Bee Master: Detecting Host-Based Code Injection Attacks

Thomas Barabosch, Sebastian Eschweiler, Elmar Gerhards-Padilla

Fraunhofer FKIE,
Friedrich-Ebert-Allee 144, 53113 Bonn, Germany
{firstname.lastname}@fkie.fraunhofer.de
www.fkie.fraunhofer.de

Abstract. A technique commonly used by malware for hiding on a targeted system is the host-based code injection attack. It allows malware to execute its code in a foreign process space enabling it to operate covertly and access critical information of other processes. Since there exists a plethora of different ways for injecting and executing code in a foreign process space, a generic approach spanning all these possibilities is needed. Approaches just focussing on low-level operating system details (e.g. API hooking) do not suffice since the suspicious API set is constantly extended. Thus, approaches focussing on low level operating system details are prone to miss novel attacks. Furthermore, such approaches are restricted to intimate knowledge of exactly one operating system.
In this paper, we present *Bee Master*, a novel approach for detecting host-based code injection attacks. *Bee Master* applies the honeypot paradigm to OS processes and by that it does not rely on low-level OS details. The basic idea is to expose regular OS processes as a decoy to malware. Our approach focuses on concepts – such as threads or memory pages – present in every modern operating system. Therefore, *Bee Master* does not suffer from the drawbacks of low-level OS-based approaches. Furthermore, it allows OS independent detection of host-based code injection attacks. To test the capabilities of our approach, we evaluated *Bee Master* qualitatively and quantitatively on Microsoft Windows and Linux. The results show that it reaches reliable and robust detection for various current malware families.

Keywords: Host-Based Code Injection Attacks, Malware Detection, Computer Security

1 Introduction

In recent years the number of malware samples that we are facing each day steadily increased. Nowadays, cyber criminals use malware for a multitude of activities, e.g. credit card fraud or industrial espionage. Furthermore, malware developers have started to target new operating systems in addition to the classic one, Microsoft Windows. Mac OS X, Linux or Android are among the increasingly popular targets.

But not only the amount of malware and the breadth of their targeted platforms is increasing. Likewise, the number of techniques used by malware to cover its presence steadily increases. One of those techniques is the host-based code injection attack (HBCIA). HBCIAs enable malware to execute its code within the scope of a foreign process. This stands in contrast to the common belief that only one program is accountable for the behaviour of a process. From a malware author's point of view a code injection results in several benefits, amongst other avoiding detection by anti-virus software or intercepting critical information from within the targeted process like credit card information. Based on data by Symantec[1], four of the top five malware families in 2012 – Ramnit, Sality, Conficker and Virut – used HBCIAs. They were responsible for 32.1% of all new infection reports in this year. Note that this is only the tip of the iceberg and that there exist many more current malware families that employ HBCIAs.

In this paper we present *Bee Master*, a novel approach for detecting host-based code injection attacks. *Bee Master* detects HBCIAs by providing an environment vulnerable to those attacks and monitoring this environment for changes associated with HBCIAs. Thus, we apply the honeypot paradigm to the domain of operating system processes in order to detect host-based code injection attacks. The environment we provide is a set of operating system processes that we control. Almost every modern OS uses processes in order to manage the execution of computer programs. Therefore, our approach can be applied to a wide range of operating systems. Furthermore, it does neither depend on modification of the OS nor the hardware.

Due to the ever increasing malware flood and the inefficient signature-based approach used by anti-virus software, detection rates are very dissatisfying. This especially holds true for the detection rates of targeted attacks. Typically, targeted attacks slip through detection routines of anti-virus software due to being specially crafted for only one target. In 2012, it took a business on average 210 days for detecting that a breach occurred within their network[2]. By focusing on a feature which is wide-spread among todays malware, our approach can not only detect a significant portion of current mass-malware but it could also help detecting a significant amount of targeted attacks early that would otherwise have stayed under the radar for several months, with potentially severe consequences.

We have implemented *Bee Master* for Microsoft Windows as well as Linux and evaluated it. In quantitative and qualitative evaluations, we show that *Bee Master* is capable of detecting HBCIAs of current malware and is not limited to one operating system. Furthermore, we show in a study with several malware families that HBCIAs can be considered as an inherent feature, which is unlikely to change between versions and variants.

The contributions of this paper can be summarized in the following three key points:

(I) **HBCIA is an inherent malware family feature**
We show in an investigation on several malware families that host-based

code injection attacks are an inherent family feature which is unlikely to change between versions and variants of a malware family.

(II) **A novel approach for detecting HBCIAs**
We propose a novel and OS-independent approach for detecting host-based code injection attacks by applying the honeypot paradigm to OS processes.

(III) **Evaluation of a prototype with prevalent malware families**
We have implemented *Bee Master* for Microsoft Windows as well as Linux and show its feasibility in qualitative and quantitative evaluations with current and representative real-world malware families.

2 Code Injection Attacks

In this section we introduce code injection attacks. Firstly, we give a general definition of code injection attacks. Afterwards, we differentiate two different types, namely remote code injection attacks and host-based code injection attacks (HBCIA). This is followed by a closer look at HBCIAs. We conclude this section with a study on the presence of HBCIA in different versions and variants of selected HBCIA-employing malware families.

2.1 Definition of Code Injection Attacks

We give a general definition of a code injection attack in Definition 1.

Definition 1. *Let $\mathcal{E}_{attacker}$ be an entity controlled by an attacker. Let $\mathcal{P}_{victim}$ be a process targeted by the attacker. An active attack on $\mathcal{P}_{victim}$ by $\mathcal{E}_{attacker}$, that aims at executing a payload defined by $\mathcal{E}_{attacker}$ within the context of $\mathcal{P}_{victim}$ is called code injection attack.*

$\mathcal{E}_{attacker}$ can be any entity on a system that allows the attacker to execute the code that undertakes the code injection into $\mathcal{P}_{victim}$. Such entities include OS processes, kernel modules or even hardware devices. In the following, however, we assume that the entity used by the attacker is an OS process and therefore we will refer to $\mathcal{P}_{attacker}$.

