Host-Based Code Injection Attacks: A Popular Technique Used By Malware

Thomas Barabosch
Fraunhofer FKIE
Friedrich-Ebert-Allee 144
53113 Bonn, Germany
thomas.barabosch@fkie.fraunhofer

Elmar Gerhards-Padilla
Fraunhofer FKIE
Friedrich-Ebert-Allee 144
53113 Bonn, Germany
elmar.gerhards-padilla@fkie.fraunhofer.de

©2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Abstract

Common goals of malware authors are detection avoidance and gathering of critical information. There exist numerous techniques that help these actors to reach their goals. One especially popular technique is the Host-Based Code Injection Attack (HBCIA). According to our research 63.94% out of a malware set of 162850 samples use HBCIAs. The act of locally copying malicious code into a foreign process space and subsequently executing it is called a Host-Based Code Injection Attack.

In this paper, we define HBCIAs and introduce a taxonomy for HBCIA algorithms. We show that a HBCIA algorithm can be broken down into three steps. In total there are four classes of HBCIA algorithms. Then we examine a huge set of malware samples and estimate the prevalence of HBCIA-employing malware and their target process distribution. Moreover, we analyse Intrusion Prevention System data and show that HBCIA-employing malware prefers network-related processes for its network communication.

To the best of our knowledge, we are the first to thoroughly describe and formalize this phenomenon and give an estimation of its prevalence. Thus, we build a solid foundation for future work on this topic.

1. Introduction

Several reports have been published about malware families that operated for years without being detected (Uroburos, Careto or Stuxnet). Even though they have been implemented with different goals in mind, they share one common feature: they all inject code locally into foreign process spaces. One reason for this behaviour is detection avoidance. However, code injections are not limited to targeted malware. Mass-malware also uses code injections in order to stay under the radar (ZeroAccess, ZeusP2P or Conficker). Detection avoidance is not the only advantage of using code injections from a malware author's point of view. Further reasons for using code injections are interception of critical information, privilege escalation or manipulation of security products.

The above mentioned examples are all malware families for Microsoft Windows. However, code injections are platform-independent. In fact all established multitasking operating systems (OS) are prone to HBCIAs. Malware families such as Flashback (Apple Mac OS X) [16], Hanthie (Linux) [14] or Oldboot (Android) [7] employ HBCIAs on mobile and non-mobile operating systems. This fact shows that HBCIAs are present on mobile and non-mobile operating systems today. HBCIAs are therefore a relevant technique for security researchers.

In this paper, we examine the phenomenon of Host-Based Code Injection Attacks (HBCIA) employed by malware in-depth. We describe the motivation for malware authors to use HBCIAs. We define HBCIAs and propose a taxonomy for classifying them. Several evaluations on a large set of malware samples discover the prevalence of HBCIA-employing malware, typical target processes and network communicators.

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

(I) **Formalization of Host-Based Code Injection Attacks used by malware and its key components**
We derive definitions for key terms associated with Host-Based Code Injection Attacks. By formalizing HBCIAs, we build a solid foundation for future research on this topic.

(II) **Proposal of a Host-Based Code Injection Attack**

algorithm taxonomy
We examine the algorithms used for employing HBCIAs and derive a classification scheme for them. We show that there exist four different classes.

(III) **Prevalence estimation of Host-Based Code Injection Attacks used by current malware**
We estimate the prevalence of HBCIAs used by current malware based on a set of 162850 malware samples. We show in addition that not all processes are targeted equally and that such malware prefers a different set of processes for network communication.

2 Code Injections

Probably the first code-injecting malware was the Morris worm [8] in 1988. It was able to infect large parts of the Internet by remotely exploiting a buffer overflow.

We discuss Code Injections and more specifically Host-Based Code Injection Attacks employed by malware in this section. At first, the term Code Injection is defined. Then we define Host-Based and Remote Code Injections. Afterwards, we differentiate between Host-Based and Remote Code Injections and Host-Based and Remote Code Injection Attacks.

While Remote Code Injections Attacks have been intensively researched (e.g. [18] or [25]), there is little research on Host-Based Code Injection Attacks.

2.1 Code Injections

In this and the latter sections, we define the term Host-Based Code Injection Attack. We achieve this by developing a set of definitions beginning with simple Code Injections.

Firstly, we define a Code Injection as follows

Definition 1 *A Code Injection denominates copying of code from an injecting entity ϵ_{inject} into a victim entity ϵ_{victim} and executing this code within the scope of ϵ_{victim}.*

For example, these entities ϵ_{inject} and ϵ_{victim} can be hardware devices or operating system processes. But they are not limited to these examples. It is important to notice that there are two crucial things needed for a Code Injection: executable code and an execution context for this code.

2.2 Host-Based Code Injections versus Remote Code Injections

The definition of a Code Injection does not specify the place of residence of ϵ_{inject} and ϵ_{victim}. It can be distinguished between two cases. The attacker and the victim reside on the same system (Host-Based Code Injection) and the attacker and the victim reside on different systems (Remote Code Injection).

