A Practical Approach for Generic Bootkit Detection and Prevention

Bernhard Grill, Christian Platzer, Jürgen Eckel
Vienna University of Technology, IKARUS Security Software GmbH

bgrill@seclab.tuwien.ac.at

About This Talk

- Master student at the Secure Systems Lab @ Vienna University of Technology → http://www.iseclab.org/people/bgrill/
- Still ongoing research → preliminary results
- **Feedback / improvement ideas / discussion is very welcome! :)**

Outline

- Background (Objectives, Boot Process, Bootkits)
- System Overview & Detection Heuristics
- Implementation
- Preliminary Evaluation
- Limitations & Evasion Techniques
- Future Work & Open Questions

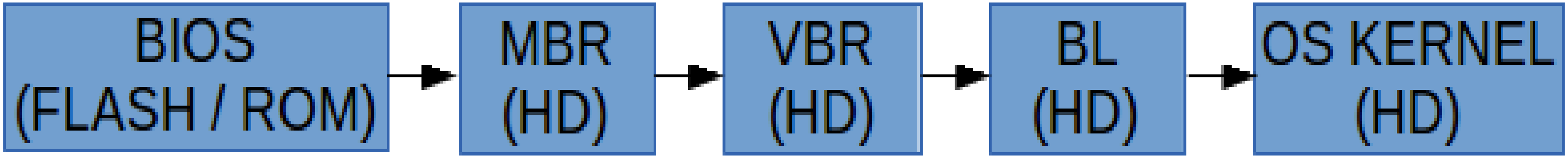
Background

- Objectives
- Boot Process
- Bootkits

Objectives

- Develop system to **detect** and **prevent bootkit attacks**
- Integrate with existing security measures like DEP, ASLR, AV, IDS,…
- Capable of detecting 0-days

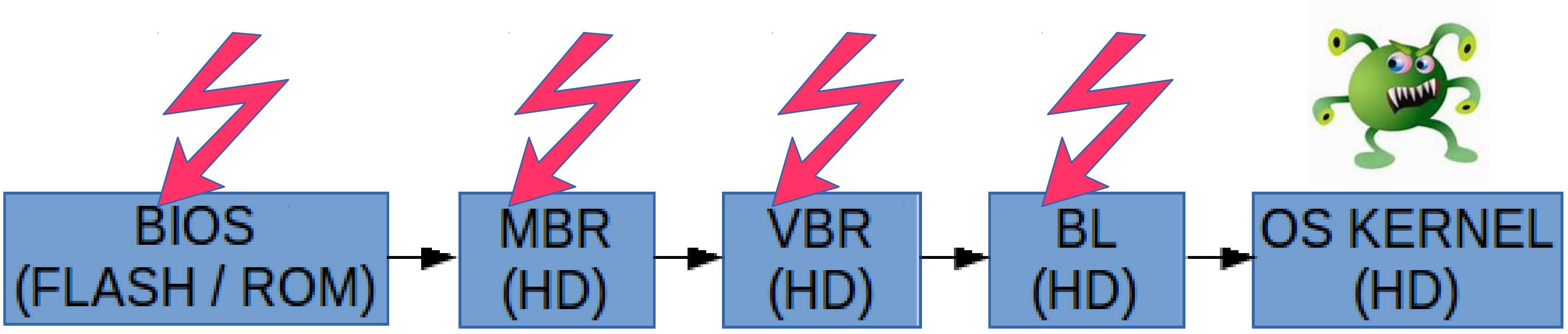
Boot Process Overview



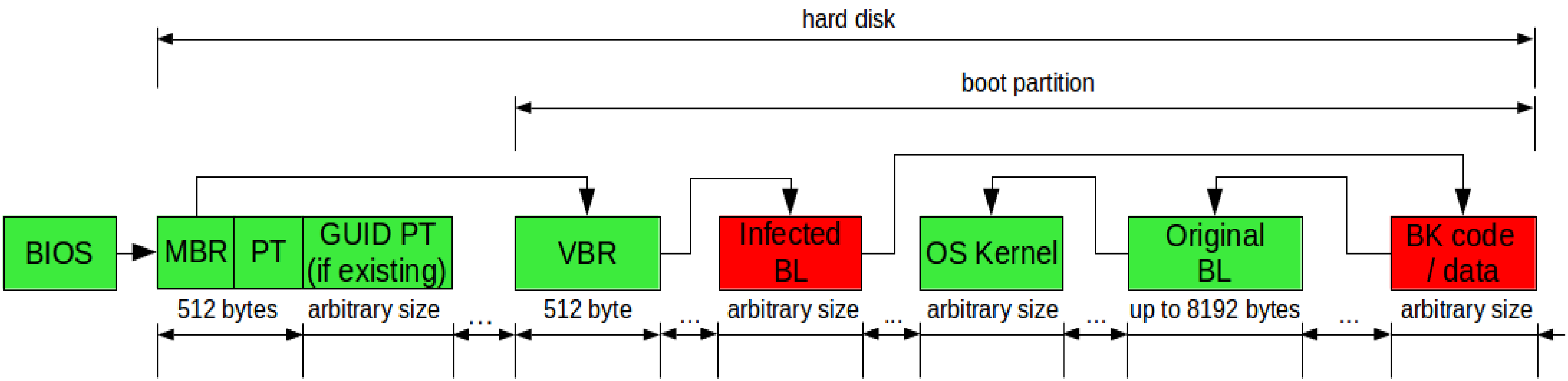
boot process overview on BIOS / MBR based systems