There are two kinds of code injection attacks: host-based and remote code injection attacks. The first is limited to one computer system i.e. the attacking process $\mathcal{P}_{attacker}$ is executed on the same machine as $\mathcal{P}_{victim}$. Malware uses this kind of code injection intensively, e.g. for hiding purposes (cf. section 2.2). The latter code injection attack involves two systems: the attacker system and the victim system, which are interconnected by a network. The attacker sends a special payload to a network service – executed in the context of a victim process $\mathcal{P}_{victim}$ – of the victim via the network. This payload – called exploit – aims at triggering a software vulnerability in the addressed network service. In case the network service is vulnerable to the exploit, parts of the payload are executed within the victim′s network service process space. Many Internet worms use this technique as infection vector. However, our solutions solely focus on the detection of host-based code injection attacks.

The execution of code within a victim process $\mathcal{P}_{victim}$ usually has never been intended by the author of the underlying program. Even though there are some legitimate uses of code injections such as debugging or hot patching, based on our experience we believe that such benign code injections present only a very small fraction of all code injections.

2.2 Host-Based Code Injection Attacks

HBCIAs are used by all kinds of malware ranging from consumer-focused malware like banking Trojans to malware used in targeted attacks on enterprises like remote administration tools (RATs). Therefore, this problem affects private parties as well as office or even government computers.

Attacker Model Before discussing HBCIAs in a malware context, we introduce the attacker model that we assume throughout the remainder of the paper. We assume that a malicious binary – creating a process $\mathcal{P}_{attacker}$ and targeting at least one process $\mathcal{P}_{victim}$ on the local system – already resides on the victim machine. We do not assume a specific way how this binary has been transferred to the machine. Possible ways are for example a drive-by-download, a download by the user due to social engineering or the use of an infected removable medium. Furthermore, we do not assume a specific way how or by whom this binary is executed. Possible ways are for example execution by the user due to social engineering or the execution by shellcode. Finally, we do not assume a specific privilege level of the entity that executes the malicious binary. The success of HBCIAs depends of course on the privilege level of $\mathcal{P}_{attacker}$.

HBCIA in a Malware Context Malware uses HBCIAs due to various reasons. Firstly, when malware executes its code in $\mathcal{P}_{victim}$ – which shelters a benign program – it can possibly avoid detection by anti-virus software. Secondly, malware might bypass personal firewalls by using HBCIAs. Thirdly, malware can gather critical information handled by $\mathcal{P}_{victim}$.

Since Microsoft Windows is still the platform most targeted by malware, we consider Microsoft Windows as a running example in the following. Our approach is not limited to this platform, though (cf. section 4.4). There exist many ways of achieving HBCIAs on Microsoft Windows. For example, malware uses functionality provided by Microsoft Windows APIs for debugging and interprocess communication or even functionality provided in the kernel space for its HBCIAs.

Family Feature Host-Based Code Injection Attacks Using HBCIAs comes with a lot of advantages from a malware author's point of view like access to unencrypted critical information. However, there is one architectural weakness:

once HBCIA is implemented, it is an integral component of the malware. Such an implementation decision influences a great deal of the malware's code base, e.g. the synchronization between infected processes.

Therefore, it is very unlikely that a malware author changes its malware's injection method or even completely removes the HBCIA feature. Furthermore, the implementation decision of using HBCIAs is usually taken at the very beginning of the malware's implementation process, once given the objectives that it should accomplish. This especially holds for malware that is derived from other malware families, e.g. by code reusage as in the case of several successors of the banking Trojan Zbot.

Given those considerations, we claim the following working hypothesis

Hypothesis 1. *The HBCIA is an inherent malware family feature, i.e. a malware author does neither remove this feature nor does he change the underlying injection method over time.*

2.3 Family Feature Investigation

In the following, we corroborate Hypothesis 1 with an investigation over time of eight code injecting malware families. At first we present the considered dataset. Then we explain the realisation of the study. Finally we describe our observations and results of the investigation.

Description of the Dataset Our dataset consists of eight code injecting malware families. We have gathered several versions as well as several variants of each version. The exact numbers are given in Table 1. Even though we are dealing with an incredible flood of malware samples each day, the number of malware families is actually by several orders of magnitude smaller[3]. Tables 1 summarizes the malware families included in the dataset. In total we considered 32514 samples of eight malware families. Even though Citadel comprises the lion's share of the data set, this does not affect the results since we examine each family separately. For all the considered malware families, we list the number of samples, the number of versions as well as the time span that lies between the first and the last version. The time span has been determined with the help of VirusTotal (first time seen)[4], except in the cases of Bebloh and Citadel where in-house unpackers exist that enable us to read the timestamp of the original PE file. Based on this information, Figure 1 shows the distribution of the considered samples over time.

Realisation of the Investigation We manually inspected a couple of samples of each family in order to understand how it employs its HBCIAs. As a result, we were able to extract a characteristic API call sequence for each malware family. Then we implemented a Cuckoo sandbox[5] behaviour analysis processing module and ran each member of the family in this sandbox. We used a Windows XP SP3 32 bit virtual machine in this investigation (cf. section 4.2). The

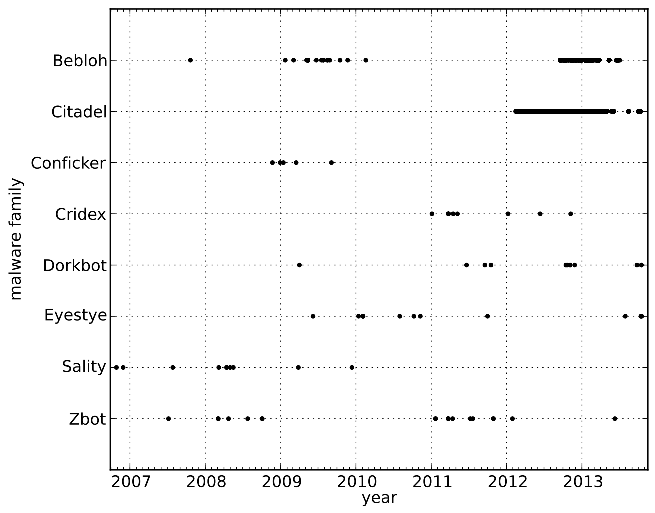

Fig. 1. Distribution of the considered samples over time

behaviour processing module processed the recorded API calls looking for the characteristic API call sequence.

