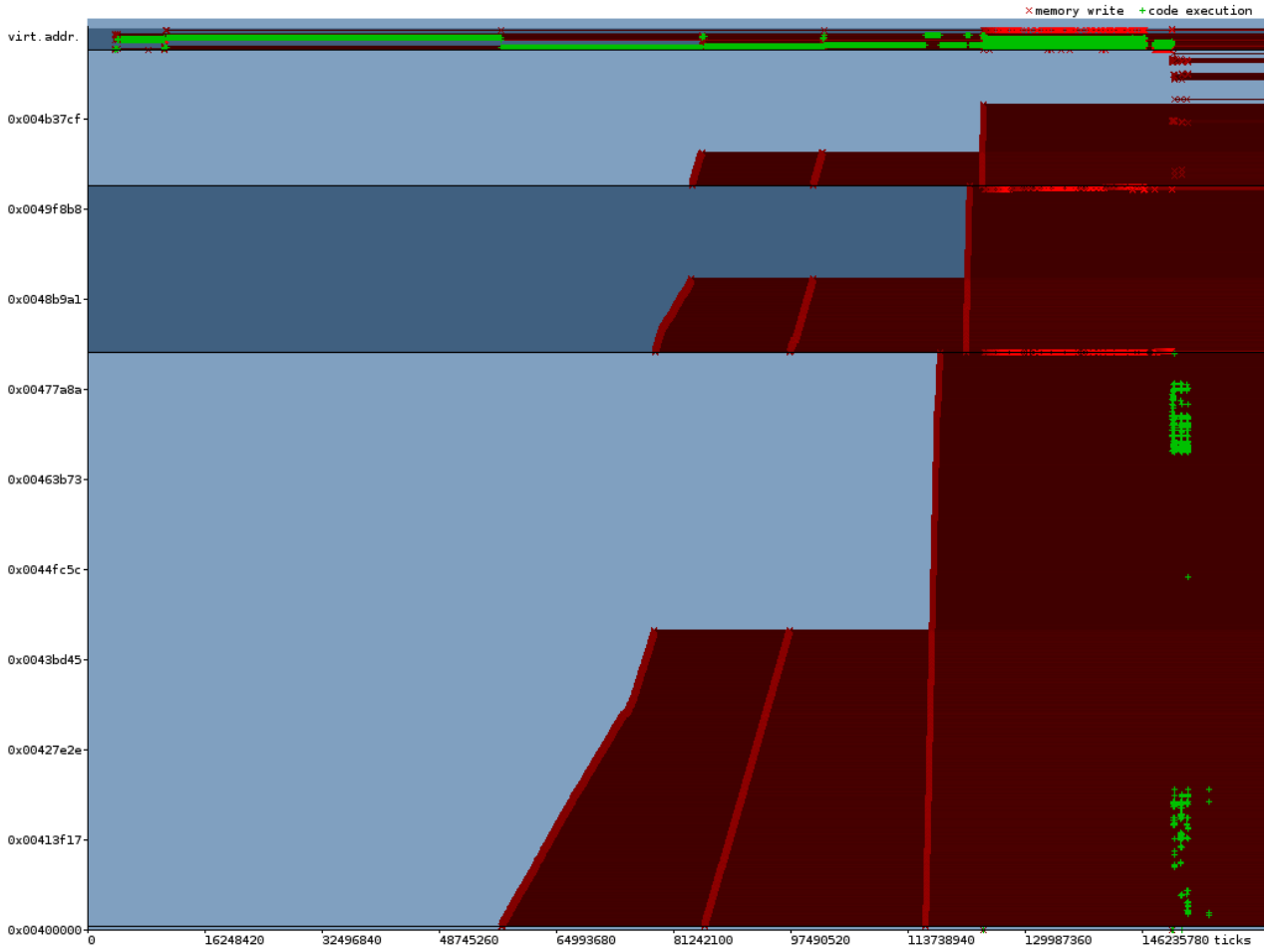


Figure: Unpacking PESpin 1.304

