"Nice Boots!" - A Large-Scale Analysis of Bootkits and New Ways to Stop Them

Bernhard Grill[1,3], Andrei Bacs[2], Christian Platzer[3], and Herbert Bos[2]

[1] SBA Research, Vienna, Austria
bgrill@sba-research.org
[2] VU University Amsterdam, Amsterdam, Netherlands
{a.bacs,h.j.bos}@vu.nl
[3] Secure Systems Lab, Vienna University of Technology, Vienna, Austria
{bgrill,cplatzer}@seclab.tuwien.ac.at

Abstract. Bootkits are among the most advanced and persistent technologies used in modern malware. For a deeper insight into their behavior, we conducted the first large-scale analysis of bootkit technology, covering 2,424 bootkit samples on Windows 7 and XP over the past 8 years. From the analysis, we derive a core set of fundamental properties that hold for all bootkits on these systems and result in abnormalities during the system's boot process. Based on those abnormalities we developed heuristics allowing us to detect bootkit infections. Moreover, by judiciously blocking the bootkit's infection and persistence vector, we can prevent bootkit infections in the first place. Furthermore, we present a survey on their evolution and describe how bootkits can evolve in the future.

Keywords: Bootkits, Large-scale Malware Analysis, Bootkit Detection, Bootkit Infection Prevention, Bootkit Evolution

1 Introduction

Bootkits are a class of malware specifically designed to interfere with the operating system's (OS) boot process. They were hugely popular in the 1980s/1990s [37], then faded into oblivion in the years after, to return with a vengeance from 2006 onward. Some of the most advanced and persistent malware today builds on bootkit technology. Their renewed popularity is caused by modern security mechanisms in Windows, which forced malware authors to look for alternative ways to "own" the OS. In particular, most of the attacks utilize bootkit technology to circumvent common measures like the Windows driver signing policy [26], kernel patch protection [25], but also regular AV software. Moreover, even governmental and commercial *"Remote Surveillance and Forensic Solutions"*, as offered by FinFisher and Hacking Team, apply bootkit technology in their tools [4].

Virtually all protection strategies today have a weakness in common: they rely on the integrity of the underlying operating system, either because they use its services (e.g., AV solutions), or because they reside in the OS itself. Consequently, malware subverting the system's initialization steps remains undetected.

A bootkit executes early during the boot process, long before the OS protection mechanisms kick in, allowing the bootkit to retain control throughout the infected system's boot phase. Combined with the fact that startup code is rarely modified on a typical system, bootkits often survive and stay undetected for a long time after the initial infection.

To this end, we developed *Bootcamp*, a bootkit detection, analysis and prevention framework. We performed a dynamic analysis for 25,513 malware samples from 29 different bootkit relevant families, spanning almost a decade of time (our first sample is from 2006 and our last from 2014) and analyzed their infection and runtime behavior, whereof 2,424 samples revealed bootkit behavior. To our knowledge, this makes it the single largest bootkit study ever conducted. From our study, we extract key properties to detect and prevent bootkit infections in common computer systems. Finally, we present an overview of bootkit technology evolution and describe how it may evolve.

Contributions. Our paper makes the following contributions:

1. We conduct the first large-scale study of 8 years worth of bootkit technology.
2. Based on the study, we propose new techniques to stop bootkit attacks.
3. We present *Bootcamp*, a platform to detect, analyze and prevent bootkits.

To our knowledge, the detection method we present cannot be evaded by any known MBR, VBR, or bootloader based bootkit. Thus, it significantly raises the bar for developers of malicious software who now have to target other components, such as the BIOS. The focus of our study is on systems that boot from a device's master boot record. As there are virtually no UEFI bootkits that are not research proof-of-concepts, we left them out of scope for our study.

2 How bootkits interfere with the boot process

Booting. Figure 1 sketches the boot process for BIOS based systems. The CPU boots in *real mode* and executes the BIOS, which locates and passes control to the Master Boot Record (MBR). The MBR is located within the first 512 bytes of the system's hard disk. The MBR code parses the partition table (PT) to determine the "bootable" partition (containing the OS) and hands over control to the Volume Boot Record (VBR). The VBR resides within the bootable partition's first 512 bytes and contains further information necessary for booting such as the filesystem parameters and the bootloader's disk location. The VBR code then loads the bootloader's (BL) first stage into memory and passes control to it. Finally, the BL loads further code from the disk, switches to *protected mode*, and loads and executes the kernel.

Dark regions. We define a *dark region* (DR) as a contiguous physical disk region which is not part of a filesystem. Since it is not part of a filesystem, it is also not accessible and invisible to the user during normal system operation. Examples include the MBR, the sectors between the MBR and VBR, the bootloader, the inter-partition gaps, and the space *beyond the last partition* and the *end of the*

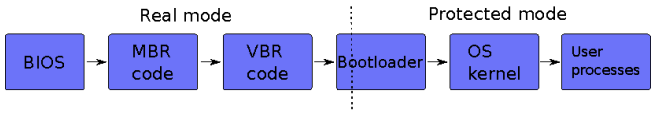

Fig. 1. Boot sequence for BIOS based systems

physical disk (these gaps have up to several MB size on most systems). Figure 2 shows a typical disk layout and corresponding *dark regions* (*dark regions* are colored in gray). Some of these regions (e.g., the MBR, VBR, and bootloader) are used during system startup, while others (like inter-partition gaps) are never used at all. What makes dark regions interesting for attackers is that only very rare events (like major OS updates) ever modify them and current protection mechanisms typically do not cover them, as they operate at the filesystem level.

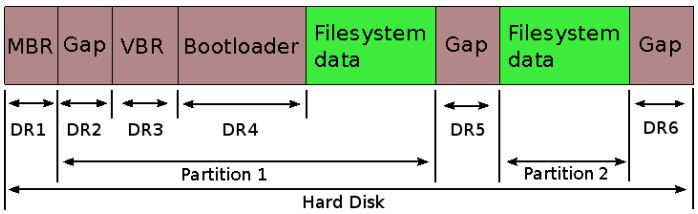

Fig. 2. Typical *dark region* (DR) layout - *dark regions* in gray, filesystem areas in green

Bootkit infection techniques. Bootkits interfere with the startup process by replacing any boot stage prior to the kernel's execution (e.g., the MBR, VBR or bootloader) with the initial infection vector. They execute malicious code before the OS employs protection techniques and subvert the kernel to keep control throughout the infected system's runtime. Typically the initial infection vector initializes the bootkit (e.g,. by hooking certain interrupts, see below), loads further malicious code from the disk and redirects execution there. After executing the malicious code, the bootkit returns to the original bootstrap code continuing the intended (but now infected) boot process.