Results of our Investigation The result of our observation backs our hypothesis. All eight families did not remove or change their injection behaviour over time. Thus, HBCIA can be considered as an elementary feature of malware families. Especially, it is invariant over different versions and variants of a malware family.

The vast majority of the samples – Bebloh, Citadel, Cridex, Dorkbot and Zbot – used WriteProcessMemory/CreateRemoteThread for injecting code into their target processes. Conficker uses a two-stage injection process. Firstly, it creates a thread in the victim process that loads Conficker as a library. Then it triggers the execution of the libraries' main function from the attacking process. Sality uses a message hook in order to load a dynamic linked library into other processes. Eyestye is able to inject code into foreign processes via either ZwWriteVirtualMemory/CreateRemoteThread or during child process creation by hooking NtResumeThread.

Another interesting observation is that Citadel's HBCIA code is identical to the code of its predecessor Zbot. We verified this by creating a binary diff of a Zbot variant and a Citadel variant. The intuition here is that malware authors rather build on leaked source code than fundamentally change it due to, for example, lack of time or missing deep knowledge of the original code base.

Table 1. Summary of the dataset for the family feature investigation

malware family	**considered samples**	**versions**	**date of first/last sample**
Bebloh	701	63	2007-10-21/2013-07-02
Citadel	31713	18	2012-02-14/2013-10-10
Conficker	5	5	2008-11-22/2009-10-31
Cridex	12	4	2011-01-04/2012-11-07
Dorkbot	21	7	2009-04-01/2013-10-16
Eyestye	12	4	2009-06-06/2013-10-17
Sality	20	6	2006-10-27/2013-07-02
Zbot	30	10	2007-07-07/2013-06-09
Total	**32514**	**117**	**2006-10-27/2013-10-17**

3 Bee Master

There exist several ways how a HBCIA can be accomplished. This includes local exploitation or functionality provided by the underlying OS. However, $\mathcal{P}_{attacker}$ must somehow insert code into its victim process $\mathcal{P}_{victim}$ and this code must be visible to the OS in order to run. This forms a paradox known as the *Rootkit Paradox*[6]. Hence, this hidden code can be detected.

Our approach – called *Bee Master* – for detecting host-based code injection attacks transfers the honeypot paradigm to OS processes. In short, we create processes and observe them for signs of attacks. Since we previously know the behaviour of those observed processes, any behaviour that deviates – such as new memory pages or new threads – from our expectations is considered suspicious. With it, we are able to detect HBCIAs without the knowledge of any special OS API – e.g. Microsoft Windows debugging API – by only relying on concepts – for example processes, threads or memory pages – common to almost all current multi-tasking operating systems.

Figure 2 depicts the architecture of *Bee Master*. The *Queen Bee* checks processes for signs of HBCIAs. These processes spawned by the *Queen Bee* are called *Worker Bees*. To ensure full knowledge of the *Worker Bee's* internals for the *Queen Bee*, the *Worker Bees* are created as child processes of the *Queen Bee*.

Due to the fact that the *Queen Bee* can totally observe its *Worker Bees*, it can detect HBCIAs within them. This is represented in Figure 2 by either a hazardous symbol or a green circle, signifying code has been injected or not, respectively.

The underlying assumption is that malware chooses its victim process either by resolving a process name to a process space or via a shotgun approach, meaning blindly injecting code in every accessible process space. In order to detect both kinds of approaches, the *Queen Bee* can deploy *Worker Bees* with random and configurable names. By the latter the *Queen Bee* may trick a malware into injecting in the *Worker Bee* despite checking for its process name. To our knowledge, there exist no malware family that verifies the genuineness of its victim processes.

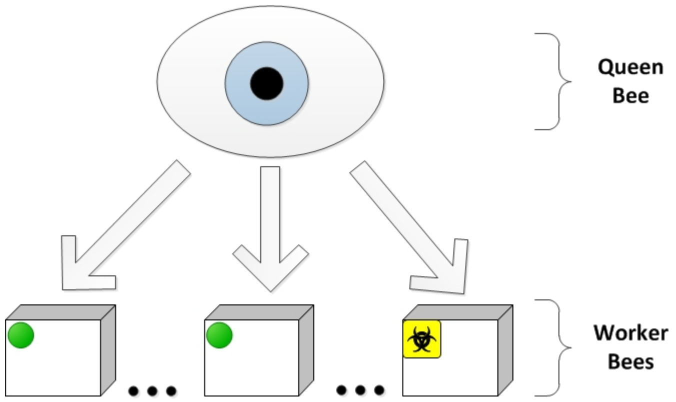

Fig. 2. Overview of the approach′s architecture: the *Queen Bee* and its *Worker Bees*

The following sections describe the *Queen Bee* and its *Worker Bees* in detail. Finally, we discuss limitations of our approach.

