A Honeypot for Arbitrary Malware on USB Storage Devices

Sebastian Poeplau
University of Bonn
Institute of Computer Science 4
Friedrich-Ebert-Allee 144
53113 Bonn

Jan Gassen
Elmar Gerhards-Padilla
Fraunhofer FKIE
Cyber Defense research group
Friedrich-Ebert-Allee 144
53113 Bonn

***Abstract*—Malware is a serious threat for modern information technology. It is therefore vital to be able to detect and analyze such malicious software in order to develop contermeasures. Honeypots are a tool supporting that task—they collect malware samples for analysis. Unfortunately, existing honeypots concentrate on malware that spreads over networks, thus missing any malware that does not use a network for propagation.**

A popular network-independent technique for malware to spread is copying itself to USB flash drives. In this article we present Ghost, a new kind of honeypot for such USB malware. It detects malware by simulating a removable device in software, thereby tricking malware into copying itself to the virtual device. We explain the concept in detail and evaluate it using samples of wide-spread malware. We conclude that this new approach works reliably even for sophisticated malware, thus rendering the concept a promising new idea.

I. Introduction

In this article we will deal with the topic of malware – malicious software that is executed illegally on machines not owned by the authors of that software in order to achieve some kind of political or economical benefit. Although computer security is continuously improved, malware imposes a severe threat not only on common computer users, but also on companies, governments and the Internet as a whole. Once infected by malware, a computer may be bound to behave however the author of that particular piece of malware likes it to. An infected machine might, for example, disclose confidential information or personal data [1], participate in distributed attacks against computer systems, or it might sabotage computer-controlled industrial processes [2].

Malware is a serious threat in today's interconnected information technology: The Internet provides a convenient way for malicious software to spread fast across the world, and there are further means of propagation, as we will discuss later. The anti-virus software vendor McAfee has captured more than 20 million new unique malware samples in the year 2011 [3]. Thus, it is important to encounter the threat of malware by developing ever more sophisticated defense mechanisms. This, in turn, requires extensive insights into current malware technologies like spreading mechanisms, ways of infection and rootkit techniques.

In order to collect samples of malware for analysis, honeypots are employed – automated systems, that attempt to be infected usually by pretending to be vulnerable computers or services. Most commonly, honeypots are run on dedicated systems, i. e. machines that are maintained exclusively for the purpose of capturing malware. Honeypots enable researchers and anti-virus companies to analyze new types of malware and to develop countermeasures, and they are vital in generating signatures of previously unknown malware. Even though there are various different honeypot concepts, they all face one basic problem: How to trick malware into infecting a honeypot machine?

Many of the employed concepts make use of the fact that modern malware often spreads across networks, in particular the Internet. Those honeypots often imitate vulnerable network services or even whole operating systems in order to catch an infection over the network. This approach has proved to successfully attract malware: By exposing a seemingly vulnerable machine to the Internet openly detailed insights can be gathered, such as information about the security vulnerability that a certain malware exploits, its techniques for hiding on an infected system or the goals it pursues on that machine. Also, by collecting statistical data from incoming network traffic it is possible to estimate the number of hosts that are infected with some malware.

But some kinds of malware, among them very well-known recent malware families such as Conficker [4], [5] Stuxnet [2] and Flame [6], do not only use networks as their medium of choice for spreading. They also copy themselves onto removable devices, which enables them to reach even hosts that are not connected to any network at all. Larimer shows in his 2011 paper a variety of different attack approaches using removable devices [7]. Also, according to Microsoft's Security Intelligence Report [8], more than one quarter of the malware that was detected by the Malicious Software Removal Tool in the first half of the year 2011 was able to exploit the Windows autorun feature from a USB flash drive. For targeted attacks, i. e. if a certain malware aims to reach a predetermined system for some reason, it might even be crucial not to depend solely on the internet for propagation. Stuxnet, for example, was alleged to target industrial control systems that are not connected to the Internet for security reasons. In such cases, malware cannot reach its target machine without additional propagation methods.

978-1-4673-3089-3/12/$31.00 ©2012 IEEE

This results in a major issue. As long as malware spreads via both, networks and removable devices, our traditional network-based honeypots can capture it. In contrast, if some malware only uses removable devices for propagation, those kinds of honeypots are completely unable to detect it. The implications are severe: Anti-virus companies can only generate signatures for malware if their honeypots eventually collect samples of it. But if malware evades all honeypots, no specialised protection can be provided.