Bootkits

- “Bootkit” is a combination of the terms “boot” and “rootkit”
- Bootkits are a **very aggressive** kind of malware **deeply infecting** the **system**
- Bootkits interfere with the boot process to **gain control before the kernel starts** (and is able to protect itself)
- Target is to infect the kernel and **gain kernel-level privileges**



Bootkit Behavior



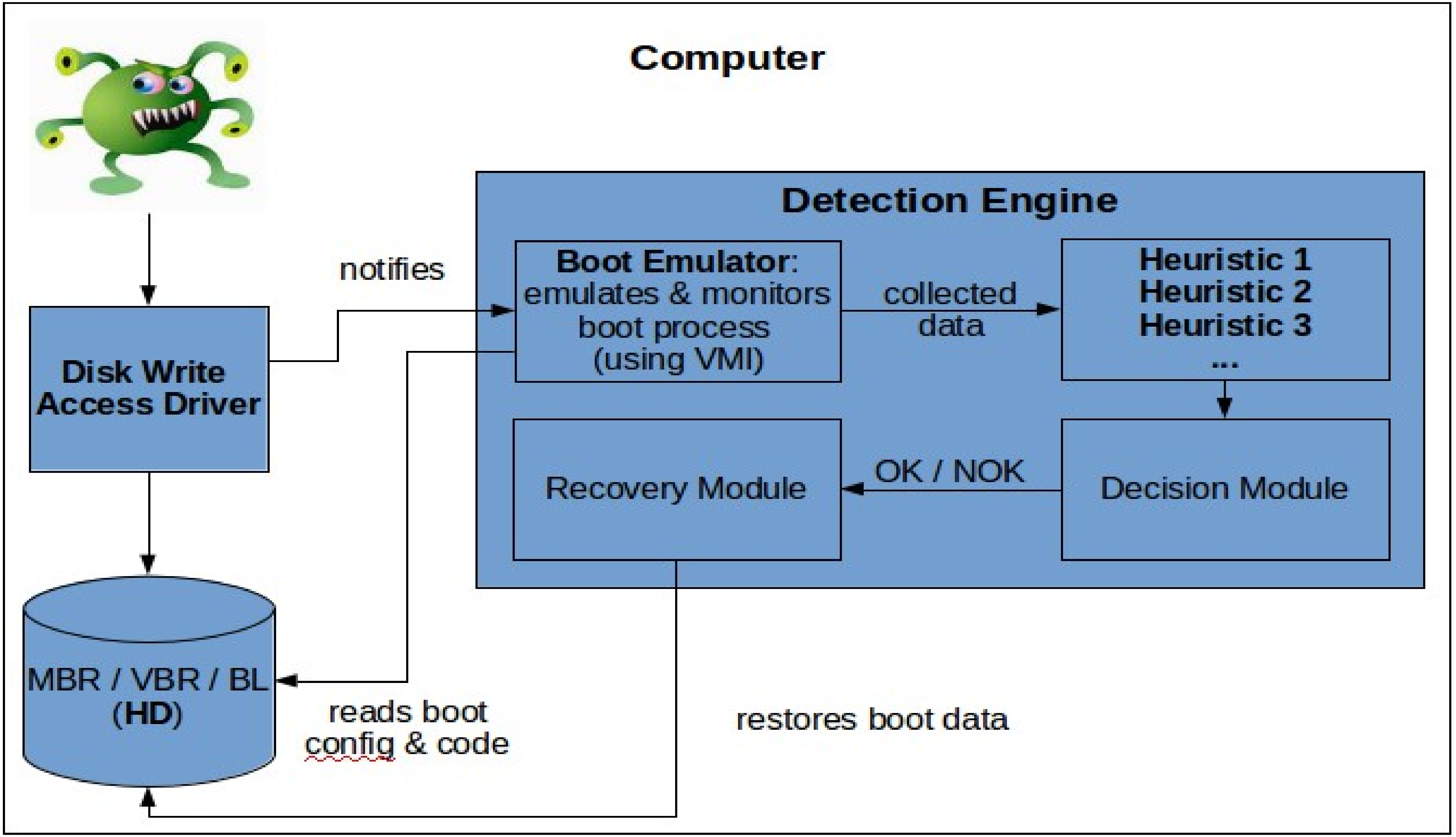
boot process with infected bootloader (BL)