3.1 Queen Bee

The *Queen Bee* is the main component of our approach. It creates and handles *Worker Bees*, aggregates information from all of them and detects HBCIAs within them. Each *Worker Bee* is intended to pose as a $\mathcal{P}_{victim}$. Once the *Queen Bee* has detected a HBCIA, it creates a memory dump of the attacked *Worker Bee* for further analysis and shuts down the attacked *Worker Bee*. Note that in a real-world scenario the user should be warned and appropriate countermeasures should be taken.

Figure 3 sketches how the *Queen Bee* handles one of its *Worker Bees*. Firstly, the *Queen Bee* starts a *Worker Bee*. Note that this process creation depends on the privilege level of the *Queen Bee*: in user space the *Queen Bee* relies on the underlying operating system's API, in kernel space or as a virtual machine introspection component it could directly create those processes by manipulating kernel data structures. Subsequently this newly created *Worker Bee* is monitored by the *Queen Bee*.

This monitoring is split in three steps: gathering information on the *Worker Bee's* state, analysing this information and deciding whether or not a suspicious change occurred within the *Worker Bee*. In the first step the *Queen Bee* gathers information on the state of the *Worker Bee*. Two requirements have to be met for a successful HBCIA: the planting of additional code in a victim process and afterwards the execution of this code. Therefore, the *Queen Bee* gathers information on loaded libraries, memory pages as well as executed threads. This information comprises the two components that are needed. The source of the information depends on the actual implementation. In a user space implementation it has to rely on information provided by the OS, for example through

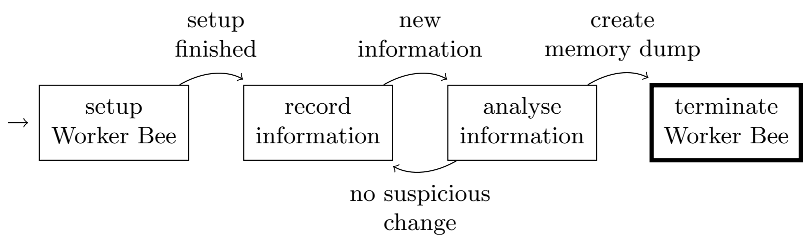

Fig. 3. Control flow of the *Queen Bee′s Worker Bee* handling algorithm

system calls. In an implementation as a virtual machine introspection component it could parse several sources including the kernel's internal data structures.

Once the state of the *Worker Bee* has been obtained, the *Queen Bee* analyses this information. It compares this information with the assumed behaviour of the *Worker Bee.* Since every *Worker Bee's* behaviour is previously known, any change within a *Worker Bee* is highly suspicious. As soon as the *Queen Bee* detects a suspicious change, it creates a memory dump of the *Worker Bee* for further analysis. Finally the *Queen Bee* terminates the *Worker Bee.*

The *Queen Bee* can be either implemented as a user space program, a kernel module or even as a virtual machine introspection component. In the last case the *Queen Bee* is executed with higher privileges than any malware executed inside of the virtual machine. We recommend to implement the *Queen Bee* with the highest privilege level possible to ensure its integrity.

3.2 Worker Bees

Worker Bees are the second component of our approach. Each *Worker Bee* is a common process created by the *Queen Bee* and it serves the *Queen Bee* as a sensor. There can be one or more *Worker Bees* acting as a possible victim process. Thus, the user can model multiple processes – e.g. by using different process names – that pose as a possible target for an attacker.

The behaviour of each *Worker Bee* is passive. It is just waiting for being compromised. For it, a *Worker Bee* can be configured up front. Configurable parameters of a *Worker Bee* are parameters that are common to almost every current multi-tasking OS. In this way it is possible to imitate real processes, e.g. a web browser. Therefore, a malware is tricked into believing that it is targeting the alleged process. At the moment configurable parameters include the number of threads, memory mapped files, the list of loaded libraries, the process name, the process window name, in case it is executed in a graphical environment, and the command line string of the process.

3.3 Limitations

We discuss limitations of *Bee Master* in this section.

Missing Attacks The success of detecting HBCIAs depends on the process identification feature used by the malware. Currently, this can be, for example, the process name, the process window name or loaded libraries. Since it is not feasible to provide a process for every possible process identification feature combination, it is possible that attacks are missed. Note that network honeypots suffer from a similar problem: presenting the right network service on the right port in the right version. Furthermore, note that in many cases no process identification takes place at all and malware injects code into every accessible process space.

Detection of Process Hollowing *Bee Master* cannot detect process hollowing. While injected code is usually executed in parallel with the original code of the process space, in process hollowing the injected code replaces the original code and the process just executes the injected code[7]. For it, the attacker has to have full control over the victim process. Therefore, the victim process is usually created by the attacker. Hence, our approach is not capable of detecting such HBCIAs. This stems from the fact that processes which are not created by the *Queen Bee* cannot be controlled by it.

4 Evaluation

In this section we evaluate a prototype implementation of *Bee Master*. While most of the evaluation focuses on Microsoft Windows – due to the fact that it is still the prevalent target for malware –, we show in a case study with a Linux banking Trojan that our approach is not limited to solely one operating system.

First off, we explain the prototype implementation and configuration of *Bee Master* used throughout the evaluation. Then we describe the evaluation environment. Subsequently we proceed to evaluate *Bee Master*'s ability to detect HBCIA in a quantitative evaluation. In this evaluation we also show that our approach can handle a broad variety of prevalent malware families. This quantitative evaluation is followed by two detailed case studies in order to show the capturing process in detail as well as the OS-agnosticism of our approach. At the end of this section we conduct a performance evaluation of our prototype implementation.

4.1 Implementation and Configuration of the Prototype

This section describes briefly how the prototype of *Bee Master* was implemented and how it was configured for the evaluation.