That is one of the reasons for us to introduce a new type of honeypot which concentrates on malware that spreads via removable storage devices. Consider again the example of Stuxnet: As long as the malware was not known, it could spread in networks freely, not being recognized by intrusion detection systems or anti-virus scanners. If it had been possible to detect the infection of machines by using a honeypot such as the one we propose here, certainly less computers would have been infected before the malware became publicly known. Another advantage of the approach is that focussing on USB malware significantly simplifies the process of collecting the actual malware executable, as we will show within this article.

By mounting a virtual USB flash driver on the infected system we will be able to detect the infection and capture a sample of the malware without having any knowledge about it – the only assumption is that it infects removable storage devices. Since the machines that are likely to be targeted by malware on USB devices are not dedicated honeypot systems but rather productively used machines, our honeypot will be deployed to computers that are used actively. Note that for the concept to work we need the honeypot system to be infected with malware, so our honeypot can be seen as the last line of defense: If all other protection mechanisms have failed, then we can detect the infection and take immediate action. Obviously, such a kind of honeypot is not going to capture as many malware samples as a system that follows a more traditional approach, but especially on highly threatened machines (e. g. publicly accessible ones) it can detect otherwise unnoticed malware and thus provides for excellent intrusion detection.

The article is organised as follows: Section II outlines related work in the field and compares it to the ideas that we introduce. Section III works out the concept of our honeypot and describes Ghost, its implementation. In section IV we evaluate the honeypot with various samples of malware and draw conclusions from the evaluation. Possible further work is discussed in section V, and finally we summarize the results in section VI.

II. Related Work

In this section we will discuss what has been done already and why the work presented in this text is necessary. We will cover classical honeypots as well as tools that can be used to create virtual devices.

There are many honeypots, but most of them only target malware that spreads across networks. Dionaea [9] and Nepenthes [10], for example, emulate certain network services and thereby capture malware that tries to attack those services, but they are not able to deal with malware that does not spread across the network. Similarly, honeytrap [11] listens for incoming network connections, but does not deal with malware on removable storage devices.

Argos [12] follows another approach: It emulates a whole operating system and tracks the path of all data received from the network through the system. If at some point such data or a variation of it is executed as machine code, Argos is able to detect the infection. Tracking of incoming data is restricted to network data though, so that infections of the virtual machine by USB devices are not detected. Theoretically, Argos could be extended to also track data from removable storage devices, but then other problems would arise: Many malware samples rely on social engineering in order to infect a machine from a removable device. But Argos would not be able to provide such interaction in an automated fashion. Furthermore, since our honeypot is going to be deployed on productive machines, we rely on those machines' operating systems. So in many cases the honeypot will run on Windows machines. The Argos virtual machine would have to be provided with the contents of the removable device by its host, but we cannot expect a Windows host to be reliable in such a situation – as the host itself is a possible target for the malware, it might be infected and the files on the USB flash drive would subsequently be hidden hidden by a rootkit. We could use other host operating systems, but Windows systems are the ones that USB memory sticks are plugged into frequently in most organizations, and it is less convenient to have to connect each USB flash drive to a dedicated honeypot machine first before plugging it into the actual productive machine. Also, Argos's approach of emulating a whole operating system introduces overhead that makes it unsuitable for use on production computers and rather requires a dedicated honeypot machine.

The approach proposed in this article requires emulation of USB storage devices. There are several solutions that provide storage device emulation, but none of them are able to emulate *removable* storage devices:

- DaemonTools [13] is a well-known emulator of CD and DVD drives, but it cannot emulate other types of storage devices and therefore does not meet the requirements for our concept.
- FileDisk [14] can emulate hard disks. Unfortunately, the driver that is part of its implementation is not compliant with the Windows Driver Model [15] and, as a result, does not support plug and play. But since plug and play is an essential concept when emulating a USB flash drive, the software cannot be used for this purpose.
- The Windows Driver Kit by Microsoft contains the driver Ramdisk [16], which emulates a hard disk backed by a data structure in the computer's main memory. However, the emulated hard disk is mounted at system startup, which again prevents realistic emulation of a removable storage device.

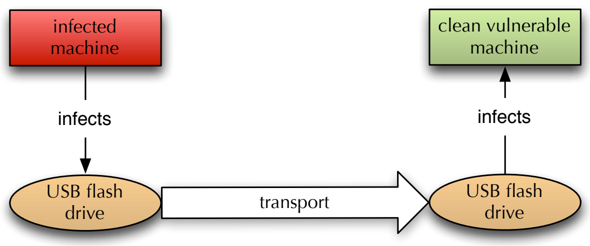

Figure 1. The infection process for USB malware.