We define a Host-Based Code Injection as follows

Definition 2 *A Host-Based Code Injection (HBCI) is a Code Injection, where the two entities ϵ_{inject} and ϵ_{victim} reside on the same computer system.*

ϵ_{inject} injects code into ϵ_{victim} typically with the help of the operating system. In this scenario, ϵ_{inject} or ϵ_{victim} can be, for example, a user space process, a kernel module or a hardware device.

In contrary to a Host-Based Code Injection, ϵ_{inject} and ϵ_{victim} reside on two different systems in a Remote Code Injection. That leads to Definition 3 for a Remote Code Injection.

Definition 3 *A Remote Code Injection (RCI) is a Code Injection, where the two entities ϵ_{inject} and ϵ_{victim} do not reside on the same computer system. They communicate by means of a connecting channel.*

For example, such a connecting channel can be a computer network. The injection is typically triggered by exploiting a vulnerability in a network service. In such an scenario ϵ_{inject} would be a network client and ϵ_{victim} a network service. ϵ_{inject} sends a specially crafted payload containing code to exploit ϵ_{victim}. In case ϵ_{victim} is vulnerable to this exploit, the code is executed within the scope of ϵ_{victim}.

2.3 HBCI/RCI vs. HBCIA/RCIA

Host-Based and Remote Code Injections are not malicious per se. There are legitimate uses for injecting code. These legitimate uses include hot patching [12], software diagnostics [9], malware analysis [22] and debugging [19]. The Microsoft patent "Method for injecting code into another process" also suggests benign use cases, because their "invention relates generally to computer software diagnostic tools" [9]. In general, injecting code is seldom a feature that is needed by a common program in order to fulfil its task. It is rather needed during application development. However, there is no way to distinguish between Host-Based/Remote Code Injections and corresponding attack versions without taking the purpose of the injection into account.

Thus, we define a Host-Based Code Injection Attack/Remote Code Injection Attack as follows.

Definition 4 *If a Host-based Code Injection or a Remote Code Injection serves a nefarious purpose, i.e. it has not been intended by the original author of ϵ_{victim}, then it is called a Host-Based Code Injection Attack (HBCIA) or Remote Code Injection Attack (RCIA), respectively.*

3 HBCIAs from a Malware Author's Point of View

Host-Based Code Injection Attacks are an important technique for the successful operation of several malware families such as Citadel [21], Flame [2] or Flashback [16].

On the one hand, using HBCIAs comes with a lot of advantages from a malware author's point of view such as privilege escalation or detection avoidance. On the other hand, a malware author has to meet additional challenges when implementing HBCIA-employing malware such as maintaining system stability or handling increased architectural complexity. This section discusses these advantages and disadvantages.

3.1 Advantages of Employing HBCIAs

Using HBCIAs is beneficial for a malware author. HBCIAs allow malware to intercept critical information, escalate privileges, avoid detection and manipulate security products.

3.1.1 Interception of Critical Information

Once malware has injected itself into a foreign process space, it can access all information that this process space holds. There is no access restriction within a process. Malware can therefore read or write data, but also code. This enables malware to intercept critical information. Even if the information is encrypted before transmission to a communication partner.

For example, malware can hook API functions of a browser or it can scarpe the process space for valuable data. The first behaviour can be seen in banking Trojans like Citadel [21]. The latter behaviour is a feature shown by Point-of-Sale (POS) malware like Dexter.

The interception of critical information is typically done in user mode [6]. Since data that is sent over the network is usually encrypted with the help of user mode libraries. This information cannot be intercepted in clear text in kernel mode.

3.1.2 Privilege Escalation

HBCIAs can also be used for escalating privileges. Malware can gain the same access rights as the foreign process space by injecting its code into it. This might enable malware to access files or bypass process-based local firewall rules. One example is the POS malware Dexter. It injects its code into Microsoft Windows' Internet Explorer in order to circumvent local firewall policies [11].

3.1.3 Detection Avoidance

Another reason for using HBCIAs is avoiding detection. If a user is suspicious, he might investigate the currently running processes. In case he finds a suspicious process name, he might kill this process. On account of this the malware stops operating and the actor in-behind loses a valuable resource. But also automatic detection by security products might be evaded. Once malware resides in a victim process space, it blends into the behaviour of its victim. Hiding within another process space might enable malware to continue its operation for an even longer period of time. One example is the cyber espionage malware Flame. It uses HBCIAs in order to avoid its detection [2].

3.1.4 Manipulation of Security Products

Another application of HBCIAs is the manipulation of security products. Since a security product's goal is detecting and removing malware, it is natural that malware employs self-defence. Its objective is making security products unresponsive or even removing them completely in order to survive.

The malware injects itself into processes of security products. Once injected, several options exist in order to manipulate a victim process. One option is terminating the process from within. Other processes would suppose that this behaviour has been intended by the victim process. But malware can also alter the code of the victim process. Malware can achieve this via hooks, but also via overwriting critical code section with no operation instructions (NOPs). One example is the ZeroAccess rootkit. It uses HBCIAs in order to terminate security products [10].