Implementation We have implemented a prototype of *Bee Master* for Microsoft Windows as well as Ubuntu Linux. Our prototype implementation is split into two layers: an OS abstraction layer and a logic layer. The OS abstraction layer helps abstracting from the underlying OS and allows a quick portability to other operating systems. Based on this layer the logic layer implements all OS independent functionality. The *Queen Bee* and its *Worker Bees* are both implemented as user mode programs. The *Queen Bee* uses the Windows Debugging API on Microsoft Windows and procfs on Ubuntu Linux for continuously checking on its *Worker Bees*.

Of course malware can detect if a process is being debugged. This and the fact that the prototype is implemented as a user mode program are two shortcomings of the prototype. Note that these shortcomings do not apply to the underlying approach in general. Possible solutions for these drawbacks are discussed in section 6.

Configuration We ran *Bee Master* with the default configuration. There is one configuration file for each OS *Bee Master* is executed on. These configuration files were compiled based on our experience with HBCIA-employing malware. On Microsoft Windows the configuration file comprises five victim processes: the Windows shell (explorer.exe), the default Microsoft browser (iexplore.exe), a popular browser (firefox.exe), a service (svchost.exe) and a random process (pdtyzgxm.exe). The first four processes are known to be frequently attacked. The latter one is chosen in order to discover HBCIA malware families that employ a shotgun approach. On Linux the configuration file just compromises two victim processes: a popular browser (firefox) and a random process (pdtyzgxm). These two victim processes were chosen for the same reasons as above.

4.2 Description of the Evaluation Environments

We used VirtualBox 4.2.10 as a virtualization environment throughout the evaluation. Three different Windows versions – namely Windows XP SP3 32 bit, Windows 7 SP1 32 bit and Windows 8 SP0 32 bit – and one Linux distribution – Ubuntu 13.04 64 bit – were used. The Windows VMs are 32 bit systems, because in our experience the majority of malware families focuses on this architecture. The Linux VM is a 64 bit system because the considered malware family requires such a system in order to execute. Each VM has one GB of RAM and one core of a Intel Core i7-2760QM CPU running at 2.40 GHz. All VMs have been installed without additional software packages. We have hardened all VMs against several VM detection methods in order to cope with evasive malware.

4.3 Quantitative Evaluation

We have evaluated *Bee Master* in quantitative evaluations on Windows XP, Windows 7 and Windows 8. At first, we have evaluated it on a set of malware families known to employ HBCIAs. This is followed by an evaluation on benign programs in order to estimate potential false positives.

Description of the Datasets We have compiled two datasets for the quantitative evaluation: one dataset consists of malware families known to employ HBCIAs and one consists of goodware.

The dataset for the known malware family evaluation compromises representatives of 38 malware families. Again, we would like to point out that HBCIAs are a family feature (cf. Hypothesis 1 in section 2.2) and therefore it is sufficient to pick one representative for each malware family. The malware dataset also includes those four families that were responsible for 32,1% of all new infection reports in 2012[1]. In addition we added 34 prevalent malware families such as Carberp, Hesperbot or Vawtrak. We host a full list of all malware families used in this paper on our server[8]. We have manually verified in all 38 cases, that the representative employs HBCIAs. As stated in section 3.3, process hollowing cannot be detected with our approach. Therefore, we did not consider any malware family that uses this technique.

Unfortunately, malware as any other software is not compatible with every OS. While we have been able to successfully execute each sample of the dataset on Windows XP, we were not able to execute samples from some malware families on Windows 7 and Windows 8 due to incompatibilities. In the case of Windows 7 no representative of the Poison family executed. In the case of Windows 8 we could not find a working representative for the following families: Bamital, Conficker, Gamker, Ice X, Poison and Sykipot. Therefore, the dataset for Windows 7/Windows 8 were reduced to 37 and 32 families, respectively.

The dataset for the false positive estimation consists of goodware ranging from system tools to office software. The goodware has been obtained from two sources. Firstly, we have gathered Microsoft Windows system tools originating from Windows' system paths (321 for Windows XP, 440 for Windows 7, 470 for Windows 8). Secondly, we have chosen 13 very common programs such as web browsers, instant messaging clients or encryption software. In total this sums up to 334/453/483 known goodware programs for Windows XP/Windows 7/Windows 8.

Realisation of the Evaluation We have conducted this evaluation as described in the following. At first we prepared a virtual machine (VM) with our prototype implementation already set up and running and took a snapshot of this original state. We configured the prototype as described in section 4.1. Then each representative was executed for five minutes in this VM. Afterwards, the logs and dumped files were extracted from the VM and the VM was reverted to its original state.

Malware Families In all cases we have been able to detect at least one HBCIA in one of the five processes by each malware family. Hence, we have detected the malicious behaviour in all cases on all three Windows operating system versions.

In Figure 4 the total observed injections per process are shown. Many of the considered malware families employ at least one injection into explorer.exe. On Windows XP 34 families (89%) show this behaviour. Whereas we can observe

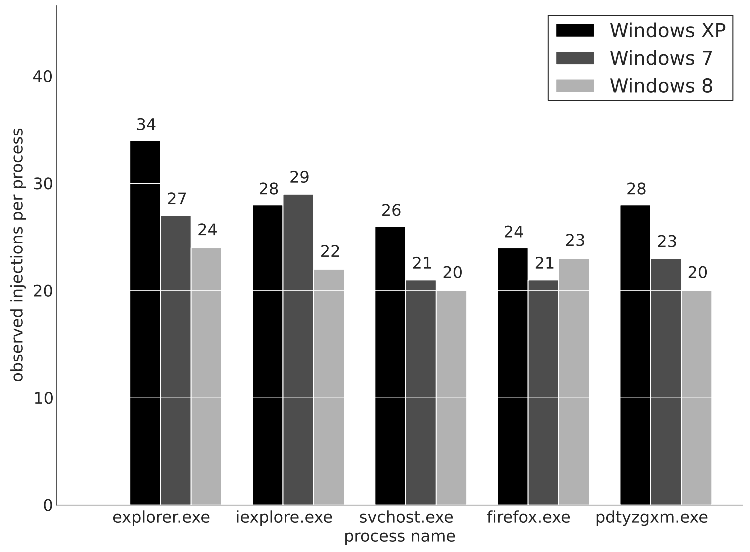

Fig. 4. Observed injections on Windows XP, Windows 7 and Windows 8

few injections in firefox.exe. Intuitively this process is either attacked by banking Trojans or malware families that employ a shotgun approach. Malware families that target an exclusive set of processes are more likely to select those targeted processes from processes that are already installed and running by default on the OS. Another interesting fact is that a significant quantity attacks the random process (24 families [63%]/21 families [56%]/23 families [69%] on Windows XP/7/8).