III. Concept

In this section we first describe the concept of a honeypot for malware on removable devices before we detail the actual implementation. It is important to keep in mind that no assumptions have to be made about the target malware except that it is able to spread via USB devices.

A. *The Idea*

The goal of any honeypot is to collect information like insights into the malware itself, information about its author or about infections with that particular malware. In our case, the effort is targeted at learning about an infection of the honeypot machine in the first place and at obtaining a sample of the malicious software.

There are different points in the lifecycle of a malicious piece of software at which a honeypot can try to gather a copy of the executable malware files. In this context, the term "lifecycle" refers to the process of malware infecting a machine, hiding itself on the system, executing unwanted activities such as sending spam or collecting personal data, and ultimately infecting other machines.

Propagation, i. e. infection of other machines, is the phase that many honeypots interested in self-spreading malware aim at, as has been described in section II. One of the reasons is that propagation is an often-seen characteristic of such malware. However, much effort is put into capturing malware spreading across networks, while there is little consideration of other mediums.

Especially Stuxnet has shown that spreading via removable devices is an important characteristic of recent malware [2]: In order to achieve its alleged goal – manipulation of industrial facilities – it was vital for Stuxnet to infect machines that were not connected to a network. As USB storage devices are cheap and easy to use, they are widely employed for data exchange with such disconnected machines and provide an excellent way for malware to infect those systems. Stuxnet is able to infect machines via files that it previously copied to USB storage devices when they were plugged into infected machines. Figure 1 illustrates the process.

This demonstrates the need for a honeypot that targets USB malware and gives us a possible way to construct such a honeypot. Since USB malware – like its network-oriented counterparts – propagates to other machines at some point in its lifecycle, we can build a honeypot that exploits the general behavior of infecting removable devices for propagation.

However, it is not feasible to connect physical removable devices to a pool of potentially infected machines periodically in order to try and detect an infection. Therefore, we propose the implementation of a virtual USB storage device that can be connected to threatened machines on a regular basis. The implementation of the virtual device must be such that malware mistakes it for a real removable storage device and therefore infects it.

Unfortunately, the approach implies two drawbacks:

1) A machine has to be infected before we can capture any malware.
2) As the honeypot emulates its virtual device on an infected machine, it is possible for the malware to detect and subvert the honeypot, provided that it undertakes sufficient efforts – as is also the case with host-based intrusion detection systems.

Those two points emphasize that the system is not quite a typical honeypot but also carries characteristics of an intrusion detection system. We cannot expect it to capture unknown malware before any harm is done, but it provides us with a means to learn about infections that would otherwise remain completely undetected. Thus it serves as a final defense after other protection schemes have failed.

Knowing about this basic difference between common honeypots and the system proposed here, the concept of a virtual USB flash drive promises substantial benefits:

1) It provides a means of host-based intrusion detection with very low false-positive rate.
2) If malware infects the virtual device, we are likely to be provided with all executables of the malware.

In order to record a false positive, two scenarios are possible. Either some application legitimately writes data to newly connected removable storage devices, or the user accidentally copies files to the virtual device. Although the former is possible, it is not at all common behavior of software to write data to any newly connected USB flash drive. The latter, however, has to be avoided by the honeypot. So the virtual USB device must be hidden from the user in order to prevent accidental write operations. Section V discusses different approaches to achieve this. Obviously, the shorter the virtual device is mounted the easier it gets to hide the whole process from the user. Fortunately, we will see in section IV that the amount of time during which the device must be mounted is relatively small.

Since the purpose of the files written to the USB storage device is infection of other hosts, we will find all necessary executables on the device most of the time. Theoretically, it would be possible for malware to only write a loader to the USB fash drive which then downloads the actual malware from some network location. But such an approach would require the target machine to have network access (which cannot be guaranteed if the infection is conducted via a USB memory stick) and has not yet been seen in practice. So with high probability we are provided with all files, and if not so then at least we learn where to find the remaining data.