3.2 Disadvantages of Employing HBCIAs

Even though employing HBCIAs comes with a lot of advantages from a malware author's point of view, there exist at least two disadvantages, when compared with non HBCIA-employing malware. These disadvantages are an increased architectural complexity as well as the risk of system instability.

3.2.1 Architectural Complexity

One clear disadvantage of using HBCIAs is the increased architectural complexity of the malware. The implementation costs increase from a malware author's point of view. Due to the fact that instead of implementing a self-contained – i.e. single process software – they have to implement a parasitic component for other process spaces.

In some cases malware authors even have to implement a distributed system. The nodes of such a distributed system are the processes that are currently running. They have to

deal additionally with characteristics of distributed systems such as timing issues, failure tolerance, message passing or synchronization. Furthermore, the testing and debugging increases in complexity. Not only the malware analyst has to grapple with such a piece of code, but also its developer.

3.2.2 Risk of System Instability

Injecting code into foreign process spaces is risky. Every bug in the injected code could crash the victim process. This could lead to system instability. This holds especially true if code has been injected into critical system processes. The probability increases that the user gets suspicious and that he starts investigations. These investigations might lead to a removal of the malware and to a loss of a valuable resource. Hence, the malware author has to work very carefully during the development of a HBCIA-employing malware. He has not only to cope with the behaviour of its own code's threads, but also with the behaviour of the victim process' threads. Furthermore, he has to ensure that these threads do not interfere with each other.

4 Taking a Closer Look at HBCIA Algorithms

This section takes a closer look at the algorithms that underlie HBCIAs. We assume in the following sections that ϵ_{victim} is a process. Since today's malware targets exclusively processes.

The basic idea of a HBCIA algorithm can be sketched by the simplified HBCIA algorithm in Figure 1. At first the attacking entity has to select a victim process (step one). Once it has found a victim process, the attacking entity copies the to be executed code into the victim process (step two). Then it executes the injected code within its victim process (step three). This algorithm repeats for other processes or it terminates. Each step can result in an error. However, we have omitted possible failure states for clarity reasons in this figure.

The following sections examine each of the sketched steps. The last section closes with the proposal of a HBCIA algorithm taxonomy.

4.1 Victim Process Selection

The attacker entity ϵ_{inject} has to select a victim entity ϵ_{victim} before the actual injection can take place. A malware family might use either Shotgun Injections or Targeted Injections. These two terms are introduced in this section.

4.1.1 Needed Definitions

Before we can define Shotgun and Targeted Injections, we have to introduce the set of all ϵ_{victim} entities and the set

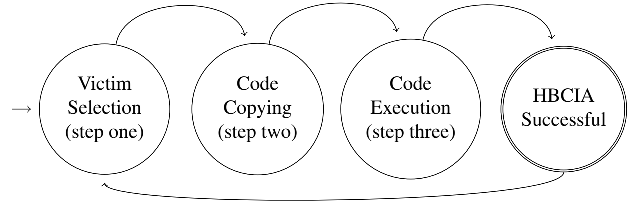

Figure 1. Simplified HBCIA algorithm consisting of three steps

of all accessible ϵ_{victim} entities from an arbitrary attacker's point of view.

We define set $\mathbb{A}$ as the set of all entities that can be detected by an arbitrary entity. For example, this can be the list of running processes that is provided by the Linux tool *ps*. $\mathbb{A}$ is defined in Definition 5.

Definition 5 *Let* $\mathbb{A} = \{\epsilon_{victim_1}, ..., \epsilon_{victim_n}\}$ *be the set of all* ϵ_{victim} *entities that can be seen by an arbitrary* ϵ_{inject}*, where* $n \in \mathbb{N}_{>0}$.

There might be many possible victim entities running on a system, but not all of them are accessible for an attacker. For example, even though the list provided by the Linux tool *ps* includes processes started by the root user, a malware might run with lower privileges and cannot access processes created by the root user. We need therefore the second set $\mathbb{B}$. It is the set of all ϵ_{victim} entities accessible to ϵ_{inject}, which is defined as follows.

Definition 6 *Let* $\mathbb{B} = \{\epsilon_{victim_1}, ..., \epsilon_{victim_m}\}$ *be the set of all entities that are accessible to* ϵ_{inject}*, where* $m \in \mathbb{N}_{>0}, m \leq n, \mathbb{B} \subseteq \mathbb{A}$.

4.1.2 Shotgun Injection

A malware family uses a Shotgun Injection, if it blindly injects code into all accessible victim entities. These victim entities usually include important system processes. A Shotgun Injection is defined as follows

Definition 7 *If* ϵ_{inject} *injects code into every element of* $\mathbb{B}$*, then this is called a Shotgun Injection.*

Shotgun injecting malware uses a greedy victim process selection method. This kind of malware tries to inject itself into every running process on a system. This might lead to complications, because the malware author cannot anticipate what processes will be running. The malware could therefore target antivirus processes and raise suspicion.

Credential stealing malware such as Zeus [24] uses Shotgun Injections. Such malware injects itself into as many processes as possible for accomplishing its goal.