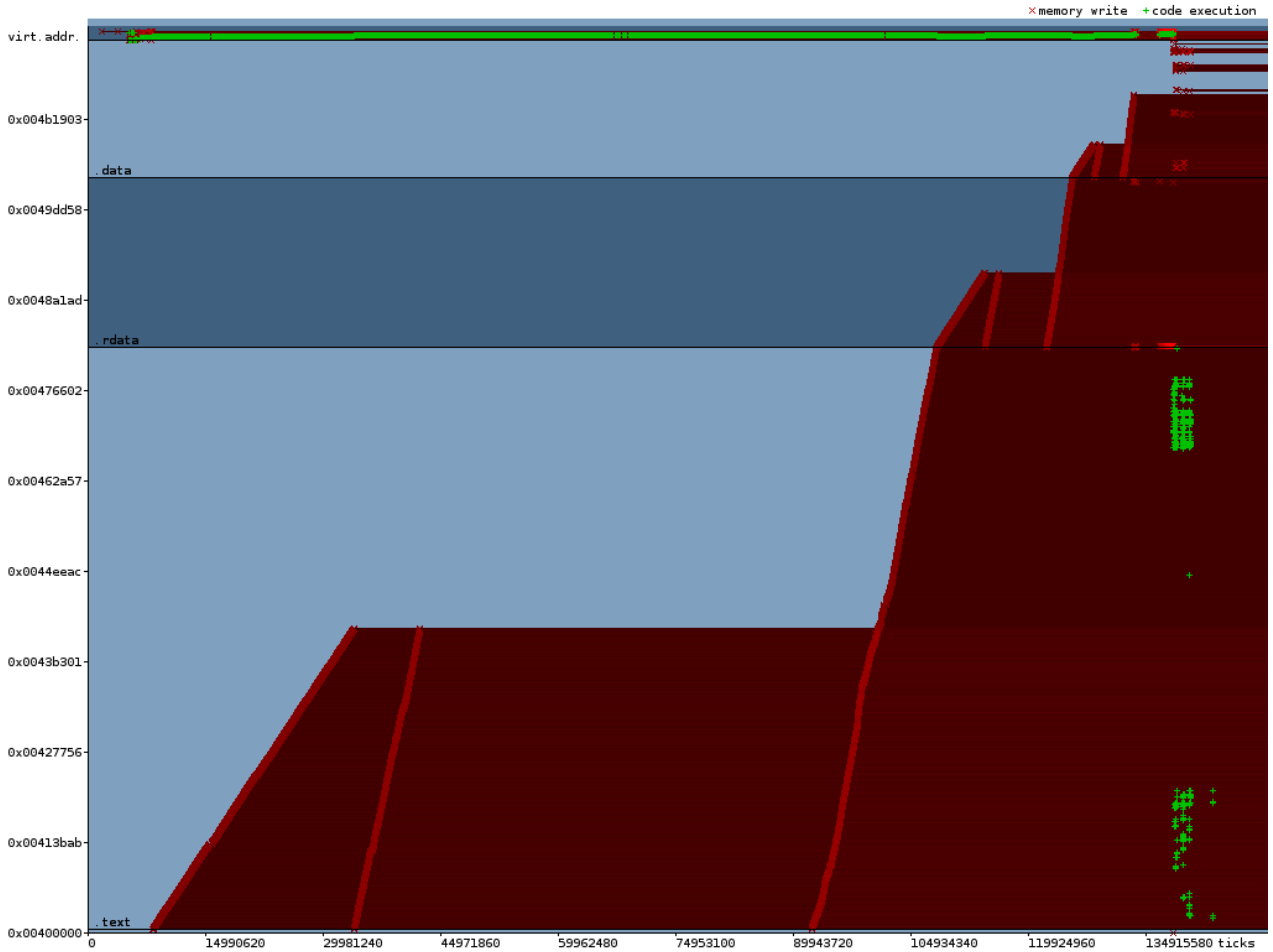


Figure: Unpacking tELock 0.98