The idea that we have discussed so far requires a restructuring of the traditional honeypot infrastructure: With traditional honeypots it is common to have dedicated machines that run them, machines that solely exist for the purpose of detecting malware. But we need to run the virtual USB drive on machines that are used frequently, and thus the honeypot software has to function transparently on machines that are in productive day-to-day use, e. g. office computers in some company. After all, USB storage devices are frequently plugged into production machines, but users would not like to undertake the effort of connecting each and every such device to a dedicated honeypot machine beforehand. The notion of a virtual USB memory stick as honeypot, though, crucially depends on the host machine being infected, so that we can consider it a kind of intrusion detection system rather than a pure honeypot.

When discussing the concrete implementation, we will show that it is possible to implement the above in a transparent, reliable and efficient manner. Furthermore, the evaluation will demonstrate that modern malware is not capable of detecting such a honeypot (see section IV).

B. The Implementation

After we have covered the theoretical basics of the new concept, we will now discuss how the proposed system can be implemented. We chose Windows XP as target system for our implementation, because despite its age it is still widely used and targeted by many pieces of malware [17]. However, the code can be extended to work on other versions of Windows with little effort.

The challenge of implementing a virtual USB flash drive is twofold: First, we would like to have virtual storage, i. e. an emulated storage device that is backed by an image file. Applications must be able to write data to and read from the device, while we route all those I/O operations to an image file that comprises our storage. Secondly, the device has to look exactly like a *removable* storage device to any application that queries information about it. This is where the available solutions do not provide sufficient flexibility – tools like FileDisk [14] emulate a hard disk but cannot make it appear as a removable device. Of course, the implementation of the honeypot must be complex enough to provide such features, but we do not want it to be more complex than necessary. Therefore, different approaches are considered for achieving the two main goals stated above: simulating storage and making it look removable.

Microsoft provides the Device Simulation Framework [18], a software framework that simplifies simulation of USB devices. It allows to execute simulation code in user mode and routes it to the kernel by using a custom driver. However, the main intention of the framework seems to be driver testing. Also, it requires an implementation of the simulated device at a very low level, which is not necessary for the concept presented here. Furthermore, licensing issues would have to be considered if the Device Simulation Framework were to be part of the honeypot, so this is not a practical option.

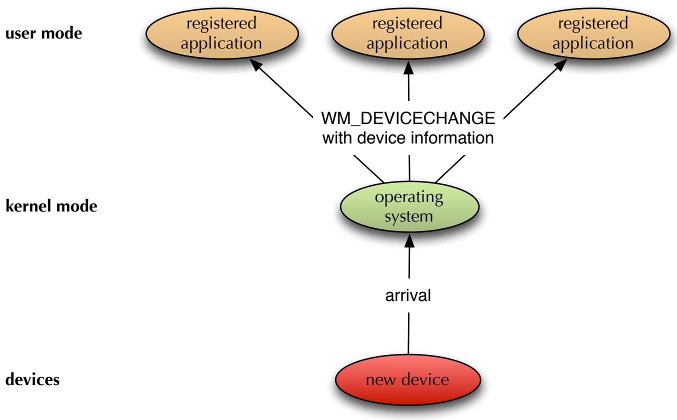

Figure 2. Device detection by using the window message WM_DEVICECHANGE.

In order to find out how to achieve the goal of maximum resemblance to real removable devices in a custom implementation, we first examine the possibilities for software to notice when new removable storage is plugged in. A common way of doing so is to register for the window message WM_DEVICECHANGE (see [19] for an implementation). The operating system sends this message to registered applications whenever new devices are attached. The receiving application can examine the message's additional parameters to find out whether the newly attached device is a removable storage device (see figure 2 for an illustration). This technique is used by Conficker [4] and Stuxnet [2]. Another (less elegant) way of learning about new storage devices is to regularly poll all available drive names (usually C: through Z:) and query whether the underlying devices are removable. Our virtual device has to be implemented such that malware will find it using either technique.

Before we can discuss how to make a virtual device appear removable in those respects, we need to elaborate a little more on the handling of devices within the Windows kernel: Devices are represented by device objects, and each device object is managed by a device driver. Upon connection of new devices, Windows decides which driver to load based on the device identifier of the newly connected device. The driver that is loaded may subsequently create a device object itself and provide it with a certain device identifier. Again, a driver is loaded for that device based on the identifier, and so on. This leads to device objects and associated drivers being stacked on each other, where each driver processes queries to its device object and then passes them on to the next lower device object's driver. On the lower layers of the driver stack for a storage device we usually find the respective bus driver (e. g. IDE, USB), somewhere above that the disk class driver (a generic driver for storage of all kinds) and associated systems reside, and on the very top a file system driver is loaded in order to give user space applications access to the device.