Figure 5 shows the count of targeted processes per malware family on Microsoft Windows XP. Two thirds of the malware families target at least four or all *Worker Bees*. This includes especially information stealing malware families such as Cridex, Hesperbot or Zeus. In particular, there is a considerable amount of malware families that attack all *Worker Bees*. Again this implies that many malware families use a shotgun approach. Further, there is a large share of families attacking four *Worker Bees*.

Interestingly, a lot of those families attack the random *Worker Bee* but skip one of the other processes. Most probably, some malware families have implemented a blacklist feature in order to exclude specific processes. One third of the considered families target one, two or three *Worker Bees*. The sample set incorporates a wide range of malware types such as RATs (Poison), network worms (Conficker) but also banking Trojans (Tinba). As all selected samples utilize HBCIAs, it can be considered a reliable indicator of compromise (IOC). Above all, we would be able to detect all families of our dataset with just two *Worker Bees*, because all malware families target at least either explorer.exe or iexplore.exe.

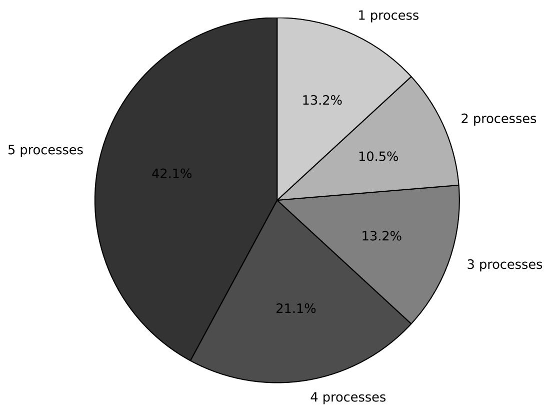

Fig. 5. Count of targeted processes per sample on Windows XP

Goodware After conducting experiments with malware, we determine the false positive rate of our detector. The setup for this experiment is in line with the setup for the known malware families experiment. Our system could not detect any sign for HBCIAs during any of the 334 executions on Windows XP, 453 executions on Windows 7 and 483 executions on Windows 8.

Discussion In the quantitative evaluation we have shown that our approach can cope with prevalent malware families and it detects each malware family in the dataset. As expected, the explorer.exe process is the one targeted by most families. A majority of the considered families attacks all five processes including the random process. This suggests that the shotgun approach is widely spread. Furthermore, many families attack four processes including the random process which suggest that there exists blacklisting employed by HBCIA malware. A key observation is that with only two *Worker Bees* – explorer.exe and iexplore.exe – it is possible to cover 100% of our dataset. Furthermore, we have shown for a diverse set of goodware, ranging from system tools to office programs, that our detector has a false positive rate of 0%.

4.4 Case Studies

We examine two malware families in the case studies. Each case study details a HBCIA on a different operating system. At first we look at Hanthie, a banking Trojan for Linux. Then we cover Poison, a RAT for Microsoft Windows.

Hanthie Hanthie is the first Linux banking Trojan that has been seen in the wild[9]. It gained a lot of attention in August 2013. This banking Trojan is capable of form-grabbing in a handful of browser like Firefox.

Therefore, it injects a shared object into all processes except the ones that match some predefined substrings like *dbus*. In order to load a shared object into a foreign process space, the injecting process has to attach to the targeted process. This is achieved with the help of a system call (ptrace) that allows the manipulation of processes on Linux. Once the injecting process has attached to its victim process, it tries to determine the address of a function (dlopen) that is part of the interface to the dynamic linking loader on Linux. With it, it is possible to load shared libraries during runtime. The injecting process uses this function in order to let the victim process load such a shared library. Once the shared library has been loaded by the dynamic linking loader, its initialisation function is executed (_init).

In this case study we used Ubuntu Linux as evaluation environment. Therefore, we booted the Ubuntu 13.04 VM and started *Bee Master* with the default configuration for Linux (cf. section 4.1). Afterwards, we executed Hanthie. Once executed, Hanthie installed itself and started its injection mechanism. Our prototype detected two new threads and new modules within its two *Worker Bees*. Hence, it dumped the new modules for further analysis. Manual analysis revealed that the linux-based prototype had successfully captured Hanthie's injected shared library.

Poison Poison is a RAT consisting of a server component and a client component. The server component has to be installed on the victim′s machine and can be remotely administrated with the help of the client component. It is publicly distributed by its author[10]. This RAT emerged in 2006 and the last publicly available version dates back to 2008.

While malware families such as Zbot or Conficker inject their code into their victim process as a whole, Poison injects its position-independent code function by function to several memory regions. The main reason for this behaviour is that it allows flexibility because only needed parts of the code have to be deployed. This also implies that the analysis is more complex compared to other injecting malware families such as Zbot or Conficker. Because the reverse engineer has to dump not only one memory region but several regions.

We conducted this case study on Windows XP SP3. We started the *Queen Bee* with the default configuration for Microsoft Windows (cf. section 4.1). Once the *Queen Bee* and its *Worker Bee* have been started, we started Poison. The *Queen Bee* immediately detected 19 new memory regions and one new thread within one of its *Worker Bees*, namely iexplore.exe. Hence, it created a memory dump of it. We verified the successful attack by manually inspecting the created memory dump.