4.1.3 Targeted Injection

While shotgun injecting malware attacks all accessible victim entities, malware that implements Targeted Injections only attacks a subset of them. We define a Targeted Injection as in Definition 8.

Definition 8 *If ϵ_{inject} injects code only into a subset $\mathbb{C} \subset \mathbb{B}$, then this is called a Targeted Injection.*

Malware must therefore carry out a selection process. It detects its victim entities through several features. The most simple one and commonly used feature is the process name. Malware matches the process name of each currently running process against an internal list. If a match is found, then the malware attacks this process. But there exist further ways how malware detects or could detect its victim processes like signatures, the parent process or opened file handles.

One advantage is that this kind of injection is less suspicious compared to the Shotgun Injection. Risky victim entities like system processes can be shunned. Furthermore, the malware does not waste unnecessarily system resources. Because it does not infect every accessible victim entity. It is therefore not occupying at least one thread and several megabytes of memory in each of the attacked victim entities.

The cyber espionage malware Flame [2] uses Targeted Injections. For such a kind of malware it is crucible to carefully select its victim entities, if it does not want to raise suspicion due to frequent system crashes.

4.2 Code Copying

After ϵ_{inject} has chosen an ϵ_{victim}, it has to copy its code into ϵ_{victim}. There exist several ways how code can be copied into ϵ_{victim}. Exemplarily can be named the following: debugging APIs of the OS [9], via a kernel module [10] or a buffer overflow exploitation [15].

4.3 Code Execution

Once ϵ_{inject} has copied its code into ϵ_{victim}, it has to trigger the execution of this code. There exist several OS-dependent ways of triggering a code execution. For example, this can be achieved with the help of debugging APIs (e.g. *CreateRemoteThread*) or Asynchronous procedure calls (e.g. *QueueUserAPC*) on the Windows NT platform.

Even though there exist several ways of triggering a code execution, only two distinguishable execution models exist: Concurrent Execution and Thread Manipulation. While in Concurrent Execution the original code of ϵ_{victim} continues to execute, it does not so in Thread Manipulation.

4.3.1 Needed Definitions

Before we can formally define Concurrent Execution and Thread Manipulation, we need to introduce several definitions. At first, we introduce ϵ_{victim}'s program and the set of its possibly spawned threads in Definition 9.

Definition 9 *Let $victim_{program}$ be the program that is executed in the context of ϵ_{victim}. Let $\mathbb{T}_{program} = \{t_{program_1}, ..., t_{program_n}\}$ be the set of all threads that can be possibly spawned by $victim_{program}$, where $n \in \mathbb{N}_{>0}$.*

The payload that is injected into ϵ_{victim} as well as the set of its possibly spawned threads is defined as follows.

Definition 10 *Let $payload$ be the payload that is injected into ϵ_{victim} by ϵ_{inject}. Let $\mathbb{T}_{payload} = \{t_{payload_1}, ..., t_{payload_m}\}$ be the set of all threads that can be possibly spawned by $payload$, where $m \in \mathbb{N}_{>0}$.*

Finally, the set of currently running threads in the context of ϵ_{victim} is defined as follows.

Definition 11 *Let $\mathbb{T}_{current} = \{t_1, ..., t_o\}$ be the set of currently running threads of $victim_{program}$, where $t_i \in \mathbb{T}_{program} \cup \mathbb{T}_{payload}$ and $i, o \in \mathbb{N}_{>0} \wedge i \leq o$.*

4.3.2 Concurrent Execution

If ϵ_{inject} copies its payload in addition to ϵ_{victim}'s original program $victim_{attack}$ and executes this payload in addition to $victim_{program}$, then this is called Concurrent Execution.

We define Concurrent Execution as in Definition 12.

Definition 12 *If the following assumptions hold after the injection of the payload into ϵ_{victim} by ϵ_{inject}*

1. $\exists t_i \in \mathbb{T}_{program}$
2. $\exists t_j \in \mathbb{T}_{payload}$
3. $t_i, t_j \in \mathbb{T}_{current}$, *where* $i, j \in \mathbb{N}_{>0} \wedge i \neq j$.

then this is called Concurrent Execution.

A prerequisite for a running process is at least one thread that is responsible for the process' behaviour. Once ϵ_{inject} has copied its payload into ϵ_{victim} and has executed it, the number of active threads has been increased at least by one. Afterwards, there exist at least two threads, which define the behaviour of the victim entity.