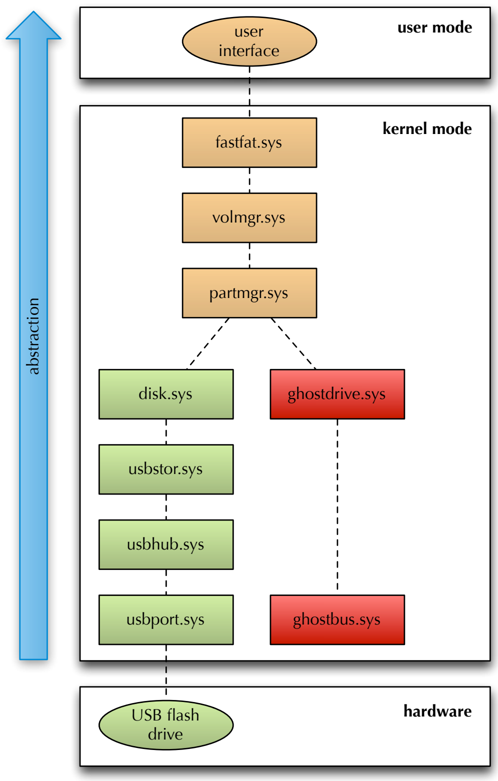

Figure 3. Driver stack illustrating the hooks of the Ghost honeypot (based on [20]).

Our goal is to make a particular device seem removable to the operating system. It turns out that the notion of removability in Windows is established by the disk class driver. It determines whether a storage device is classified as removable or not and thus decides whether the device is seen as a removable device by applications or not. Therefore, we decided to implement the virtual flash drive as a Windows driver that operates on the same level as the disk class driver and flags the virtual device removable: ghostdrive.sys. Figure 3 compares the driver stacks for real USB storage devices to our virtual device. The driver ghostdrive.sys replaces the disk class driver for the virtual USB flash drive and makes sure that it is reported as a removable storage device to the more high-level drivers. From user space the device then looks very similar to a real removable storage device. Of course, close inspection of a storage device's driver stack can reveal a USB drive's true nature, but such a check requires considerably more effort than the simple user mode API calls described above.

The second driver in figure 3, ghostbus.sys, still requires some explanation. It is related to *device enumeration*. Generally speaking, new devices have to be announced to the Windows kernel somehow in order for drivers to be loaded. This process is called device enumeration. For drivers compliant with the Windows Driver Model [15], there are two options:

1) A device can be *root-enumerated*. That is, a device with a specified identifier is assumed to be present at system startup, and the required drivers are loaded accordingly.

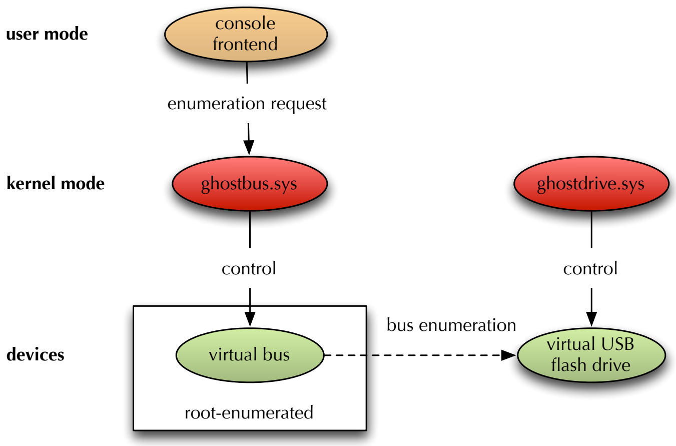

Figure 4. Basic components of the Ghost honeypot.

The PCI bus driver is an example of a driver that is loaded this way.

2) Devices can be enumerated by other device objects (*bus enumeration*). If the driver of some existing device object creates a new device object, then Windows loads the appropriate driver. Obviously, such a chain of enumerations has to start somewhere, which is where root-enumeration is needed. Examples of drivers loaded by bus enumeration include USB drivers (initially enumerated by the driver of the PCI bus that the USB controller is attached to) and the disk class driver (enumerated by the driver for the respective bus that a storage device connects to).