System Overview &
Used Detection Heuristics

System Overview

- System consists of two major components (**driver**, **detection engine**)
- **Driver triggers detection engine** on write requests to hard disk areas containing boot code or data
- **Engine emulates** and **monitors** the **system boot process** during normal system operation

System Overview



Used Detection Heuristics

1) Disk access heur: bootkits store config & code at the end of the hard disk → we define **loading content from the disk's end** during boot process as **malicious**

2) Self-modifying code heur: **self-modifying code** is **prohibited**

3) Decryption routine heur: **loops** with **large iteration counts performing certain instructions** are **prohibited**

4) Hook heur: **Modifying** the **interrupt vector table** (IVT) during boot process is forbidden. To the best of our knowledge, this step is mandatory for bootkits.

Implementation

Implementation

- Implemented the system for Windows
- Kernel-level driver + user-land **detection engine** based on a **custom system emulator**
- Whitelisting for benign boot processes to avoid false-positives

Implementation

- System partially implemented
  - Driver PoC
  - Necessary emulator adoptions finished
  - Finished heuristics: decryption loop heur, disk read access heur, hooking heur
  - Todo: self modifying code heur, recovery module

Preliminary Evaluation

- Driver Performance Evaluation
- Engine Evaluation
  - Decryption Loop Filter
  - Disk Read Request Filter

Driver Performance Evaluation

5.06 GiB copy time without driver	19:57
5.06 GiB copy time with driver	20:09
Performance overhead	**1.0%**
Handled read requests (copy)	140511
Handled write requests (copy)	128724
Handled read requests (IDLE)	59
Handled write requests (IDLE)	409

Table 1: Overview on the performance measurement results for the driver.

Engine Evaluation

- Leaked Carberp bootkit was used for first evaluation
- Let's check the results of the implemented heuristics
  - Decryption loop filter
  - Disk read access filter

Engine Evaluation

```
=========== printing potential decryption loop info ===========
loop entry point: 0xd00008c9
loop exit point: 0xd0008ea

printing loop iteration information:

loop iteration counter: 1217
instruction count of loop iteration: 7

printing instructions:
0xd00d8c9: 33c2       xor        AX, DX
0xd00d8cb: 268905     mov        [ES:DI], AX ; 9f00:00ca = 0
0xd00d8ce: 83c602     add        SI, 02
0xd00d8d1: 83c702     add        DI, 02
0xd00d8d4: e212       loop       d8e8
0xd00d8e8: 8b04       mov        AX, [DS:SI] ; 0d00:0344 = cc9c
0xd00d8ea: ebdd       jmp        d8c9

===========    potential decryption loop info end    ===========
```

decryption loop heuristic output

Engine Evaluation

```
===========     printing potential malicious disk read requests     ===========
Size of hard disk in sectors 31457280 (15 GB)
Malicious read requests within the last 10 percent of the disk
starting malicious sector is 28311481 (13.5 GB)

number of sectors to read: 127
start sector to read: 31430339
target address to store content: 0x85c00000

number of sectors to read: 73
start sector to read: 31430466
target address to store content: 0x95a00000
===========       potential malicious disk read requests end       ===========
```

disk read request heuristic output

Limitations & Evasion Techniques

Limitations

- **No UEFI** support -> fundamentally different from BIOS/MBR boot process
- **No GPT** (GUID Partition Table) support **yet** -> will be included later
- **BIOS**- and **Hive-based bootkits not detected** (but they are **very rare**)

Evasion Techniques

- **Driver / engine detection** by full disk search (before infection)
- **Driver / engine removal** (assuming sufficient permissions & system restart) → self protection
- **Environment detection** during emulation (CPU, HDD model,…)
- **Instruction counter exhausting** (due to limited amount of emulated instructions) → emulate until kernel starts

Evasion Techniques on Heuristics

- <u>Disk read access heur</u>: **store** bootkits‘ **code and data** in unsuspicious areas, e.g. **not at hdd‘s end** (risky, due to accidential overwrites by OS)

- <u>Self-modifying code & decryption loop heur</u>: **refrain from** using **such code** -> prone to **pattern-based detection**

- <u>Interrupt hook filter</u>: to the best of our knowledge, **every bootkit performs interrupt hooking** to regain control after executing original code -> conjecture part of future work

Future Work & Open Questions

Future Work

- Implement missing parts
- Perform **larger evaluation** with different malware families
- Check whether **every bootkit relies on interrupt hooking**
- Check whether **benign boot processes trigger false positives** -> if not, remove white-listing

Conclusion

- Developed bootkit detection & prevention engine
- Based on **boot process emulation** and **virtual machine introspection (VMI)** to separate benign form **malicious boot processes**
- Prepared a demo if you're interested

Open Questions

- How to detect self-modifying code in x86? No, checking w+x on memory is not sufficient :)

Questions?

bgrill@seclab.tuwien.ac.at