Due to the limited disk space for startup code, bootkits typically utilize further disk areas (e.g. *dark regions* with sufficient space) to store configuration settings and additional code, which is finally executed or injected into the kernel's memory (e.g. a kernel driver). Modern bootkits often encrypt those hidden storage areas [24]. Bootkits have several hard requirements. 1) They must execute at least once before the kernel takes control and activates defensive measures. 2) Bootkits need to survive system restarts. 3) They must be space efficient as the available space for the initial infection vector is highly limited. 4) The code has to run in *real* and *protected mode* and survive mode switches. 5) They should not delay OS startup significantly.

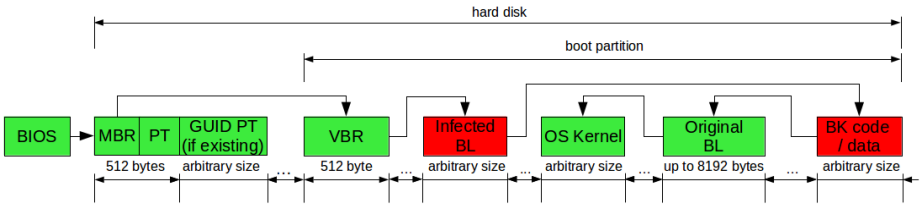

Fig. 3. An example execution flow exploiting the bootloader as initial infection vector - infected areas in red, benign in green [14]

As an example, Figure 3 shows the execution flow for a bootkit utilizing an infected bootloader. The BIOS passes control to the MBR code which in turn loads and executes the VBR. The VBR code loads the infected bootloader (BL) which executes further malicious code from the system's disk (e.g. the *dark region* at the drive's end). The latter code performs additional malicious activity—like interrupt hooking. Later, the bootkit executes the original bootloader in order to load the kernel into memory. When this is done, the bootkit regains control by means of a hooked interrupt and infects the kernel in memory. Finally, it proceeds initializing and executing the (now infected) system.

Strictly speaking, there are more possible bootkit types. Besides the ones targeting the MBR or VBR (like TDSS, Sinowal, and Pihar [3]) and the ones aiming for the bootloader (like Cidox/Carberp [21]), bootkits could target the BIOS and other technology also. For instance, the research PoC IceLord bootkit [2], writes its code directly into the BIOS FLASH/ROM. At the other extreme, bootkits may in principle target Windows specific components like `boot.ini` or hive [23]. As MBR, VBR, and bootloader bootkits are by far the most popular (and the others are hardly seen in practice), we will focus on these.

Interrupt hooking. Bootkits need to regain control *after* the kernel is loaded in memory by the bootloader, but *before* it executes, so they can infect it before it runs. For this purpose, a bootkit hooks the system's interrupts. Specifically, the CPU holds a data structure known as the Interrupt Vector Table (IVT) in the lower 1K of memory in *real mode*. The IVT contains 256 entries, each representing one interrupt and containing a pointer to the corresponding Interrupt Service Routine (ISR). Whenever an interrupt occurs, the CPU consults the IVT and executes the corresponding ISR. After processing the interrupt, the ISR returns control to the instruction executed at the time the CPU was interrupted. Bootkits typically install interrupt hooks by overwriting the original ISR's address with a malicious one (e.g., interrupt `0x13` which provides low-level disk services). This allows them to regain control after executing benign code such as the original bootloader. After executing the malicious routine they redirect control to the original so that it may continue its intended function.

3 A large scale analysis of bootkit technology

We conducted a large scale analysis of bootkit technology using a dataset of 25,513 malware samples to get a deeper insight into bootkit behavior. This section presents the results of the analysis, as well as the evolution of bootkit technology for a timespan of 8 years. We start by presenting the bootkit dataset, experimental setup and containment measures. Striving for sound experimentation, Table 1 shows the checklist advocated by Rossow et al. for malware experiments [34], and to what extent we apply the suggested guidelines.

Criterion	Satisfied	Criterion	Satisfied	Criterion	Satisfied
Removed goodware	Y	Interpreted FPs	Y	Removed moot samples	N
Avoided overlays	Y	Interpreted FNs	N	Real-world FP exp.	NA
Balanced families	Y	Interpreted TPs	N	Real-world TP exp.	NA
Separated dataset	NA	Listed malware families	Y	Used many families	Y
Mitigated artifacts/biases	NA	Identified environment	Y	Allowed Internet	Y
Higher privileges	Y	Mentioned OS	Y	Added user interaction	Part.
		Described naming	Y	Used multiple OSes	Y
		Described sampling	Y		
		Listed malware	Y		
		Described NAT	Y		
		Mentioned trace duration	Y		

Table 1. Rossow's checklist for malware experiments [34]

Bootkit dataset for evaluation. Altogether, we utilized 25,513 malware samples from 29 different families for our experiments. The malware samples were chosen based on their likelihood to install a bootkit [35]. The samples were selected by an AV vendor's malware label, matching one of 29 predefined families. Table 2 outlines the annual distribution on the bootkit dataset over a timespan of 8 years, while Table 5 shows more details of the families and sample set. Additionally, we analyzed 100 benign binaries to examine the behavior for non-malicious executables.

Experimental setup. We monitored the boot process in a virtualized environment running Windows XP SP3 and Windows 7 (both 32 bit) and extracted defining features for bootkits. We describe in detail the virtualized analysis environment in a Section 5. Following the guidelines from Rossow [34] we deactivated the Windows firewall on both systems and disabled the UAC on Windows 7, as the weaker security measures allow us to observe more malware behavior. The evaluation was performed between September 2014 and February 2015. We provided a dedicated CPU core and a main memory size of 1024 MB for XP and 2048 MB for Win7 for each analysis environment. Each sample had about 90 seconds to perform the infection. After infection, the systems were rebooted. They

First appeared	Sample count	Share per year
2006	30	0.1%
2007	26	0.1%
2008	575	2.3%
2009	1,737	6.8%
2010	2,902	11.4%
2011	6,431	25.2%
2012	7,771	30.5%
2013	2,647	10.4%
2014	3,394	13.3%
	25,513	100.0%

Table 2. Annual distribution of malware sample set

were allowed to use up to 2 minutes for the boot phase, which is more than sufficient for fully booting the OS. If the OS took longer to boot, the system was killed and marked as broken by the malware. The Win7 environment used a disk layout (and hence *dark region* layout) with MBR partitioning and a single system partition, identical to Figure 2 (*Partition 2* in the figure is equivalent to the hidden system partition Windows 7 is typically generating). The Windows XP environment used a similar layout but with a single partition.

