A Survey on Anti-Honeypot and Anti-Introspection Methods*

Joni Uitto, Sampsa Rauti, Samuel Laurén, and Ville Leppänen

University of Turku, 20014 Turku, Finland
{jjuitt,sjprau,smrlau,ville.leppanen}@utu.fi

Abstract. Modern virtual machines, debuggers and sandboxing solutions offer a robust and relatively safe way to run honeypots and analyze malware and other malicious activity. This analysis yields valuable data for threat-assessment, malware identification and prevention. However, the use of such introspection methods has caused malware authors to create malicious programs with the ability to detect and evade such environments. This paper presents an overview on existing research of anti-honeypot and anti-introspection methods. We also propose our own taxonomy of detection vectors used by malware.

1 Introduction

Honeypots can be thought as a means of capturing and analyzing malicious behaviour and traffic on networked computer systems. Depending on the level of involvement, honeypots can be roughly divided into three categories: low-interaction honeypots, medium-interaction honeypots and high-interaction honeypots [20].

Low-interaction honeypots (LIHPs) are fairly simple. Their main functionality is to provide an out-facing interface to masquerade as legitimate service provider and detect irregular activities. As the usual use-case for LIHPs is deployment to a corporate network (or similar), almost all traffic directed at the honeypot is illegitimate. Once an intrusion is detected, system administrators are alerted and the honeypot shuts down.

Medium-interaction honeypots (MIHPs) are a little more involved than LIHPs. Instead of simply presenting a selection of out-facing interfaces, MIHPs usually have a specialized task or target. For instance, a MIPH might be configured to service as a Telnet honeypot. Casting aside all other functionality, the honeypot aims to provide as convincing interaction with the selected interface domain as possible. Having a strict restriction upon which to build the system, it is possible to deceive malware with highly convincing interaction.

High-interaction honeypots (HIHPs) are even more involved; their purpose is to let malware infiltrate into the system in order to gain better understanding of the methods, technologies and identities of the malicious adversaries. While it is clear that HIHPs offer a much better view on the activities of these malicious characters and better opportunities to collect valuable data, they are also more vulnerable. Having access on the inside means that the malicious program has much wider attack surface at its disposal and the honeypot might be compromised.

In this paper, we survey the anti-honeypot and anti-introspection methods used by malware based on existing literature. Based on the existing taxonomies of honeypot detection vectors, we propose our own taxonomy that we believe to be a good fit for the current threat landscape. We also categorize the existing research on anti-honeypot techniques and anti-introspection methods according to our presented taxonomy.

2 Anti-honeypot methods against low- and medium-interaction honeypots

Low- and medium-interaction honeypots both differ from high-interaction honeypots by not letting the attacker infect the system. LIHPs and MIPHs simply 'converse' with their mark

* The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation, DIMECC Oy and Cyber Trust research program for their support.

to the extent they are allowed, trying to gather as much data as possible. Thus the common way to detect these two types of honeypots is by developing and applying fingerprinting methods.

2.1 Network level fingerprinting

Low-interaction honeypots are usually deployed using some form of virtualization or simulation. It is not uncommon to have multiple LIHPs running on the same hardware. This in turn has a clear performance impact that reflects on the network response times. As discussed in [19], these latencies can be used to profile network traffic and remotely detect honeypots. While the results depend heavily on the network topology, technologies and profiling methods in use, ideal conditions may yield up to 95% detection rate simply using ICMP ECHO request which is a fairly low-priority network message, resulting in large variance.
Similar work on network traffic profiling is done in [8]. Fu et al. measure network link latencies and build a classifier based on Neyman-Pearson decision theory. This classifier is used to detect honeynets build on Honeyd [22] low-interaction honeypot implementation. They achieve a similar detection rate as in [19]. Large portion of this result is explained by large difference between software and hardware timing resolution. Hardware-based timers used by real network devices operate on microsecond scale while the software based timers operate on millisecond scale. In their paper Fu et al. [8] suggest and implement patches to Linux kernel and Honeyed software in order to increase the timing resolution. According to their experiments, this counter-acts the fingerprinting quite well.