Discussion We have evaluated our approach's prototype in two detailed case studies on two different types of malware (a banking Trojan and a RAT) as well as on two different operating systems (Linux and Microsoft Windows). *Bee Master* detects the HBCIAs in both case studies. Furthermore, it delivers a memory dump and many valuable pointers towards the intrusion technique used.

This qualitative evaluation shows in detail that *Bee Master* is not limited to the type of the underlying operating system and that it can be easily ported to possible platforms prone to HBCIAs.

In addition to the above, it has to be noted that none of the considered malware families check the genuineness of their victim process before the actual injection. This clearly shows that current malware families are prone to detection at this stage of their execution.

4.5 Performance Evaluation

After evaluating the functionality of our prototype, we focus on its performance on Windows XP SP3 32 bit (cf. section 4.2) in this section.

For it, we have evaluated the CPU usage of the prototype with a different number of *Worker Bees*. The considered number of *Worker Bees* were {1,3,5,7}. The measured time period was 300 seconds. No other programs were running on the system during the measurements. The CPU usage was captured with the help of *Performance Counters* provided by Microsoft Windows.

Figure 6 shows the results of the performance evaluation. The first observation is that the more *Worker Bees* need to be handled, the more CPU usage is needed. But as one can see in section 4.3, only a limited set of *Worker Bees* is needed in order to detect a large set of prevalent malware families.

The second observation is the pattern of the graphs. Our prototype checks on all its *Worker Bees* every two seconds. Therefore, the graphs show spikes every two seconds.

Since this parameter is configurable, one can tweak it to his needs. From a pragmatic point of view, we believe that the choice of two seconds in combination with a small set of *Worker Bees* is an acceptable one. Without occupying to many CPU cycles in such a scenario, we are able to instantaneously detect HBCIAs.

5 Related Work

We split the discussion of related work in detecting changes in the process behaviour in general, detecting HBCIAs and honeypots.

Detecting Changes in Process Behaviour Forrest et al. [11] propose a method for detecting anomalies in Unix processes. They record sequences of system calls and use them to build process specific signatures beforehand. Then they apply these signatures on-line in order to detect anomalies in the system. Warrender et al. present further data models for anomaly detection based on system calls[12].

Wagner et al. propose an approach for detecting anomalies in the program behaviour by applying a static analysis to each program that should run on a system[13]. Thereby, they model a transition system that is capable of detecting anomalies in system call traces.

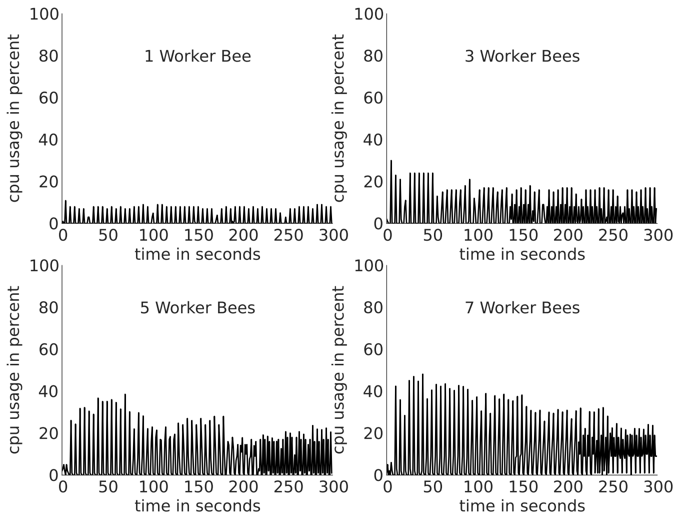

Fig. 6. System load in relation to running *Worker Bees*

While these approaches are more general than *Bee Master*, they fail to detect an attack if the malware mimics the original application. *Bee Master* is not vulnerable to mimicry attacks since it does not depend on system call tracing.

Detecting Host-Based Code Injection Attacks While there has been a lot work on thwarting code injection attacks (e.g. [14] or [15]), the research community has not focused intensively on detecting (host-based) code injection attacks.

Sun et al. [16] propose a system for detecting HBCIAs by hooking certain system calls associated with this behaviour. The hooking is performed in kernel mode. Since the approach relies on certain system calls it depends on low-level OS details. Furthermore, the system by Sun et al. is not capable of detecting unknown code injection attacks, because it only hooks system calls known to be related to code injection attacks.

White et al. [17] describe an approach for detecting the provenance of malicious code in memory dumps of Microsoft Windows operating systems. They achieve this by hashing memory pages and compare the hashes to a previously built hash database. Thereby they can reduce the amount of memory pages that has to be analysed manually. The memory forensic framework Volatility[18] comes with a plug-in called Malfind for detecting HBCIAs in memory dumps. Malfind detects host-based code injection attacks based on several low-level characteristics of Microsoft Windows. Those characteristics include Virtual Address Descriptors and PE file format characteristics. Both approaches focus on forensic analysis. Thus, they are not suited for real-time analysis. Furthermore, both approaches rely on low-level details of Microsoft Windows and cannot be easily ported to another OS.

Hanel [19] presents a tool for detecting HBCIAs in Windows processes. This is achieved by scanning each process for a handful of low-level characteristics similar to Volatility. Furthermore, this tool can spawn an instance of the Internet Explorer and scan it for those aforementioned characteristics. While this tool focuses on real-time analysis, it suffers from relying on low-level details, non-portability as well as not being extensible in order to detect a larger set of malware families.

To the best of our knowledge, there exists no related approach that is capable of detecting host-based code injection attacks OS-independently as well as detecting previously unknown host-based code injection attacks during runtime.