Containment. We performed initial experiments allowing internet access for the samples but performed the main evaluation without. When we allowed internet access we state it next to the experiment. In general, we did not allow internet access, as it would distort the results on the historic evolution of bootkit technology: a dropper with internet access may fetch a new bootkit version instead of installing the embedded, historic one. Furthermore, there is hardly any difference allowing or denying internet for our sample set, as shown in Section 3.1. When allowing internet, the system was behind a NAT and rate limited.

3.1 Large-Scale Bootkit Analysis Results

We define the terms "***bootkit like behavior***" as samples writing to any *dark region*, "***bootkit detected***" as samples writing either the MBR, VBR or bootloader during the bootkit infection (so *bootkit detected* implies *bootkit like behavior* but not necessarily the other way round), "***working infections***" as samples satisfying *bootkit detected* definition and the system reboots successfully after infection and "***successful infection rate***" as the rate (in %) between the number of samples with *bootkit like behavior* and *working infections*. Table 3 compares the results for Windows XP allowing or denying internet access to samples. Allowing internet access increased the number of samples with *bootkit like behavior* by 16.7% on Windows XP. Interestingly the number of *detected bootkits* and *working infections* increased only by 1.4% respectively 1.1%. Hence, there is little difference between allowing or prohibiting internet access for *working infections*. This was the only experiment we performed with internet access. The other tests denied internet access, for the reasons discussed earlier.

Table 4 outlines the analysis results separated by OS. The category *XP and Win7* defines the number of samples working on both OSes, whereas *XP or Win7*

	XP without internet	XP with internet	Difference
BK like behavior	2,405	2,888	16.7%
Bootkit detected	2,042	2,073	1.4%
Working infections	1,143	1,156	1.1%
Succ. infection rate	47.5%	40.0%	7.5%

Table 3. Comparing analysis' outcome with and without internet access on XP

specifies the amount of executables working on at least one of both systems. Though, we detected slightly more bootkits or XP, we observed more *working bootkit infections* on Win7. Only 43.8% of the samples operate on both OSes. 258 samples work exclusively on XP, whereas 15 samples operate on Windows 7 only. Hence, nearly all samples working on Win7 are functional on XP too. This observation does not hold for the other direction. Altogether, 2,424 samples out of 25,513 revealed *bootkit like behavior* on at least one OS.

	XP	Win7	XP and Win7	XP or Win7
BK like behavior	2,405	2,147	2,128	2,424
Bootkit detected	2,042	1,799	1,784	2,057
Working infections	1,143	1,647	931	1,859
System boot fails	1,262	500	1,197	565
Succ. infection rate	47.5%	76.7%	43.8%	76.7%

Table 4. Experimental sample set's operability on different operating systems

Table 5 highlights details and outlines evaluation results for each malware family contained in our bootkit dataset. The *successful infection rate* strongly varies between families and OS. Some families target only XP e.g. Finfish and Lapka, while others like Pihar and TDSS exclusively focus on Win7, but in general most target both OSes, as Cidox, Sinowal and Smitnyl do. Especially most recent samples have relatively high *successful infection rates* as Pihar, Sinowal, Smitnyl and TDSS have for Windows 7 and Niwa, Sinowal, Smitnyl have for XP. Tough some families hold a high *sample count* they feature a very low *successful infection rates*. This might have multiple explanations. For example they might target only particular systems (e.g. a specific language), perform anti-VM / anti-analysis techniques (as discussed in Section 7) or fail to install a properly working bootkit for our analysis system.

Recall, bootkits have to acquire additional disk space as the initial infection vector is typically too small for their complete code and data (requirement 3 in Section 2). Hence, they have to store their data somewhere else. Table 6 outlines which dark regions (DR) were exploited as data storage by the bootkits. *End of disk* indicates exploitation of the space beyond the last partition (*DR6* in Figure 2), whereas *between partitions* includes any space between the partitions, including the gap between MBR and VBR (*DR5* and *DR2* for Win7 and *DR2* for XP). Sometimes bootkits use both locations to store data, indicated by *both*. On average 81.1% store their data at the *end of the disk*, 18.2% on average utilize

Malware family	First appeared	Last appeared	Sample count	XP Working infections	XP Succ. infection rate	Win7 Working infections	Win7 Succ. infection rate
Yurn	2006-05	2013-06	43	1	2.3%	1	2.3%
SST	2006-05	2012-10	61	0	0.0%	0	0.0%
Sinowal	2006-05	2014-06	2,938	678	23.1%	582	19.8%
Plite	2006-10	2014-06	3,908	1	0.0%	1	0.0%
Infinaeon	2007-10	2007-10	1	0	0.0%	0	0.0%
Trup	2007-12	2014-06	1,862	72	3.9%	50	2.7%
TDSS	2008-07	2014-06	2,945	0	0.0%	495	16.8%
Zhaba	2008-10	2010-05	5	0	0.0%	0	0.0%
Qvod	2009-03	2014-06	2,831	0	0.0%	0	0.0%
Stoned	2009-07	2014-02	23	4	17.4%	2	8.7%
Smitnyl	2009-08	2014-03	297	56	18.6%	65	21.9%
Xpaj	2009-10	2014-06	547	16	2.9%	17	3.1%
Niwa	2009-10	2014-06	38	8	21.1%	1	2.6%
Phanta	2010-03	2014-05	954	80	8.4%	27	2.8%
Wistler	2010-05	2014-06	814	1	0.1%	0	0.0%
Nimnul	2010-07	2014-06	2,499	0	0.0%	2	0.1%
Finfish	2010-10	2014-06	29	1	3.4%	0	0.0%
Fisp	2011-03	2014-04	246	0	0.0%	0	0.0%
ZAccess	2011-05	2014-07	1,948	0	0.0%	0	0.0%
Lapka	2011-06	2014-06	94	15	16.0%	0	0.0%
Cidox	2011-07	2014-06	2,858	193	6.8%	193	6.8%
Mybios	2011-07	2014-03	47	13	27.7%	0	0.0%
Pihar	2011-08	2013-03	447	0	0.0%	209	46.8%
GoodKit	2011-11	2011-11	1	0	0.0%	0	0.0%
CPD	2012-05	2014-06	38	3	7.9%	1	2.6%
Geth	2012-06	2012-10	6	0	0.0%	0	0.0%
Korablin	2012-08	2014-06	27	0	0.0%	0	0.0%
Backboot	2013-03	2013-03	1	1	100.0%	1	100.0%
Careto	2014-02	2014-02	5	0	0.0%	0	0.0%
			25,513	1,143	4.8%	1,647	6.9%

Table 5. Overview on evaluation results per malware family contained in sample set

the space *between partitions*, while only a few samples split the data storage and exploit both locations. In general, we observe a trend toward utilizing the space *between partitions*. All samples wrote to *dark region* 2 or 5 on XP respectively 2, 5 or 6 on Win7 (see Figure 2). Later, we use this property in our detection and prevention system.