2.2 Application and service level fingerprinting

Low-interaction and medium-interaction honeypots are usually built in a compact manner. They aim to simulate a very specific kind of a system, usually targeted to detect specific type of attacks. The simplest low-interaction honeypot is a simple logging program listening on port 22 (SSH protocol) and recording all received data to a log file. This makes the honeypots easier to develop, deploy and maintain. However, it also makes them easier to recognize.
Attackers build complex fingerprints based on the notion of related services. If a network host A offers services X and Y, it is common that the host A should also offer services W, E, and R. A real network host would then have the whole set of services (or some large subset) while the honeypot might be offering only X and Y. This leads to easy detection by attackers.[19]

3 Anti-honeypot methods against high-interaction honeypots

The aim of high-interaction honeypots is to get infected by malicious software and act like a real system as convincingly as possible. This allows the researchers or other security experts to monitor and gather data on the malicious entity. Currently, the most common and interesting catch is a botnet worm, as these entities are highly undetectable and persistent. Being part of a botnet with deep anti-introspection and data analysis tools can yield invaluable operational and technological information.
Today's malware is very introspection-aware and in most cases it is only a matter of time before the honeypot is recognized and the infected host is cast aside. This is especially true for botnets, where the bot-herder has a greater control over the infected hosts. On the positive side, gaining understanding on the methods of discovery of honeypots is valuable information on its own right [15]. HIHPs are more or less real systems which renders them less secure than LIHPs and MIHPs, as the potential attack surface is much larger and the honeypot might end up under the attacker's control.

3.1 System level fingerprinting

High-interaction honeypots must resemble real systems as closely as possible in order to fool a dedicated black hat hacker. This requires convincing system components and software. To

achieve this, HIHPs are often implemented as real operating systems with real software. This operating system is either run on a virtualization platform or on real hardware. In the former case, the introspection and data collection tools are bundled in with the honeypot. In the latter, some hooks are used to monitor the virtualized operating system. In either case, these monitoring systems leave traces in the system. Examples of such traces are irregular hardware components, uncharacteristic behavior (e.g. large system call latencies) and hints of monitoring software [6, 15, 13, 14].
The attackers develop capabilities for detecting these anomalies (or do it manually, if need be) and build methods and fingerprints for detecting monitored systems that are potentially honeypots. Research in the early 2000 is highly concerned with virtualization being an instant kill-switch for the attackers. However, fifteen years of network infrastructure development has seen the raise of huge numbers of virtualized services, which means those concerns are no longer as valid as they used to be. Although we do not cover this issue in detail in this study, techniques malware uses to detect virtual machines and hypervisors still have some significance [4].

3.2 Operational analysis

A harder problem for honeypots that wish to remain undetected is the law. Infected hosts are not allowed to continue to infect other systems if the administrator is aware of the infection. To do otherwise is negligent. This enables the botnet masters to simply use infection as means of verifying the nature of the current host. The infected host tries to infect a collection of potential targets. Within those targets, a control node is hidden. If the control node receives the infection, it can then verify the host as a valid infection target. Otherwise the host is dropped. [26]
Other operational analysis consists of monitoring the system behavior and trying different functions and contrasting the results against expected values. Port scanning and fetching resources from Internet is also common, as the honeypots often have limitations related to this functionality.

4 Detection vector taxonomies

4.1 Detection vectors

A detection vector describes the means the malware can use to detect an unwanted execution environment. Such environments can consist of but are not limited to operating systems with sub-optimal update compositions, virtualized execution environments, sand-boxing solutions or debuggers. Modern malware is more and more environment-aware.
Chen et al. have captured and run 6900 malware samples in different environments [2]. They found out that when run in a virtual machine, 95.3% of the malware continued to exhibit the same behaviour as in a normal operating system. However, when run with a debugger attached, 58.5% continued to function as before. Nearly half of the tested malware had some capability to identify debuggers and upon detection reduce or stop malicious behaviour. After these experiments, the malware has probably become even more sophisticated.

4.2 Existing taxonomies

In the next subsection, we will present a proposed taxonomy of detection vectors. Our taxonomy is based upon two existing taxonomies, the first one presented by Chen et al. in [2] and the second by Gajrani et al. in [9], with some extensions to accommodate detection vectors available to botnets. This subsection briefly outlines the two earlier taxonomies, presenting the differences between the two. We then proceed to tie the two taxonomies together.
Chen et al. present four abstract classifications, each with two subcategories. These categories are fairly general. They also describe the access level the attacker needs to have in order to leverage these detection vectors, how accurate the methods are, how complex they are to employ and how easily these detection vectors can be evaded. The categories identified by Chen et al. are presented below. Concrete examples for most of these categories can be found in [16].