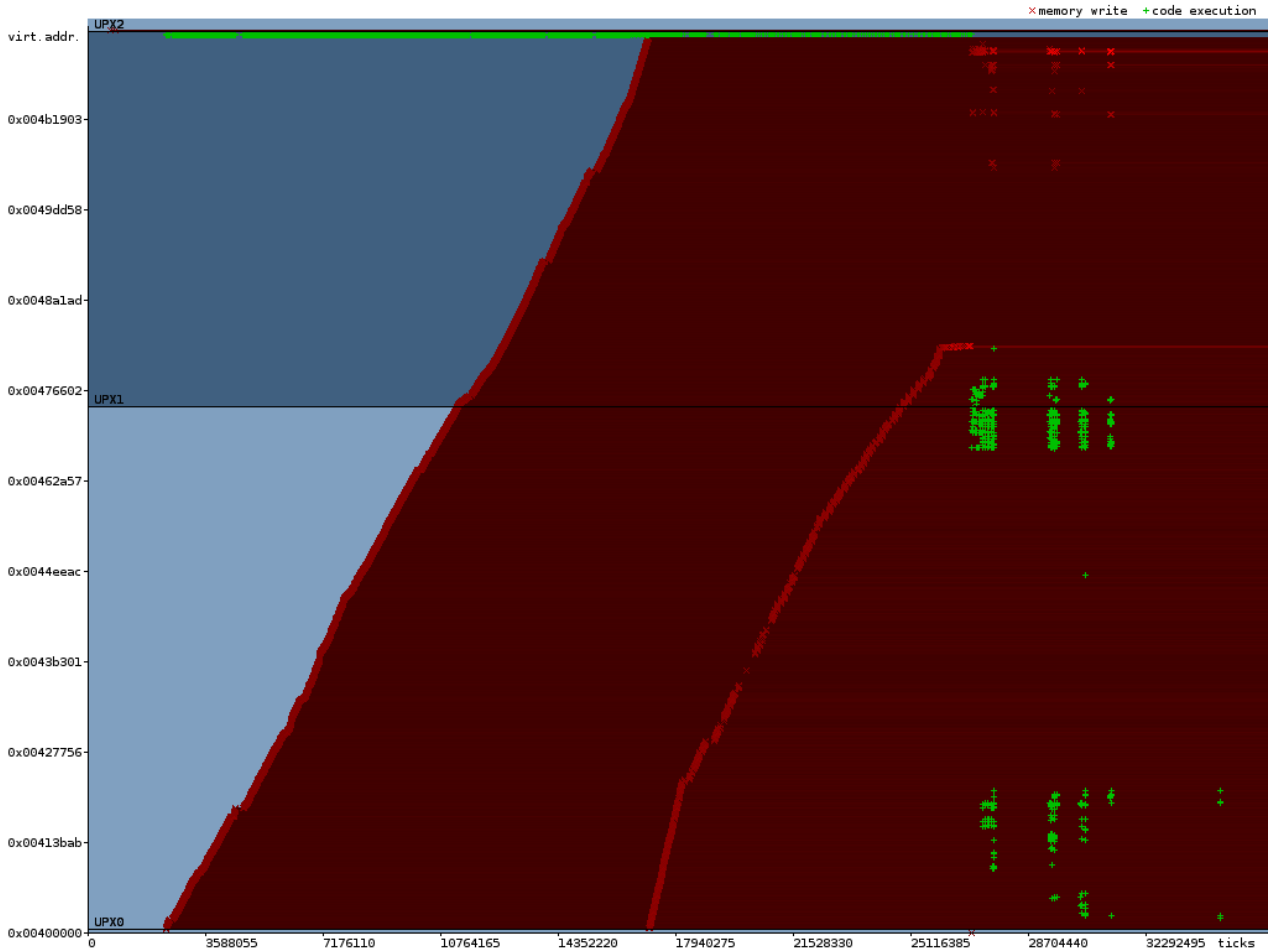


Figure: Unpacking UPX 3.01