Data storage location	XP Count	XP Share	Win7 Count	Win7 Share	Avg on both
End of disk	957	83.7%	1,306	79.3%	81.1%
Between partitions	170	14.9%	337	20.5%	18.2%
Both	16	1.4%	4	0.2%	0.7%
Writes to DR	**1,143**	**100.0%**	**1,647**	**100.0%**	**100.0%**
	1,143	100.0%	1,647	100.0%	100.0%

Table 6. Exploited *dark regions* as data storage locations from successful infections

Table 7 outlines the exploited initial infection vectors to gain control during the boot process. The MBR is by far the dominant infection vector utilized by 86.1% on average, followed by the bootloader exploited by 13.8% samples on

average. The VBR is used by hardly any sample. There is a noticeable trend from exploiting the MBR toward utilizing the bootloader (and perhaps VBR, in an incipient phase). We describe this shift in more detail in Section 3.2.

	XP		Win7		
Infection vector	Count	Share	Count	Share	Avg on both
MBR	948	82.9%	1,453	88.2%	86.1%
VBR	2	0.2%	1	0.1%	0.1%
BL	193	16.9%	193	11.7%	13.8%
	1,143	100.0%	1,647	100.0%	100.0%

Table 7. Initial infection vectors from successful infections

The hooked interrupts are shown in Table 8. As some samples hook multiple interrupts the sum increased to 1,172 / 1,660 (XP / Win7). The most exploited interrupt are `0x13` (85.1%), and `0x15` (14.1%). Interrupt `0x13` is normally used to load disk content into memory and is therefore convenient to verify whether the kernel is loaded into memory. The obvious drawback of interrupt `0x13` is the huge number of calls, as it is called a few thousand times during startup. On the other hand interrupt `0x15` is typically called once during the boot phase—to gain the system's memory map and executed before passing control to the kernel. Hence, it's an excellent interrupt hooking target for bootkits. The hooks for interrupt `0x83` and `0x85` were exploited by 8 *Niwa* samples on XP and one on Win7 between 2009 and 2012. 13 *Mybios* samples on XP and 1 *Plite* on both OSes do not hook any interrupt as those families feature a slightly different attack model. Instead of patching the kernel directly in memory, those samples replace explorer.exe with a malicious one on the disk, but therefore leave traces on the filesystem [1]. We discuss these techniques in more detail in Section 3.2.

	XP		Win7		
Hooked interrupt	Count	Share	Count	Share	Avg on both
0x13	947	80.8%	1,464	88.2%	85.1%
0x15	203	17.3%	194	14.0%	14.1%
0x83 & 0x85	8	0.7%	1	0.1%	0.3%
None	14	1.2%	1	0.1%	0.5%
	1,172	100.0%	1,660	100.0%	100.0%

Table 8. Exploited interrupt hooks from successful infections

3.2 Historic Perspective on the Evolution of Bootkit Technology

This subsection discusses the historic evolution of bootkits. Table 9 highlights the historic perspective on successful infections rates. We observe a high *successful infection rate* for the samples dating from 2008 and 2009, followed by intense decline hitting the low-point in 2011. Starting with 2012 the *successful infection*

	XP			Win7		
Year	BK like behavior	Working infections	Succ. infections	BK like behavior	Working infections	Succ. infections
2006	0	0	-	0	0	-
2007	0	0	-	0	0	-
2008	7	7	100.0%	0	0	-
2009	464	450	97.0%	453	380	83.9%
2010	422	189	44.8%	379	216	57.0%
2011	907	128	14.1%	793	577	72.8%
2012	318	127	39.9%	275	240	87.3%
2013	48	37	77.1%	37	27	73.0%
2014	239	205	85.8%	210	207	98.6%
	2,405	1,143		2,147	1,647	

Table 9. Overview on the historic evolution on successful infection rates

rate raised again to peak in 2014. We did not observe any working bootkit for Win7 before 2009, since Win7 was released in the end of that year. From 2010 to 2012 the *successful infection rate* for XP was relatively low. This might indicate a focus on Win7 and Win8 (which we did not evaluate in this paper) prior to maintain operability on the older XP. But in general the trend indicates an improvement and professionalisation in the underground malware industry for bootkits. The historic development on used infection vectors is outlined in Figure 4. Until 2011 the MBR was the only initial infection vector exploited. In 2012 and 2013 a few samples used the VBR as initial infection vector, while the bootloader (BL) became more popular starting with 2013. In 2014, the BL was the most exploited infection vector. The tremendous increase in BL infections is very likely related to the *Carberp* botnet's source code leak from 2013 [21]. It contained the source code of the *Cidox* bootkit which was applied by *Carberp* [35]. Likely *Cidox* will become the new de facto standard template for bootkits as *Zeus* did for banking trojans after its infamous leak in 2011 [10]. Regrettably the *Zeus* leak also lead to a dramatic amount of mutations and peaked in various highly sophisticated banking trojans like Citadel [5], SpyEye [36] or Zeus P2P variants like ZeroAccess and Kelihos [32]. Figure 5 shows the evolution of the