- **Hardware.** System anomalies that are hardware-detectable, like hardware breakpoints.
  - *Device.* Virtual environments tend to display devices that either differ from original models or are completely VM dependent.
  - *Driver.* Many virtual machines and debuggers tend to create drivers that are characteristic to those environments.
- **Environment.** This category covers notable differences in the actual execution context.
  - *Memory.* Instrumentable environments tend to have memory traces that identify those environments. Open pipes or channels, OS flags, altered memory layout, etc.
  - *System.* There are often execution environment specific idiosyncrasies which are results of bugs, implementation details and other factors. Examples include CPU instruction bugs in VMs and call-stack modifications by debuggers.
- **Application.** This category deals with the surrounding software ecosystem and its inherent fingerprints.
  - *Installation.* The tools used to instrument malicious software have well-known components installed in well-known locations. While this information is relatively easy to mask, it is also a very potent detection vector.
  - *Execution.* This category deals with running processes and is similar to installation. Processes are easy to detect but also easy to hide.
- **Behavioral.** Measurements on how the system performs operations and responds to requests.
  - *Timing.* Introspection environments tend to display some latency in instructions and network messages. The latencies are hard to hide and provide a fairly reliable detection vector.

The taxonomy presented by Gajrani et al. is of much finer granularity: the paper outlines a total of 12 different categories. Some of these are phone and Android dependent as the paper discusses sandboxing and detection of sandboxes for Android applications. These categories are included for completeness. Categories identified by Gajrani et al. are:

- **Background Process.** Processes specific to an emulator/VM.
- **Performance.** CPU instruction latency, graphics performance, etc.
- **Behavior.** SMS operation, in- and out-bound phone calls.
- **Software Components.** Google Play services etc.
- **API.** Binder APIs like isTetheringSupported().
- **Initial System Design.** Contact list, battery status, network status, etc.
- **Hypervisor.** QEMU scheduling, instruction pointer updates (QEMU only updates upon non-linear execution points), cache behavior, etc.
- **File.** Emulator specific files, or device-specific files.
- **Network.** Android emulators tend to sit behind a default gateway and DNS.
- **Sensors.** Emulators tend to not simulate sensors (or only simulate a small subset).
- **Device Build.** Collection of build values can be used to fingerprint and identify the execution platform. For example, *Build.BOARD = unknown* could be a sign of an emulator.
- **Phone ID.** Smart phones have unique IMEI (International Mobile Equipment Identity) and IMSI (International Mobile Subscriber Identity) numbers, phone numbers, etc. These are typically set to default values on emulation platforms.

5 Our detection vector taxonomy

The taxonomy presented by Chen et al. is more abstract and broad whereas Gajrani et al. present a more concrete taxonomy, where each category is tied to fairly specific system aspects. We believe that the taxonomy by Chen et al. is more usable for the general case. However, for the current threat landscape, it would benefit from increased granularity. A more fine-grained approach would also allow us to better separate different categories between LIHPs and HIHPs. We therefore suggest a modified and expanded 2-tier taxonomy:

- **Temporal.** As timing is a fairly prevalent detection vector, it has its own category.
  - *Network.* Ways of detecting latencies in the network environment.
  - *Local.* Detection of latency in the actual system, on API and instruction level.
- **Operational.** This category covers vectors related to how the machine operates and which operations are possibly allowed/disallowed.

- *Propagation.* In most honeypots, further malicious propagation is denied.
- *Communication.* Does the device communicate as usual and what kind of communication is allowed?
- *Operation.* This covers other operations such as port scanning which may be limited in certain environments.
- *Idiosyncrasies.* As described by Gajrani, some execution environments display clear operational differences, such as the lazy program counter in QEMU.

– **Hardware.** This matches the categories presented by Chen et al.
- *Device.*
- *Driver.*

– **Environment** Details about the operation environment serve as detection vectors. These are the most simple to access and leverage.
- *Data.* What data resides on the machine; files, installed programs, etc.
- *Execution.* What else is executing on the machine. Identifiable monitor processes, certain auxiliary services, etc.
- *Identity.* Values associated with device/operating system identity, mostly present on mobile platforms.
- *Memory.* Identifiable memory traces such as operating system flags, open pipes, altered memory layout, etc.