The virtual USB drive is supposed to be mountable on demand, so we cannot accept its driver to be loaded at startup by root-enumeration. The solution is a second driver, ghostbus.sys: It is loaded by root-enumeration and represents a virtual bus that can report new virtual USB flash drives on demand. So whenever we would like to mount the virtual device, we instruct the virtual bus driver to report a new device with a certain device identifier for which Windows then loads the driver ghostdrive.sys. In the current implementation this is done via a console frontend, which is sufficient for the purpose of demonstrating that the system works, but more sophisticated techniques are possible (see section V). Figure 4 shows the overall structure of the implementation as described above.

Detection of infections and capture of malware samples now work as follows: The virtual bus driver is instructed to load the virtual storage device, which is reported to the system as a removable device. The malware, upon noticing the arrival of a new removable drive, infects that device by writing data to it. The malware's write requests are routed down the driver stack and eventually passed on to the virtual USB memory stick's driver, ghostdrive.sys, which now does two things: Since data was written to the removable device, we can assume an infection and thus triggers some sort of user notification. Also, the driver writes all data to an image file for later analysis and to be able to properly serve read requests issued by the malware (e. g. for double-checking success of the write operation). After some time (which will be made more precise by the evaluation) the virtual device is unmounted, and its contents can be sent to another machine for analysis.

It would be more convenient to analyze the virtual device's contents directly on the honeypot machine, but we have to be careful: Since the machine is infected at this stage, the malware has probably installed some kind of rootkit in order to hide its presence on the system. If we just mounted the image again and tried to access it with the usual APIs, we would most likely be shown bogus results. See section V for a discussion of a possible way around this issue.

IV. Evaluation

In this section we will evaluate the proposed concept based on the implementation presented above. All of the evaluation was conducted on a machine with Windows XP SP2 (English) installed on it, which was reset after each test run by replacing the hard disk contents with a prepared binary image. The focus of the evaluation lies on examining whether the concept works at all, whether given malware samples infect the virtual USB flash drive reliably and how long it takes until an infection takes place.

A. Test samples

The malware samples used for testing were mainly selected by one crucial property: A test sample must reliably infect a USB storage device that is attached to the infected machine. Since this is the basic assumption that the honeypot makes about its target malware, we require it to hold for all test samples. As part of the test set two samples of the Conficker malware were selected because they can be referred to as quite sophisticated pieces of malware. If they are not able to detect the honeypot, then this might indicate that even existing high-level malware (in terms of technical sophistication) can be tricked into infecting a virtual USB flash drive.

Table I shows the test set used for evaluation and the names that the samples are given by three different virus scanners. The reader might notice that Stuxnet, although mentioned frequently throughout this article, is not part of the set. The reason is that we were not able to make that particular malware infect even a physical USB memory stick reliably. The analysis by Symantec [2] suggests that a change in the malware's configuration file suffice to enable removable drive infection. However, the approach did not work for us, nor were we able to call the corresponding routines in the malware's DLL file directly. The DLL is only saved in an encrypted form and injected into selected processes at system startup. Thus we did not incorporate Stuxnet into the test set. However, if Stuxnet's infection routine proceeds as described by Symantec [4], then the malware is not able to differentiate between a physical USB flash drive and our virtual equivalent.

B. Procedure

The evaluation procedure works as follows: Select a sample $S \in M_T$ from the sample set (see table I) and infect the test machine with it. Then mount the virtual USB flash drive and wait for $D_M \in \mathbb{N}$ seconds before the virtual device is unmounted. D_M is called *mount duration*. Afterwards, the image that provided the backend for the virtual storage device is copied to a clean Unix machine and we check whether files have been written to it and whether similar were also written to the real USB memory stick when the sample was checked for USB infection initially. We call a run of the test procedure $P(S, D_M)$ *successful* if and only if the files written to the virtual device contain the same malware as those on the physical USB flash drive.

In order to make sure that malware samples infect the virtual USB memory stick reliably, we repeat the above procedure exactly three times for each sample. Let the results of these test procedures be $P_1(S, D_M)$, $P_2(S, D_M)$ and $P_3(S, D_M)$, respectively. We call the result of such a multiple execution $I_3(S, D_M)$ *successful* if and only if all three runs of the test procedure were successful.

$$\begin{aligned} & I_3(S, D_M) &= \text{successful} \\ \Leftrightarrow \forall i \in \{1, 2, 3\} : \quad & P_i(S, D_M) &= \text{successful} \end{aligned}$$

If we can trick the malware into infecting the virtual device reliably within some mount duration D_M, then there is a minimal mount duration for which this is possible:

$$D_{\min}(S) = \min\{D_M \in \mathbb{N} : I_3(S, D_M) = \text{successful}\}$$

The goal of this evaluation is to find $D_{\min}(S)$ for all test samples $S \in M_T$ (if it exists) in order to get an idea for how long the virtual flash drive must be mounted to be infected reliably.