Figure 2 depicts Concurrent Execution. In this figure two scenarios are depicted: the state of two processes during an ongoing HBCIA (1) and after a HBCIA (2). While the left side shows the attacker process space, the right side shows the victim process space. In (1) the attacker process runs a dropper module, which has a pointer to a payload. The

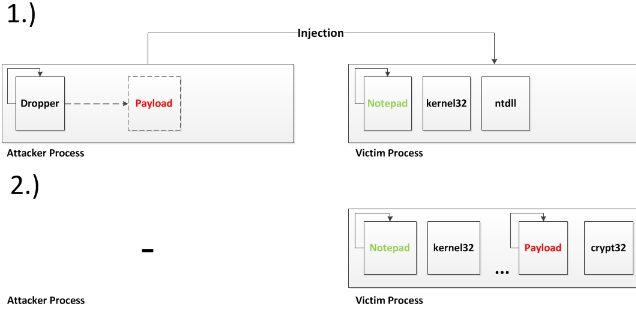

Figure 2. Concurrent execution: a dropper injects code into an office program

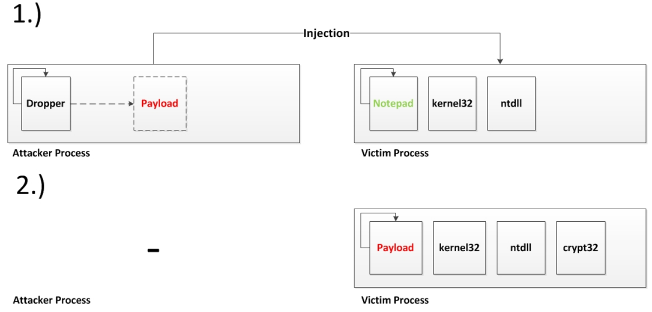

Figure 3. Thread Manipulation: a dropper hollows an office program

victim process is running the main program Notepad. Additional modules are mapped into this process space that ensure Notepad's functionality. There exists only one thread in the victim process space. In this depiction the attacker process is about to inject its payload into the victim process space. In (2) the attacker process does not exist any more, because it has terminated itself. Now the victim process has loaded additional modules (payload and crypt32). A new thread is also associated with the payload. It is running concurrently with the original thread of Notepad. This breaks with the common belief that there is only one program responsible for the behaviour of a process.

Concurrent Execution is used by banking Trojans like Zeus [24]. They inject their code into a running browser in order to employ a Man-in-the-Browser-Attack (MitBA). While the user interacts with the browser, the banking Trojan intercepts unencrypted banking credentials.

4.3.3 Thread Manipulation

If ϵ_{inject} has copied its payload to ϵ_{victim} and executes this payload by bypassing ϵ_{victim}'s threads, then this is called Thread Manipulation. Threat Manipulation typically renders $victim_{program}$ useless. A special case of Thread Manipulation is Return-Oriented Programming [20]. So far there is no malware family known that implements its whole logic with the help of this technique.

Even though a malicious payload is running inside of the victim process space, a lot of information about the victim process has not been changed. On first sight a HBCIA is hard to detect. In contrary to concurrent execution, the original program of the victim entity does not reflect 100% of the original code's behaviour. In many cases it reflects 100% of the payload's behaviour.

Thread Manipulation is formally defined in Definition 13.

Definition 13 *If the following three assumptions hold*

1. $\mathbb{T}_{current} \neq \emptyset$
2. $\mathbb{T}_{current} \cap \mathbb{T}_{\epsilon_{victim_{program}}} = \emptyset$
3. $\mathbb{T}_{current} \subseteq \mathbb{T}_{payload}$

then this is called Thread Manipulation.

Figure 3 depicts Thread Manipulation. This figure shows the state of two processes during an ongoing HBCIA (1) and after a HBCIA (2). The setting for this figure is the same as in Figure 2 in section 4.3.2. Additionally, we assume that the attacker process started the victim process. In (2) the attacker process does not exist any more, because it terminated itself. Now the victim process' old main module (Notepad) has been replaced by the payload. The payload has loaded an additional module (crypt32). The original thread executes the injected payload.

Stuxnet uses Thread Manipulation for operating secretly on a targeted system [15].

4.4 HBCIA Algorithm Taxonomy

We have identified three fundamental steps of a HBCIA algorithm in the previous sections. These steps are victim process selection, code copying and code execution. We use these fundamental steps for classifying HBCIA algorithms.

Two different models exist for the victim selection and code execution steps. There exist four different HBCIA algorithm classes in total if we combine these four models. For the first step – victim selection – exist the Targeted Injection (**TI**) and the Shotgun Injection (**SI**). For the third step – code execution – it can be distinguished between Concurrent Execution (**CE**) and Thread Manipulation (**TM**).

It is possible to name the four different classes given these abbreviations. An example would be the HBCIA algorithm of Conficker. It consists of a Targeted Injection (**TI**) and Concurrent Execution (**CE**). This allows the HBCIA algorithm of Conficker to be classified as **TICE**.

HBCIA Algorithm Class	Malware Family Example
TICE	Hanthie (Linux)
TITM	Stuxnet (Windows)
SICE	Flashback (Mac OS X)
SITM	-

Table 1. HBCIA algorithm taxonomy

Table 1 lists the four different HBCIA algorithm classes. It also lists a malware family as an example, if one exists. In only one case an example is missing. This class is the Shotgun Injection and Thread Manipulation algorithm (**SITM**). So far we could not encounter any malware family implementing such an algorithm. Manipulating the threads of as many processes as possible can be rather seen as a denial-of-service attack than a HBCIA.