Table 1 divides the research in the field of anti-honeypot techniques into categories we proposed in the previous section. Some categories have a marking "HIHP-only". This means that these techniques are not necessarily available for LIHPs as they tend to simulate smaller, incomplete systems.
While the collection of papers is by not exhaustive, we believe it conveys a picture about the current research field and subjects. While the details of attack vectors are usually quite implementation specific [25], they also have common characteristics that make categorization worthwhile.

Category	Subcategory	Papers	Honeypot
Temporal	*Network*	[8, 19]	LIHP & HIHP
	Local	[15]	Mostly HIHP
Operational	*Propagation*	[3, 27]	HIHP-only
	Communication	[17, 27]	Mostly HIHP
	Operation	[6, 12, 16, 27]	LIHP & HIHP
	Idiosyncrasies	[16, 24, 13]	LIHP & HIHP
Hardware	*Device*	[12, 15]	HIHP-only
	Driver	[9]	HIHP-only
Environment	*Data*	[6, 7, 12, 15, 16]	HIHP-only
	Execution	[6, 12, 15, 16, 19]	LIHP & HIHP
	Identity	[9, 13]	LIHP & HIHP
	Memory	[6, 7, 15, 16, 13]	HIHP-only

Table 1. Distribution of anti-honeypot related research within our taxonomy.

6 Conclusions and future work

In this paper, we have surveyed anti-honeypot and anti-introspection methods malware uses against low-, medium- and high-interaction honeypots. We also discussed different ways to

categorize these approaches and also presented our own improved taxonomy that better corresponds to the present situation. According to current research, botnets are the most notable threat in the wild. Botnets are versatile, controllable, and offer various business opportunities for the enterprising black hat hacker. Honeypots are an invaluable tool for detecting and monitoring botnets, but botnets are also the most honeypot-resistant malicious entities [3, 26]. New ways to categorize detection vectors and new kinds of honeypot solutions are therefore needed.

Modern malware can leverage multiple detection vectors to determine whether or not it is operating in a monitored system or under direct introspection. These methods rely on finger-printing different aspects of the operation environment and interfaces. The most effective strategy against such finger-printing are novel monitoring solutions, rendering existing finger-prints void. However, this is prohibitively expensive and better solutions are needed. In [2] Chen et al. suggested that normal operating systems could try to imitate surveilling machines, rendering surveillance detection less effective. Also, more robust tools are required. In [1] Bahram et al. devise a method for totally circumventing introspection based on virtual machines by modifying kernel data structures used by the guest operating system.

As future work, we aim to develop a honeypot system which provides several dual interfaces [18]. The other group of interfaces consists of the old interface and the other contains new, diversified interfaces. The aim is that legitimate software and users interact with the new diversified interfaces. The old, unmodified interfaces work as fakes. Any interaction with the fake interfaces immediately rises alarm in the system. To gather meaningful data, the fake interface should engage the malware in a meaningful way. This setup has the typical requirements of a high-interaction honeypot. The (assumed) malicious entity needs to be provided with an illusion of proper request-response chains and the introspection system must remain undetected.

Other interesting avenue of research could be some form of self-aware, adaptive introspection environment. This system needs to be able to detect that is has been detected. After detection it would initiate a replication phase, launching multiple clones of the previous setup with a few modifications, much in a genetic fashion. These test would be iterated until a candidate system which evades the detection emerges.

Finally, the increased use of virtualization today may have rendered many of the old anti-honeypot and anti-introspection methods useless, but malware authors keep coming up with new ways to detect introspection. In light of this development, we need to consider good ways to keep monitoring malware without being noticed. One topic of research we consider interesting is advanced deception in the context of HIHPs. Some steps have already been taken into this direction. Methods that can be used in HIHPs to communicate with malware and keep deceiving it as long as possible are addressed in [23], [5] and [11]. A game theoretic approach to the same problem is taken in [21] and [10]. These kinds of advanced approaches greatly help us in convincing malware it is indeed operating in an ordinary system.