C. Results

It turns out that $D_{\min}(S)$ exists for all $S \in M_T$, i. e. all test samples infect the virtual USB memory stick within some mount duration $D_{\min}(S)$. Furthermore, the mount duration that is necessary for the device to be infected is 35 seconds for the slowest sample – in most other cases it is much less (see figure 5 for the measured minimal mount durations). On average across the test set, an infection takes place after only 7.9 seconds. Especially in the case of Conficker the infection is completed within only one second.

Drawing a conclusion, the evaluation shows:

1) None of the malware samples is able to distinguish between the virtual USB flash and a real (i. e. physical) USB storage device. So the concept works as expected.
2) The minimal mount duration is low enough to hide the whole process from the user, thereby avoiding user-triggered write operations to the virtual device.

So we can conclude that the proposed concept of a virtual USB flash drive as honeypot for malware that spreads via USB storage devices works reliably and fast.

Table I
SET OF TEST SAMPLES M_T WITH RESPECTIVE CLASSIFICATION BY VIRUS SCANNERS.

Title	Classification by virus scanners		
	Kaspersky	Symantec	Microsoft
Sample A	Worm.Win32.AutoIt.r	W32.Badday.A	Worm:Win32/Yuner.A
Sample B	Worm.Win32.AutoRun.rwp	W32.Ircbrute	Worm:Win32/Hamweq.G
Sample C	Worm.Win32.AutoRun.nuh	Trojan.Packed.NsAnti	Worm:Win32/Taterf.B
Sample D	Worm.Win32.AutoRun.lkw	W32.Baki.C	Worm:Win32/Autorun.DX
Sample E	Worm.Win32.AutoRun.zki	W32.SillyFDC	Backdoor:Win32/Darkshell.A
Sample F	Virus.Win32.VB.ki	W32.Mikbaland	Worm:Win32/Autorun.CA
Conficker I	Net-Worm.Win32.Kido.ih	W32.Downadup.B	Worm:Win32/Conficker.B
Conficker II	Net-Worm.Win32.Kido.ih	W32.Downadup.B	Worm:Win32/Conficker.C

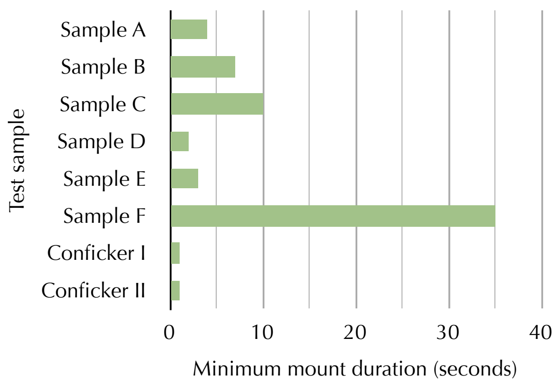

Figure 5. Minimum mount durations necessary for the sample to infect the virtual device.

V. Further work

The current implementation already works well, as was shown in the previous section. However, further work might include additional improvements.

User interaction with the virtual device must be avoided efficiently. If the user is shown the virtual device in the graphical user interface, they might accidentally copy files to it and thereby trigger a false alert. So the chance of the user interacting with the virtual USB memory stick must be minimized. A possible way to do so would be to only mount the virtual flash drive if the screen saver is active. It is unlikely that a user is working on a usual office computer in such a situation. Another option would be to instruct the operating system not to show the device in the user interface. However, such a setup could provide malware with an easy way to distinguish between the virtual device and a physical USB memory stick and thus would have to be analyzed carefully.

The files that are copied to the virtual device by malware could be analyzed on the honeypot system rather than transmitting the whole image file to some other machine. However, as mentioned before, rootkits might prevent analysis applications from reading back the data on an infected machine. In order to evade rootkits, the honeypot could be equipped with some kind of user mode file system driver. If the honeypot were able to parse the file system structures within the image file, then it would not have to rely on file system drivers supplied by the operating system which are likely to be manipulated by the malware. The approach would allow for checking whether malware on the virtual USB device is already known, in which case a further analysis of the files would be unneccessary and a simple alert that a machine is infected might be sufficient.