5 Evaluation

We quantify the problem of HBCIAs in this section. Firstly, we estimate the prevalence of HBCIA-employing malware in a large set of samples. Subsequently, we approximate the distribution of the different HBCIA algorithm classes based on a random sampling. Then we take a look at host-based as well as network-based data. We determine the preferred victim processes and preferred network communicators with this data. A discussion of the results concludes this section.

5.1 Prevalence of HBCIAs in Malware

We estimate the prevalence of HBCIA-employing malware in this section. There has not been an estimation of the prevalence of HBCIA-employing malware to the best of our knowledge. Though it is possible to derive that HBCIA-employing malware is a relevant problem. Four of the top five malware families in 2012 use HBCIAs as based on data by Symantec [23]. They were responsible for 32.1% of all new infection reports in this year.

5.1.1 Dataset & Methodology

The considered dataset consists of 162850 samples. They were collected between 2013-03-21 and 2014-06-19. Unfortunately, the data set is not continuous. There are periods of weeks where no sample is available. These periods are (2013-09-05, 2013-09-21), (2013-11-06, 2013-11-14), (2014-03-06, 2014-03-12) and (2014-03-25, 2014-04-21).

All samples were analysed on Windows XP SP3 32 bit using the automated malware analysis system *VSAMAS* [1]. Each sandbox report has been parsed for sequences of suspicious behaviour that suggest a HBCIA occurred. An example is the sequence *AllocateVirtualMemory*, *WriteMemory* and *CreateRemoteThread*.

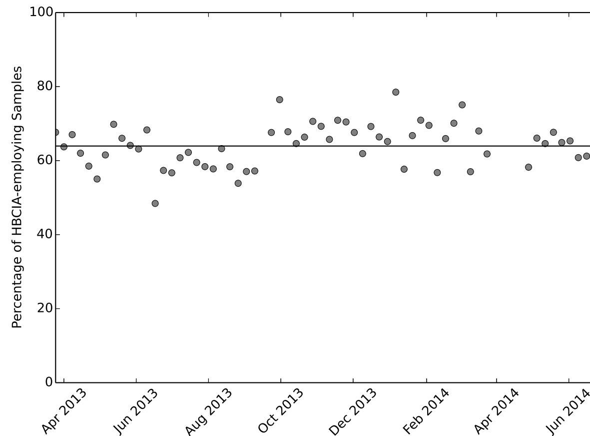

Figure 4. Weekly percentage of HBCIA-employing malware from 2013-03-21 until 2014-06-19

5.1.2 Results

Almost two thirds (63.94%, 104139/162850) of the sample set use HBCIAs. Figure 4 shows the weekly prevalence of HBCIA-employing malware as a scatter plot. The mean has been drawn as a straight line. The majority of weeks scatters closely around the mean. The minimal and maximal outliers are 48.43% and 78.5%. There is no clear upturn or downturn. This suggest that HBCIAs have been a stable feature of many malware samples throughout the last months.

The result should be considered as a lower bound. Since malware is known to use evasive techniques like sandbox detection [3], many HBCIA-employing samples might have not been successfully processed. All in all the estimation suggests that the HBCIA is a relevant problem for security researchers.

5.2 HBCIA Algorithms Used by Current Malware

We derived a classification scheme for HBCIA algorithms in section 4.4. This taxonomy consists of four classes. We also give examples for most of the classes in section 4.4. However, there exists no estimation of the distribution of current malware families to these classes. The objective of this section is therefore to roughly estimate this distribution.

HBCIA Algorithm Class	Count / Percentage
TICE	23 / 57.5%
TITM	8 / 20%
SICE	9 / 22.5%
SITM	0 / 0%

Table 2. Distribution of malware families to HBCIA algorithm classes

5.2.1 Dataset & Methodology

We collected several representatives of HBCIA-employing malware families over the course of the last months. We only have to examine one representative per malware family in order to determine the family's HBCIA algorithm since the HBCIA can be seen as a family feature [4]. The dataset contains 40 malware families of different kinds. For example, families included are banking Trojans (Bebloh), APTs (Duqu), malware droppers (Matsnu) or click fraud malware (Sirefef). A list of all the considered malware families is available on our homepage [5].

At first, we analysed each representative in a sandbox for determining its HBCIA algorithm class. In case the sample did not execute properly in the sandbox, we analysed the sample manually.

5.2.2 Results

More than one half implement a **TICE** (Targeted Injection/Concurrent Execution) algorithm. The rest is split in a group of **TITM** (Targeted Injection/Thread Manipulation) families and in a group of **SICE** (Shotgun Injection/Concurrent Execution) families. No malware family employs a **SITM** (Shotgun Injection/Thread Manipulation) algorithm. Table 2 summarizes our findings.

The results show that Targeted Injection (**TICE** + **TITM** = 77.5%) is more popular than Shotgun Injection. They show also that Concurrent Execution (**TICE** + **SICE** = 80%) is more popular than Thread Manipulation.

5.3 Preferred Victim Processes

Not only the amount of HBCIA-employing samples is of interest, but also the distribution of their victim processes. The victim processes should not be evenly distributed since not all HBCIA-employing malware families use Shotgun Injections. An analyst could prioritize the processes he investigates in order to speed up the analysis process, if this would be true.