References

1. S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 82–91. IEEE, 2010.
2. X. Chen, J. Andersen, Z.M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.
3. C. Costarella, S. Chung, B. Endicott-Popovsky, and D. Dittrich. Hardening honeynets against honeypot-aware botnet attacks. *University of Washington, US*, 2013.
4. T. Credo. Hyper-v how to: Detect if you are inside a vm. https://blogs.technet.microsoft.com/tonyso/2009/08/20/hyper-v-how-to-detect-if-you-are-inside-a-vm/, 2009.
5. W. Cui, V. Paxson, N. Weaver, and R.H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, 2006.

6. M. Dornseif, T. Holz, and C.N. Klein. Nosebreak-attacking honeynets. *arXiv preprint cs/0406052*, 2004.
7. O. Ferrand. How to detect the cuckoo sandbox and to strengthen it? *Journal of Computer Virology and Hacking Techniques*, 11(1):51–58, 2015.
8. X. Fu, W. Yu, D. Cheng, X. Tan, K. Streff, and S. Graham. On recognizing virtual honeypots and countermeasures. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 211–218. IEEE, 2006.
9. J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M.S. Gaur, and M. Conti. A robust dynamic analysis system preventing sandbox detection by android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks*, pages 290–295. ACM, 2015.
10. O. Hayatle, H. Otrok, and A. Youssef. A game theoretic investigation for high interaction honeypots. In *IEEE International Conference on Communications (ICC)*. IEEE, 2012.
11. O. Hayatle, H. Otrok, and A. Youssef. A markov decision process model for high interaction honeypots? *Information Security Journal: A global Perpective*, 22(4):159–170, 2013.
12. O. Hayatle, A. Youssef, and H. Otrok. Dempster-shafer evidence combining for (anti)-honeypot technologies. *Information Security Journal: A Global Perspective*, 21(6):306–316, 2012.
13. T. Holz and F. Raynal. Defeating Honeypots: System Issues, Part 1. http://www.symantec.com/connect/articles/defeating-honeypots-system-issues-part-1, 2005.
14. T. Holz and F. Raynal. Defeating Honeypots: System Issues, Part 2. http://www.symantec.com/connect/articles/defeating-honeypots-system-issues-part-2, 2005.
15. T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 29–36. IEEE, 2005.
16. A. Issa. Anti-virtual machines and emulations. *Journal in Computer Virology*, 8(4):141–149, 2012.
17. N. Krawetz. Anti-honeypot technology. *Security Privacy, IEEE*, 2(1):76–79, Jan 2004.
18. S. Lauren, S. Rauti, and V. Leppänen. An interface diversified honeypot for malware analysis. Accepted to MeSSa 2016.
19. S. Mukkamala, K. Yendrapalli, R. Basnet, M.K. Shankarapani, and A.H. Sung. Detection of virtual environments and low interaction honeypots. In *Information Assurance and Security Workshop, 2007. IAW'07. IEEE SMC*, pages 92–98. IEEE, 2007.
20. M. Nawrocki, M. Wahlisch, T.C. Schmidt, C. Keil, and J. Schonfelder. A survey on honeypot software and data analysis, 2016. *arXiv preprint*, 2016.
21. J. Pawlick and Q. Zhu. Deception by design: Evidence-based signaling games for network defense. In *Workshop on the Economics of Information Security (WEIS)*, 2015.
22. N. Provos. Honeyd Virtual Honeypot. http://www.honeyd.org/.
23. S. Rauti and V. Leppänen. A survey on fake entities as a method to detect and monitor malicious activity. Submitted to 5th World Conference on Information Systems and Technologies.
24. L. Spitzner. Problems and Challenges with Honeypots. http://www.symantec.com/connect/articles/problems-and-challenges-honeypots, 2004.
25. D. Sysman, S. Itamar, and E. Gadi. Breaking Honeypot For Fun and Profit Honeypots. Black Hat USA 2015. http://winehat.net/wp-content/uploads/2015/10/Dean-Sysman-BreakingHoneypots.pdf .
26. P. Wang, L. Wu, R. Cunningham, and C. Zou. Honeypot detection in advanced botnet attacks. *International Journal of Information and Computer Security*, 4(1):30–51, 2010.
27. C. Zou and R. Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 199–208. IEEE, 2006.