After the files that a certain malware writes to removable devices are known, the honeypot could hook the operating system's I/O routines and try to prevent the infection of further devices with those files until the malware is removed from the host machine. This is not originally a task for a honeypot, but as was mentioned before, the proposed system also shows characteristics of an intrusion detection system.

In order to detect malware in the future, we will have to hide the honeypot from malicious software that tries to detect whether it is about to infect a physical or an emulated device. Since we *emulate* a device, we will not be able to hide perfectly: Just like any other honeypot, our system can be fingerprinted. The goal, though, is to make it sufficiently hard for malware to detect the emulation. Since the honeypot operates in kernel mode and is installed before the malware, there is a considerable potential to defeat detection attempts. Possible approaches include file and device name randomization, rootkit techniques to hide the honeypot's files, random mount durations and working at a low level in the driver hierarchy, so that the honeypot's drivers are difficult to recognize from user space.

The next logical step after having shown that the concept works in a controlled environment would be to install the honeypot on a number of machines in order to find out how severely the problem of malware on USB devices threatens computer security. It would also be interesting to find out whether there actually is malware around that spreads via flash drives exclusively and thus has not been detected yet by traditional honeypots or intrusion detection systems.

VI. Summary

We have presented the concept of a honeypot targeted at malware that spreads via USB storage devices. We have evaluated the implementation of that concept and we have shown that even sophisticated recent malware is not able to distinguish the honeypot from any other infection target. Therefore, we can conclude that the concept is well-suited for detecting infections of computers and gathering samples of the infecting malware. Especially when considering the efforts of malicious software to avoid being caught by honeypots in networked environments, the idea of a virtual USB storage device promises to be a means of detecting infections and thus unknown malware with less effort than before. It enables organizations to learn about infections of their machines when conventional protection has failed and thus serves as a reliable intrusion detection system.

REFERENCES

[1] J. Baltazar, J. Costoya, and R. Flores, "The real face of koobface: The largest web 2.0 botnet explained," *Trend Micro Threat Research*, 2009.

[2] N. Falliere, L. Murchu, and E. Chien, "W32. stuxnet dossier," *Symantec Security Response*.

[3] McAfee Labs, "Mcafee threats report: Fourth quarter 2011," 2011.

[4] Symantec, "The downadup codex," Symantec, Tech. Rep., 2009.

[5] F. Leder and T. Werner, "Know your enemy: Containing conficker," *The Honeynet Project, University of Bonn, Germany, Tech. Rep*, 2009.

[6] Kaspersky, "Flame: Bunny, frog, munch and beetlejuice…" [Online]. Available: https://www.securelist.com/en/blog?weblogid=208193538#w208193538

[7] J. Larimer, "Beyond autorun: Exploiting vulnerabilities with removable storage," *Blackhat*, 2011.

[8] Microsoft, "Microsoft security intelligence report." [Online]. Available: http://www.microsoft.com/sir

[9] "dionaea – catches bugs." [Online]. Available: http://dionaea.carnivore.it/

[10] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *Recent Advances in Intrusion Detection*. Springer, 2006, pp. 165–184.

[11] T. Werner, "honeytrap - a dynamic meta-honeypot daemon." [Online]. Available: http://honeytrap.carnivore.it/

[12] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 15–27, 2006.

[13] DAEMON Tools, "Daemon tools lite." [Online]. Available: http://www.daemon-tools.cc/deu/products/dtLite

[14] B. Brantén, "Windows driver examples." [Online]. Available: http://www.acc.umu.se/~bosse/

[15] Microsoft, "Introduction to wdm." [Online]. Available: http://msdn.microsoft.com/en-us/library/ff548158%28v=VS.85%29.aspx

[16] ——, "Ramdisk." [Online]. Available: http://msdn.microsoft.com/en-us/library/ff544551%28v=vs.85%29.aspx

[17] Y. Namestnikov, "Kaspersky security bulletin. statistics 2011," 2011.

[18] Microsoft, "Device simulation framework design guide." [Online]. Available: http://msdn.microsoft.com/en-us/library/ff538293%28v=vs.85%29.aspx

[19] J. Dolinay, "Detecting usb drive removal in a c# program." [Online]. Available: http://www.codeproject.com/KB/system/DriveDetector.aspx

[20] Microsoft, "Device object example for a usb mass storage device." [Online]. Available: http://msdn.microsoft.com/en-us/library/ff552547%28v=VS.84%29.aspx