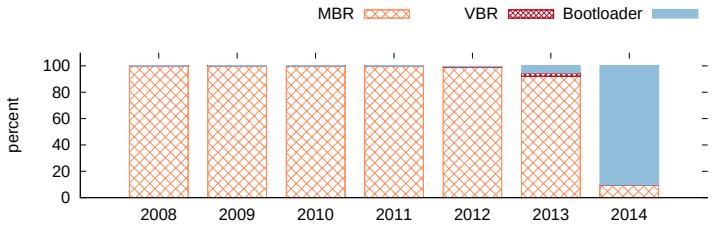

Fig. 4. Overview on initial infection vector evolution

dark region usage by bootkits. In 2008 the space *between partitions* and at the

end of the disk were both utilized by bootkits to store their code and / or data. This behavior changed dramatically in the next years, as samples almost exclusively exploited the space beyond the last partition. In 2010 the preferred data storage location started shifting again, resulting in equal exploitation of space at the *end of the disk* and *between partitions* in 2014. The evolution of

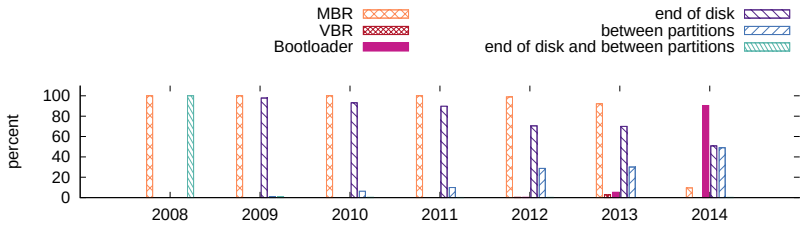

Fig. 5. Overview on *dark region* utilization evolution

interrupt hooking is outlined in Figure 6. In 2008 we observed the first samples hooking interrupt `0x13`. The *Niwa* family started hooking interrupt `0x13`, `0x15`, `0x83` and `0x85` in 2009. They exploited `0x83` and `0x85` to call the original ISR for interrupt `0x13` and `0x15`. *Niwa* is the only family we observed hooking the interrupts `0x83` and `0x85`, but we did not monitor any *Niwa* sample after 2012 anymore. There is a clear shift of exploiting interrupt `0x13` to utilizing `0x15` throughout the years. This shift might be explained by the *Carberp* leakage [21] again, as it also exploits interrupt `0x15`. Therefore interrupt `0x15` will remain the dominant hook in the future. *Mybios* introduced an interesting technique in 2011. They avoided interrupt hooking and in memory kernel patching techniques, by replacing the original *explorer.exe* with a malicious one on disk during the boot process. Though this is a highly interesting technique and simplifying bootkit development, it did not establish itself in the scene, likely because it relies on changing files within the filesystem leaving potential suspicious traces there. This technique was later adopted by *Plite* in 2014.

Fig. 6. Overview on interrupt hooking evolution

4 Detecting and Preventing Bootkit Infections

4.1 Detecting Bootkit Attacks

Bootkits are installed either (1) by a malware dropper during system operation (which is the common case), or (2) when the physical disk is modified directly while the target system is off. Given the characteristics of the previous section, we propose heuristics to detect bootkit infections either during installation but also after the system is already infected (except dark region modification heuristic).
(1) *Dark region* modification heuristic. As discussed, bootkits have to modify the target system's bootstrap code located in *dark regions* (MBR, VBR, bootloader) to infect the system and survive system restarts (first and second requirements in Section 2). Furthermore, they store additional code and data in other *dark regions*, as this space is typically not used under normal operation and hence, is not in danger of being accidentally overwritten by the OS or detected by filesystem based protection solutions (see requirement three). Thus, *dark region* modifications (i.e. disk writes to any *dark region*) are indicators for suspicious behavior. Figure 2 outlines a typical disk and *dark region* layout (with two partitions). For example in an infected system the MBR (*DR1*), the gap between the MBR and the VBR (*DR2*), and the space beyond the last partition (*DR6*) could be utilized by the bootkit.
(2) *Dark region* read heuristic. While booting an infected system its malicious code may need to load additional content from its hidden storage (requirement three), e.g. at the disk's end. Therefore a read operation from a *dark region* normally not involved in the boot process is another indicator for compromise.
(3) Interrupt hooking heuristic. Bootkits must execute at least once, but typically multiple times at different stages during the boot process. As mentioned earlier, a good and proven solution to do so is by means of hooking. Interrupt hooking is not performed in *real mode* on a clean system. Therefore, an interrupt hook during system bootstrap is a good indicator for bootkit activity.
(4) Reuse existing code heuristic. In a clean system the BIOS loads the MBR code at address `0000h:7C00h` and starts executing it. The MBR code loads the VBR code from the boot partition again at `0000h:7C00h`, and executes the address again. Hence, during a clean system's startup process the number of jumps to this fixed address is exactly two. Bootkits exploiting the MBR or VBR as initial infection vector typically backup the original code on the disk to reuse it. As the malicious and the benign code is both loaded to `0000h:7C00h` the number of jumps to this fixed address for infected systems is greater than two. This heuristic was also utilized by Haukli to detect system startup anomalies [15].
Misdetection and evading the heuristics. We now look at the strength of our heuristics. *(1) Dark region modification heuristic*: False positives may occur in case of legitimate *dark region* updates, e.g., during a major system upgrade, partitioning or a complete reinstallation of the system, but malware cannot evade detection since it has to write to boot process relevant disk sectors. Some bootkits may mark disk sectors as bad but still use them. This evasion would fail because this heuristic checks the location of changed sectors and not

their quality. *(2) Dark region read heuristic*: The bootkit may store its data in unsuspicious disk sectors inside a filesystem. This technique would induce more risk for the bootkit to get accidentally overwritten by the OS, as it is unaware of the disk sectors utilized by the bootkit. Furthermore, it may be detected by filesystem based protection solutions. *(3) Interrupt hook heuristic*: Samples may not exploit interrupt hooking as discussed in Subsection 3.2. Those samples lack the ability to regain control after executing legitimate boot code and hence rely on a different attacker model. They change files directly within the filesystem instead of patching / injecting code in memory, but replacing files on the disk leaves suspicious traces within the filesystem. *(4) Reuse existing code heuristic*: As discussed in Section 6.1 bootkits infecting the bootloader can circumvent this heuristic. Those samples do not execute a malicious MBR / VBR, followed by the benign one and therefore execute the address `0000h:7C00h` just twice. Moreover an attacker might patch the original MBR / VBR to be loaded and executed to a different memory address or refrain from reusing existing code.