Honeypots Honeypots have been intensively researched during the last years. But the majority of honeypot research focuses on network attacks. This includes honeypots that are waiting to be exploited (server honeypots) like [20] and honeypots that are actively trying to be exploited (client honeypots) like [21]. *Bee Master* does not focus on network-based attacks, but rather on attacks on local processes. Nevertheless, those attacks can be part of a larger attack chain, originating in one of todays common malware spreading techniques such as drive-by downloads or social engineering.

Poeplau et al. [22] present a honeypot that is able to emulate removable USB-devices. Therefore, they target malware that spreads via removable media. Their work can be considered the most related work to our approach. However, they focus on a different malware family feature. While *Bee Master*'s scope is a persistence feature, they focus on a spreading feature. By that they are able to detect a different class of malware. Therefore, a comparison between the two approaches is difficult.

Even though *Bee Master* applies the honeypot paradigm to OS processes, we do not consider it as a honeypot but rather as a detector.

6 Conclusion and Outlook

In this paper we have introduced a novel approach – called *Bee Master* – to detect host-based code injection attacks. At first we have shown in a study with eight malware families that such attacks are a family feature, i.e. the injection technique does not change between variants and versions. Then we have presented *Bee Master*, a novel approach for detecting such attacks. This is achieved by transferring the paradigm of honeypots to OS processes. *Bee Master* consists of two components: the *Queen Bee* and its *Worker Bees*. The *Queen Bee* continuously checks on all its *Worker Bees*. Therefore, it detects suspicious behaviour within a *Worker Bee*. In such a case, the *Queen Bee* creates a memory dump of the attacked *Worker Bee* for further analysis and terminates it.

Bee Master does not rely on special hardware or modifications of the underlying OS. Since *Bee Master* does not rely on an OS or any special API, it can be deployed on a wide range of operating systems. Further, *Bee Master* only assumes concepts – such as processes, threads or libraries – common to almost all current multi-tasking operating systems.

We have implemented *Bee Master* for Microsoft Windows as well as Ubuntu Linux. The evaluation results show that *Bee Master* can detect HBCIAs with high detection rates and no false positives by only relying on concepts – such as threads or memory pages – common to almost every current multitasking operating system. Furthermore, we have shown that current malware is very vulnerable during its HBCIA-stage and that it can be easily detected at this stage since it does not check its victim process for genuineness.

Future work focuses on the limitations of our current implementation. The *Queen Bee* will be reimplemented on a higher level of privileges to counter the current limitations of our implementation. This will improve the tamper resistance. Furthermore, we will focus on improving the overall performance making our approach even more appealing as a complementary security measure to traditional anti-virus software.

Acknowledgements We would like to thank Niklas Bergmann for the help he provided during the implementation of the prototype, Daniel Plohmann for the data he provided as well as the anonymous reviewers of this paper for discussions and comments.

The final publication is available at *Springer* via `https://doi.org/10.1007/978-3-319-08509-8_13`.

References

1. Symantec. Internet Security Threat Report 2013, Volume 18. Technical report, 2013.
2. N. Percoco. Global Security Report 2013. Technical report, Trustwave, 2013.
3. M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007.
4. VirusTotal. `https://www.virustotal.com`, Last access: August 21, 2017.
5. Cuckoo Sandbox. `http://www.cuckoosandbox.org`, Last access: August 21, 2017.
6. J. Kornblum. Exploiting the Rootkit Paradox with Windows Memory Analysis. 2006.
7. M. Hale Ligh, S. Adair, B. Hartstein, and M. Richard. *Malware Analyst's Cookbook and DVD: Tools And Techniques For Fighting Malicious Code*. Wiley Publishing, Inc., 1 edition, 2011.
8. T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla. List of malicious samples used in bee master: Detecting host-based code injection attacks. `http://net.cs.uni-bonn.de/wg/cs/staff/thomas-barabosch/`, Last access: August 21, 2017.

9. L. Kessem. Thieves Reaching for Linux – "Hand of Thief" Trojan Targets Linux. `https://blogs.rsa.com/thieves-reaching-for-linux-hand-of-thief-trojan-targets-linux-inth3wild`, August 2013 Last access: August 21, 2017.
10. Mandiant. APT1 - Exposing One of China's Cyber Espionage Units. Technical report, Mandiant, 2013.
11. S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *In Proceedings of the IEEE Symposium on Security and Privacy Proceeding*, pages 120–128. IEEE, 1996.
12. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145. IEEE, 1999.
13. D. Wagner and D. Dean. Intrusion detection via static analysis. In *In Proceedings of the IEEE Symposium on Security and Privacy, S&P 2001*, pages 156–168. IEEE, 2001.
14. G. Kc, A. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *In Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, New York, NY, USA, 2003. ACM.
15. A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. ASIST: Architectural Support for Instruction Set Randomization. *The Proceedings of the CCS13, November, 2013, Berlin, Germany*, 2013.
16. H. Sun, Y. Tseng, and Y. Lin. Detecting the Code Injection by Hooking System Calls in Windows Kernel Mode. In *In the Proceedings of the International Computer Symposium 2006*, 2006.
17. A. White, B. Schatz, and E. Foo. Integrity verification of user space code. *Digital Investigation*, 10, 2013. The Proceedings of the Thirteenth Annual {DFRWS} Conference 13th Annual Digital Forensics Research Conference.
18. Volatile Systems. The Volatility Framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`, Last access: August 21, 2017.
19. A. Hanel. injdmp. `http://hooked-on-mnemonics.blogspot.jp/p/injdmp.html`, 2013 Last access: August 21, 2017.
20. P. Baecher, M. Koetter, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *In the Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2006.
21. J. Nazario. PhoneyC: a virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'09, Berkeley, CA, USA, 2009. USENIX Association.
22. S. Poeplau and J. Gassen. A honeypot for arbitrary malware on USB storage devices. In *7th International Conference on Risk and Security of Internet and Systems (CRiSIS)*, 2012.