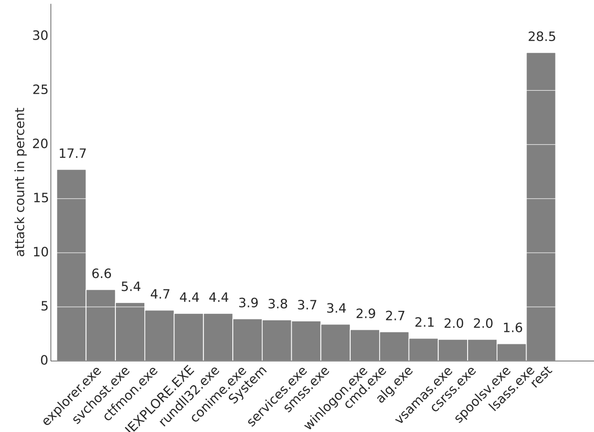

Figure 5. Relative distribution of victim processes

5.3.1 Dataset & Methodology

We use the same dataset and methodology as in the evaluation of HBCIA prevalence (section 5.1) for the determination of the preferred victim processes. Every time *VSAMAS* detects a HBCIA, it logs the attacker entity and the victim process.

5.3.2 Results

Figure 5 shows the distribution of the victim processes for the whole dataset. The most targeted process observed is explorer.exe. It has been targeted in 17.7% of all HBCIAs. Furthermore, 16 processes share almost three quarter of all attacks. These processes are all system processes – besides of vsamas.exe – found on each Windows XP installation.

This shows that the attacks are not evenly distributed. System processes are rather preferred than additional processes that do not come per default with a standard Windows installation.

5.4 Preferred Network Communicators

We have determined the preferred victim processes in the previous section. In this section, we determine the preferred network communicators. It is likely that network communication is synchronized and only one process communicates with the Internet, since many malware families inject their code into more than one process.

5.4.1 Dataset & Methodology

We analyse Intrusion Prevention System (IPS) detection data provided by a large antivirus vendor. They deploy IPS

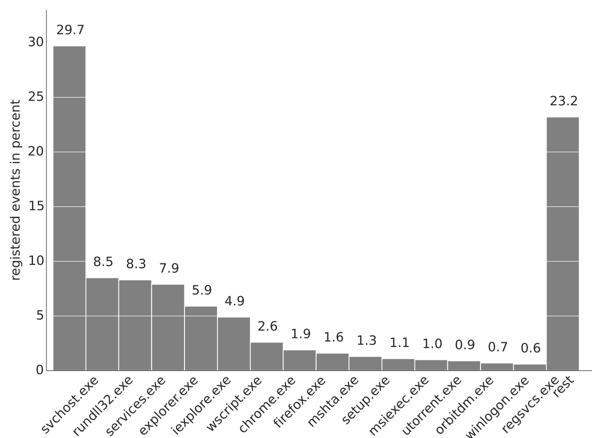

Figure 6. Relative distribution of network communicators

signatures on their customer's computers in order to block connection attempts by malware. Every time an IPS signature detects a connection attempt, a detection report is filed. Such a detection report contains the signature and the process that were responsible for triggering the alarm.

After preprocessing, i.e. removing all non HBCIA-employing malware family signatures, the data consists of 361023 detected connection attempts. All in all 274 signatures have been triggered. However, these signatures also included different malware family variants. The data was collected within a period of ten month (2013-07 - 2014-05).

5.4.2 Results

Figure 6 shows the relative distribution of network communicators. Firstly, the preferred network communicators differ from the preferred victim processes. There is a minimal overlap, for example explorer.exe, svchost.exe and rundll32.exe are in both Top 10s. Secondly, around 1/3 of all network communication is done by only one process (svchost.exe). This process is followed by other processes that are typically communicating over the network. These processes include rundll32.exe, iexplore.exe or chrome.exe.

Malware authors choose rather network-related processes for network communication. A possible explanation for their choice is that they do not want to raise suspicion. For example, personal firewalls block processes that are not known for network communication.

5.5 Discussion

HBCIAs are widely-used by today's malware. Almost two thirds (63.94%) of the malware samples in our sample set show this characteristic. A HBCIA is therefore a reliable indicator of compromise (IOC) by malware. HBCIA prevention/detection should be pursued complementary to traditional antivirus techniques.

Even though there exist four classes of HBCIA algorithms, we have only encountered malware implementing **TICE**, **TITM** or **SICE** algorithms. In general malware authors seem to prefer Target Injections to Shotgun Injections and Concurrent Execution to Thread Manipulation. The victim processes are not distributed evenly. Since not all HBCIA-employing malware families use Shotgun Injections. Analysts should prioritize the processes that they are examining. Moreover, the host perception and the network perception of a system infected with HBCIA-employing malware differs. Only the disinfection of all network communicators is not sufficient. Since processes can be infected without showing any network activity.