False-Positives. False positives may be induced by major OS upgrades which change the way bootstrapping is done until now or exotic / self written bootloaders exploiting bootkit-like techniques but for benign purposes. Such intended but suspicious behavior could be whitelisted for a specific system without affecting detection performance, as the heuristics seek for boot process anomalies and hence, such whitelists would redefine permitted boot behavior. For example one could allow the bootloader to read (even write) from (to) certain additional dark regions during the boot process. Interrupt hooks may be allowed during certain boot stages, for particular interrupt numbers, to specific ISR target addresses executing predefined (e.g. verified via cryptographic secure hashing) ISR code. The *reuse existing code* heuristic may be adapted for deviating boot behavior by adjusting the number of at most allowed jumps to a certain address, as reusing code still increases this count beyond the allowed maximum.

Application Scenarios. Our detection heuristics can be divided into two types: 1) heuristics for bootkit attacks at installation time on a running system and 2) heuristics for existing infections while booting a system, after a potential infection. Although the heuristics are implemented in a virtual machine, detecting bootkits is not restricted to VMs. The VM with heuristics can take any disk as input, e.g. another VM's disk or a physical hard disk. This use case is interesting in post-infection scenarios, forensics or in settings where one is not sure whether a system is infected or not. The *dark region modification* heuristic detects bootkits at installation time. The other three detect bootkit activity during an infected system's boot phase. The three heuristics can be used to detect an infected disk even **after** infections.

4.2 Preventing Bootkit Infections

In this subsection we discuss a bootkit prevention technique for virtualized environments. Full system emulators such as QEMU, Bochs, VMware or VirtualBox are capable of implementing our proposed preventive measures directly inside the emulated hardware. Based on the information provided in Section 2, restricting

dark region modifications (e.g. MBR, VBR, bootloader, inter-partition space) defeats the bootkit's persistence requirement (see requirements 2 and 3 in Section 2). This prevention measure blocks the initial infection vector and further data persistence within the system's *dark regions*. We can prevent write operations to the *dark regions* or redirect them to another specially created shadow area, as some malware droppers may verify whether *dark region* modifications succeeded by rereading the corresponding disk sectors. Another more conservative measure is to kill the virtual environment upon detecting a bootkit infection attempt. AV solutions often use emulators to provide a contained environment for each scan of a suspicious file. The described detection and prevention techniques can thus be integrated with existing solutions. Altough the detection system can also detect infections on physical machines (non VMs), the prevention system can only be implemented in VMs.

5 Bootcamp

This section presents *Bootcamp*, a detection, analysis and prevention framework implementing the detection heuristics, as well as the prevention measure proposed in Section 4 and already used in production. *Bootcamp* is fully automated, hence no user interaction is required during malware analysis.

Dynamic Bootkit Analysis. Armed with the heuristics described in Section 4 we distinguish two phases for dynamic bootkit analysis: a *bootkit infection phase* and an *bootkit execution phase*. During the *infection phase* the malware dropper executes and potentially installs a bootkit. In this phase, the *dark region modification* heuristic precisely describes the changes and locations within the *dark regions* by the bootkit attack, i.e., the modifications for the initial infection vector and its additional code and data. Preventive measures apply during the *infection phase* to stop bootkit attacks. After a certain time, we automatically restart the infected system to enter the *bootkit execution phase*. In this phase, we monitor the bootkit's behavior during the system's boot process. Therefore, we apply all previously described heuristics to detect, describe and analyze bootkit activity during the bootstrap process.

Architecture. Figure 7 outlines *Bootcamp*'s components. Malicious samples enter a central database via the *submission* module. The *Bootcamp server* distributes the malicious files to *Bootcamp workers* which in turn start virtual machines (VM) to analyze the samples dynamically. Each *worker* starts a different virtual analysis environment per sample and reports the analysis' results into the database. The *worker* performs the *bootkit infection phase* for each sample while recording the sample's infection behavior. After restart, the infected virtual environment enters the *bootkit execution phase* applying the bootkit detection heuristics to detect and analyze the potentially installed bootkit. In case prevention measures are applied, the *infection phase* also implements the prevention component. The analysis does not rely on any knowledge from the OS running inside the VM and all monitoring takes place in the emulated hardware.

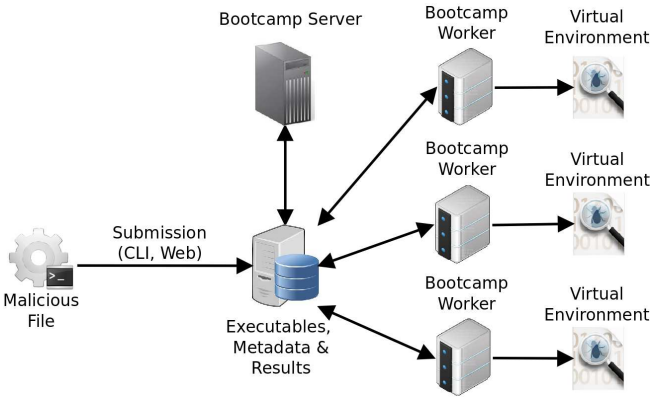

Fig. 7. Bootcamp's architecture

6 Bootcamp evaluation

To evaluate *Bootcamp* we utilized 32 bit Windows XP and Win7, with their standard bootloaders *NTLDR* and *bootmgr*, respectively, and used the same experimental setup and dataset as described in Section 3.

6.1 Bootkit Detection Results

Table 10 shows the results for our proposed bootkit detection heuristics. The *dark region read* heuristic caught all bootkit infections on both OSes. This behavior also correlates with our observation highlighted in Table 6 as every bootkit wrote to at least one *dark regions* not responsible for the boot process (MBR, VBR, BL). The *interrupt hook* heuristic also performed extremely well as it triggered for 98.8% samples on XP and 99.9% on Win7. This heuristic missed infections not employing interrupt hooking as described in Subsection 3.2. The *reuse existing code* heuristic caught 86.2% of infections on average. This heuristic missed samples infecting the bootloader stage as those do not execute a malicious MBR / VBR, followed by the benign MBR / VBR and therefore they execute the address `0000h:7C00h` just twice. Furthermore, we analyzed 100 benign samples utilizing our detection system and monitored the results during one hour of typical office usage (web browsing, text processing). To see the impact on the detection heuristics of a different bootloader, we ran Linux with *GRUB*. We did not encounter any false positives for these scenarios as writes outside partitions are very unlikely for benign samples respectively during normal system operation. There are very few executables potentially writing outside partitions (tools to repartition a system or correct errors in the MBR/VBR/BL) like *gparted* or *fixmbr.exe*. Thus *Bootcamp*'s heuristics work with setups that use 3 major bootloaders: *NTLDR*, *bootmgr* and *GRUB*.

Detection heuristic	XP	XP (%)	Win7	Win7 (%)	Avg
Dark region reads	1,143	**100.0%**	1,647	**100.0%**	**100.0%**
Interrupt hooks	1,129	**98.8%**	1,646	**99.9%**	**99.5%**
Reuse existing code	951	83.2%	1,455	88.4%	86.2%
Dark region writes (during bootkit infection phase)	1,143	**100.0%**	1,647	**100.0%**	**100.0%**

Table 10. Performance of proposed detection heuristics

6.2 Bootkit Prevention Results

Finally we evaluated the dataset applying the prevention technique described in Section 4.2. The experiment was performed for all 2,424 samples with *BK like behavior* on XP or Win7. During *bootkit infection phase* the system prevented the samples from writing outside of partitions and therefore hindering the bootkit's infection. After restarting the system and entering the *bootkit execution phase* we monitored the system's boot behavior. Applying the detection heuristics we determined whether an infection occurred in spite of the prevention system being in place. The system did not boot successfully after executing the malware in 2.9% of the cases on Windows XP and 7.3% on Win7. This was caused by the malicious modifications to essential OS components located inside the filesystem. After applying the detection heuristics on successful booting systems, we did not detect any infections during the boot phase anymore i.e. for all successful booting systems on both operating systems none of the detection heuristics triggered in any case. Applying our detection heuristics, the prevention system prohibited all 2,424 bootkit infection attempts on both OSes.

7 Discussion & Limitations

General Limitations. *Bootcamp* works for UEFI systems for the *bootkit infection phase* utilizing the heuristic for dark region tracking. The bootkit execution phase heuristics may not be effective in this case and we did not evaluate this scenario, as there are virtually no UEFI bootkits being not research proof-of-concepts. Hence, we evaluated our approach with MBR/VBR and BL-based bootkits which are the most popular bootkit types in the wild. Our heuristics are still relevant in a UEFI world for legacy systems that cannot be migrated. Bootkits may evade our system by directly attacking the BIOS, trying to flash a malicious one into the system and refraining from writing to boot process sectors. The malware can refrain attacking the boot process and stay inside the filesystem, e.g., by replacing system components directly on the filesystem like classic viruses do. However, this technique leaves suspicious tracks on the filesystem.
Evasion Techniques. There are several problems inherent to all dynamic analysis systems. Malware samples can employ techniques to detect a system emulator [29] and decline malicious behavior. Malware could fingerprint our analysis environment e.g. by MAC / IP addresses, Windows serial keys, deployed hard-/software combinations. For now we do not perform any cloaking of our system.

Additionally the sample might wait some time before performing the infection. Though Kolbitsch et al. proposed a stalling code detection and mitigation [20] it can not deal with all kinds of this awkward behavior. It is extraordinarily difficult to circumvent such behavior within limited time and resource constraints. Therefore, we are not able to monitor malware utilizing such techniques. Evasive techniques for the heuristics are already discussed in Section 4.1.

Future Bootkit Evolution. We estimate the following future bootkit evolution trends: 1) Based on our results in Section 3, we expect the majority of bootkits exploiting the **bootloader** as the initial infection vector in the future as it is stealthier and offers slightly more space compared to the MBR and VBR infections. 2) Figure 5 shows a **diversification of dark region utilization** with time and we expect this to expand to sections within the filesystem (e.g., metadata, alternate data streams, file slack space). But such techniques introduce more complexity and risk for the bootkit. For example it could be accidentally overwritten by the OS, as it is not aware of the disk sectors used by the bootkit. Furthermore, the bootkit would have to be aware of the filesystem's implementation, know the location of every single utilized file and its slack space and monitor move, copy, update and delete operations on all utilized files to keep the data consistent. 3) **BIOS/UEFI based bootkits** might be another future trend. In 2011 the first BIOS based bootkit was detected in the wild [13], but hardly spotted since then because they are highly complex to implement and have to consider BIOS vendor and version specific details for their attack to succeed. Though a few UEFI PoC bootkits exist, to the best of our knowledge none has been spotted in the wild ever. Still they might be a future trend. 4) We expect bootkits to increase their capabilities by using recent CPU features, e.g. advanced virtualization instructions. This would enable **VM based bootkits** applying ideas from SubVirt [18]. Already new bootkits sometimes inject kernel drivers in order to hide bootkit data in DRs from detection systems inside the infected host (e.g., AV software). Utilizing VM based bootkits could shift the whole infected system into a VM leveraging the concealment to a new level, as such systems could get along without injecting any code or data into the infected system, but completely control the whole machine via the malicious VM.

8 Related Work

Dynamic Malware Analysis. Researchers have proposed and developed many approaches for dynamic malware analysis. Most rely on sandboxing and execute malware in virtual environments, e.g. Anubis [9], Argos [28] or CWSandbox [38]. Most approaches focus on capturing Windows API calls in user mode as Anubis [9] or CWSandbox [38] do, while others target kernel level infections like K-Tracer [22], dAnubis [27]. Some target network traffic produced by malware like Sandnet [33], whereas others focus on analysis-based evasive malware detection [19]. None target bootkits or analyze them in detail. Large-scale malware studies were performed by many papers for various goals. Most focus on scanning and in depth analysis of malware samples, e.g. [7, 28, 38]. Other target the

problem of scaling for large malware datasets like [8, 16] do. However, none of those papers focus on the subset of malware that contains bootkit technology.
Bootkits. Li et al. [23] and Gao et al. [12] present a survey on bootkit attacking approaches like BIOS-, MBR-, NTLDR- or other technology based bootkit infection vectors. Rodionov and Matrosov [11] outline a bootkit classification based on their infection vector (MBR / VBR / BL) and present a bootkit threat evolution overview. This evolution survey is not built on certain bootkit technology characteristics as our historic evolution overview is, but highlights the first occurrence of certain technologies. Haukli [15] uses the number of jumps to address `0000h:7C00h` during the system's boot process as a heuristic to detect suspicious behavior but does not perform an evaluation of this approach. In 2007 IceLord, the first BIOS based bootkit PoC, was published which tries to inject a malicious BIOS and gains control even before the MBR is executed. Research in this area was done by Schlaikjer [31] and Wojtczuk and Tereshkin [39] in 2009 and 2013. None of them presented a historic, large-scale and detailed bootkit analysis as we do. Kaspersky Labs [17] studied GrayFish, an advanced malware created by the Equation group. GrayFish has a bootkit component that modifies the VBR and hijacks the loading of the first kernel driver. More components are loaded afterwards from the registry. Our detection heuristics would detect GrayFish's VBR component and the prevention component would stop the infection.
Preventing Infections. [14] proposed a bootkit prevention system based on an AV solution and relies on components inside of the system. Hence, an attacker can disable the system before performing the infection. This drawback does not apply to our proposed bootkit infection prevention approach. Bacs et al. [6] used low level disk monitoring to recover from infections, including bootkit attacks, and prevent persistent malicious storage inside as well as outside of the filesystem. NICKLE [30], a lightweight VMM, attempts to prevent unauthorized kernel code execution by using shadow memory. This technique may prevent bootkit kernel level code execution but not necessarily the infection. Compared to other defensive measures like the sometimes controversial trusted computing/boot or TPM approaches our solution does not require UEFI or additional hardware.