6 Related Work

This section discusses related work. To the best of our knowledge, there has not been any previous work that examines the foundation of HBCIAs in-depth from an academic point of view. Nevertheless, we discuss related work that focusses on detection or prevention of code injection attacks.

The authors of [13] present a novel approach for thwarting code injection attacks by randomizing the instruction set of each process. This approach has some drawbacks like the need of special support by the processor. Several improvements of this approach have been presented. Papadogiannakis et. al [17] proposed the most recent one. They present an architecture with software and hardware support for instruction set randomization.

Buescher et al. [6] propose a system that detects illegitimate manipulation of browser APIs. It is based on the idea that malware uses HBCIAs in order to manipulate these APIs. White et al. [26] propose a system for detecting code injections in memory dumps. The authors of [4] present a method for dynamically detecting HBCIAs. They apply the honeypot paradigm to processes for detecting HBCIAs operating system independently.

7 Conclusion

We have discussed Host-Based Code Injection Attacks as used by today's malware. We have shown that Host-Based Code Injection Attacks come with a lot of advantages from a malware author's point of view. These advantages include interception of unencrypted critical information or detection avoidance. However, HBCIAs come also with inconveniences such as architectural complexity and risk of system instability.

HBCIA algorithms consist typically of three steps. These steps are victim process selection, code copying and code execution. We have derived a taxonomy for HBCIA algorithms based on these steps. There exist four different HBCIA algorithm classes. An estimation of the distribution of the classes has been conducted. It shows that the **TICE** algorithm is very popular. Additionally, Targeted Injections are preferred to Shotgun Injections and Concurrent Execution is preferred to Thread Manipulation. We have also shown that 63.94% of a set of 162850 malware samples employ HBCIAs. HBCIAs are therefore a relevant problem for security researchers. The detection of HBCIAs is a promising approach for detecting malicious behaviour in general.

8 Acknowledgment

The final publication is available at IEEE Xplore via https://doi.org/10.1109/MALWARE.2014.6999410.

References

[1] Automated Malware Analysis System. Technical report, Virus Sign. http://virussign.com/vsamas.html.

[2] More human than human – Flames code injection techniques. Technical report, Cert Polska, August 2012.

[3] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[4] T. Barabosch, S. Eschweiler, and E. Gehards-Padilla. Bee Master: Detecting Host-Based Code Injection Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 11th International Conference, DIMVA 2014 Egham, England, July, 2014 Proceedings*. Springer, 2014.

[5] T. Barabosch and E. Gerhards-Padilla. List of Malicious Samples used in Host-Based Code Injection Attacks: A Popular Technique Used By Malware. http://net.cs.uni-bonn.de/wg/cs/staff/thomas-barabosch/.

[6] A. Buescher, F. Leder, and T. Siebert. Banksafe: Information Stealer Detection Inside the Web Browser. In R. Sommer, D. Balzarotti, and G. Maier, editors, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 262–280. Springer Berlin Heidelberg, 2011.

[7] Q. Dong, R. Zhao, and N. Sun. Oldboot.B: the hiding tricks used by bootkit on Android. http://blogs.360.cn/360mobile/2014/04/02/analysis_of_oldboot_b_en/, April 2014.

[8] M. W. Eichin and J. A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. pages 326–343, 1989.

[9] A. Ghizzoni. Method for injecting code into another process, Feb. 24 2004. US Patent 6,698,016.

[10] M. Giuliani. ZeroAccess an advanced kernel mode rootkit. Technical report, Prevx - Advanced Malware Research Team, 2011.

[11] J. Grunzweig. The Dexter Malware: Getting Your Hands Dirty. http://blog.spiderlabs.com/2012/12/the-dexter-malware-getting-your-hands-dirty.html, December 2012.

[12] I. Guilfanov. Windows WMF Metafile Vulnerability HotFix. http://www.hexblog.com/?p=21, December 2005.

[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, New York, NY, USA, 2003. ACM.

[14] 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.

[15] A. Matrosov and E. Rodionov. Stuxnet under the microscope. 2010.

[16] M.-E. M.Lveill. OSX/Flashback - The first malware to infect hundreds of thousands of Apple Mac. Technical report, McAfee Labs, 2012.

[17] 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.

[18] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network–Level polymorphic shellcode detection using emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 54–73. Springer, 2006.

[19] A. Portnoy. MindshaRE: Debugging via Code Injection with Python. http://dvlabs.tippingpoint.com/blog/2011/07/19/debugging-with-code-injection, July 2011.

[20] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[21] D. Schwarz. Citadel's Man-In-The-Firefox: An Implementation Walk-Through. Technical report, Arbor ASERT, 2013.

[22] J. Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. No Starch Press, San Francisco, CA, USA, 2009.

[23] Symantec. 2013 Internet Security Threat Report, Volume 18. Technical report, 2013.

[24] Unknown. Zeus source code. https://github.com/Visgean/Zeus.

[25] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Protecting web services from remote exploit code: a static analysis approach. In *Proceedings of the 17th international conference on World Wide Web*, pages 1139–1140. ACM, 2008.

[26] 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.