9 Conclusion

We presented a large-scale bootkit analysis and proposed detection and prevention mechanisms. We showed the results of a large scale bootkit analysis for a malware dataset composed of 25,513 samples collected over the last 8 years, whereof 2,424 samples revealed *bootkit like behavior*. The results give insights into specific bootkit behavior and show the evolution of bootkit technology from 2006 until now. We detected a major shift from exploiting the MBR as infection vector to utilizing the bootloader instead. The same applies for interrupt hooking where we detected a movement from using interrupt `0x13` to `0x15` as preferred hooking target throughout the years. Additionally recent bootkits try to hide their presence by exploiting the space between partitions instead of occupying the space at the very end of the disk. Furthermore, we detected that

every bootkit in our dataset stores its data in *dark regions* outside partitions. We evaluated our proposed detection heuristics with our complete dataset which detected all bootkit infections. We did not observe any false-positives during our evaluation with benign samples and boot processes, though false-positives may occur as discussed in Section 4. Moreover, we showed that the proposed prevention approach has successfully stopped all bootkit infections from our dataset.

Acknowledgements. The research was partly funded by the COMET K1 program by the Austrian Research Funding Agency (FFG). Sponsored by the ERC StG "Rosetta" and NWO VICI "Dowsing" projects.

References

1. Plite bootkit. http://labs.bitdefender.com/2012/05/plite-rootkit-spies-on-gamers/.
2. BIOS Rootkit: Welcome home, my Lord! http://blog.csdn.net/icelord/article/details/1604884, 2007.
3. Backdoor.pihar, 2011. http://www.symantec.com/security_response/writeup.jsp?docid=2011-120817-1417-99\&tabid=2.
4. Finfisher malware dropper analysis, 2014. https://www.codeandsec.com/FinFisher-Malware-Dropper-Analysis.
5. S. Aditya and B. Rohit. Prosecting the citadel botnet revealing the dominance of the zeus descendent. *VB*, 2014.
6. A. Bacs, R. Vermeulen, A. Slowinska, and H. Bos. System-level support for intrusion recovery. In *DIMVA*, 2012.
7. U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *USENIX LEET*, 2009.
8. U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *ACM SAC*, 2010.
9. U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *EICAR 2006*, 2006.
10. R. Dela Paz. ZeuS Source Code Leaked, Now What? http://blog.trendmicro.com/trendlabs-security-intelligence/the-zeus-source-code-leaked-now-what/, 2013.
11. D. H. Eugene Rodionov, Aleksandr Matrosov. Bootkits: Past, Present and Future. In *VB Conference*, 2014.
12. H. Gao, Q. Li, Y. Zhu, W. Wang, and L. Zhou. Research on the working mechanism of bootkit. In *ICIDT*, 2012.
13. M. Giuliani. Mebromi: the first bios rootkit in the wild. http://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/, 2011.
14. B. Grill, C. Platzer, and J. Eckel. A practical approach for generic bootkit detection and prevention. In *EUROSEC*, 2014.
15. L. Haukli. Exposing bootkits with bios emulation. *Black Hat US*, 2014.
16. X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *ACM CCS*, 2009.
17. Kaspersky Lab. Equation group: Questions and answers, 2015. https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf.
18. S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE S&P*, 2006.

19. D. Kirat, G. Vigna, and C. Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *USENIX Security*, 2014.
20. C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *ACM CCS*, 2011.
21. B. Krebs. Carberp source code leak. `https://krebsonsecurity.com/tag/carberp-source-code-leak/`, 2013.
22. A. Lanzi, M. I. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *NDSS*, 2009.
23. X. Li, Y. Wen, M. Huang, and Q. Liu. An overview of bootkit attacking approaches. In *MSN*, 2011.
24. A. Matrosov. ESET - Rovnix bootkit framework updated. `http://www.welivesecurity.com/2012/07/13/rovnix-bootkit-framework-updated`, 2012.
25. Microsoft. Kernel Patch Protection. http://msdn.microsoft.com/en-us/library/windows/hardware/Dn613955
26. MSDN. Driver Signing. http://msdn.microsoft.com/en-us/library/windows/hardware/ff544865(v=vs.85).aspx.
27. M. Neugschwandtner, C. Platzer, M. Comparetti, and U. Bayer. Danubis–dynamic device driver analysis based on virtual machine introspection. In *DIMVA*. 2010.
28. G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys*, 2006.
29. T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Information Security*. 2007.
30. R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID*, 2008.
31. S. Ross. Overview of bios rootkits, 2013. `https://tuftsdev.github.io/DefenseOfTheDarkArts/students_works/final_project/rschlaikjer.pdf`.
32. C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos. Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets. In *IEEE S&P*, 2013.
33. C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. Van Steen, F. C. Freiling, and N. Pohlmann. Sandnet: Network traffic analysis of malicious software. In *BADGERS*, 2011.
34. C. Rossow, C. J. Dietrich, C. Kreibich, C. Grier, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook . In *IEEE S&P*, 2012.
35. V. Rusakov and S. Golovanov. Attacks before startup. `http://securelist.com/blog/research/63725/attacks-before-system-startup/`, 2014.
36. B. Sean. First zeus, now spyeye. `https://www.damballa.com/first-zeus-now-spyeye-look-the-source-code-now/`.
37. S. R. White, J. O. Kephart, and D. M. Chess. Computer viruses: A global perspective. In *Virus Bulletin International Conference*, 1995.
38. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE S&P*, 2007.
39. R. Wojtczuk and A. Tereshkin. Attacking intel bios. In *BlackHat US*, 2009.