Impeding Automated Malware Analysis with Environment-sensitive Malware

Chengyu Song
Georgia Institute of Technology
csong84@gatech.edu

Paul Royal
Georgia Institute of Technology
paul@gtisc.gatech.edu

Wenke Lee
Georgia Institute of Technology
wenke@cc.gatech.edu

Abstract

To solve the scalability problem introduced by the exponential growth of malware, numerous automated malware analysis techniques have been developed. Unfortunately, all of these approaches make previously unaddressed assumptions that manifest as weaknesses to the tenability of the automated malware analysis process. To highlight this concern, we developed two obfuscation techniques that make the successful execution of a malware sample dependent on the unique properties of the original host it infects. To reinforce the potential for malware authors to leverage this type of analysis resistance, we discuss the Flashback botnet's use of a similar technique to prevent the automated analysis of its samples.

1 Introduction

Malware analysis is the process of understanding the behavior of malicious programs. Information collected from malware analysis can be used to detect similar malware, repair damaged systems, and dismantle malicious infrastructure (e.g., C&C servers) [15, 2, 7, 16, 8]. As these activities endanger the cyber criminal's profit model, attackers have continuously developed techniques to prevent malware from being analyzed. In response, defenders have created new techniques [14, 10, 1, 5, 13] to address malware analysis resistance. This arms race has lasted for decades and in general, neither side has yet claimed a definitive advantage.

In this paper we demonstrate that through novel reuse of existing techniques traditionally reserved for digital rights management (DRM), automated malware analysis can be made ineffective and unscalable. Specifically, we introduce the concepts of host identity-based encryption (HIE) and instruction set localization (ISL), which interrelate the successful execution of a malware sample with unique properties of the original host it infects. Note that these techniques are not intended to prevent a human analyst from employing manual efforts to understand the behavior of a particular piece of malware (e.g., Stuxnet).

1.1 Background

Many early software obfuscation techniques (such as packing) were designed to complicate static analysis and omitted concerns about what an analyzer could learn by running a program. Later, virtual machine-based obfuscation was used to prevent static analysis on dumped memory images (e.g., which can include unpacked code). Thus, while these techniques have successfully rendered static analysis infeasible, they do not pose a significant threat to dynamic analysis. As part of evolutions in the threat landscape, malware authors realized this flaw and began integrating dynamic analysis detections. As an example, some malware samples will attempt to detect whether they are being debugged [11], or whether execution is occurring in a virtual environment [9]. If detection is successful, a given malware instance will not exhibit malicious behavior. Although these techniques have met with some degree of success, they remain brittle to mitigation, as researchers can devise new techniques (e.g., [14, 5]) to make their analysis environment appear normal. Based on this observation, our obfuscation techniques are designed to be effective against two key conceptual challenges faced by malware authors:

- (**A1**): It is difficult to reliably distinguish an analysis environment from a production environment;
- (**A2**): It is impossible to hide high level (e.g. system call and network) behaviors from dynamic analysis.

The motivation behind overcoming these challenges is that the effectiveness of our obfuscation techniques should not be limited to existing analysis environments or mechanisms. Concisely, they should be able to resist all potential analysis techniques in the predictable future.

The Goal. Instead of *preventing a human analyst from understanding a piece of malicious software*, the goal of our obfuscation techniques is to *prevent human analysts from analyzing the malware efficiently* with what have become ubiquitous automated means. Since manually analyzing the entirety of new malware samples created each day is known to be untenable, we will focus on defeating automated malware analysis.

1.2 Defeating Automated Malware Analysis

The intuition behind our obfuscation techniques arises from the decades-old operational model of the anti-malware industry. This model consists of four phases: capture, analysis, signature generation, and runtime detection. In the first phase, malware samples are collected in several ways: honeypots, mail filters, web crawlers, client submissions, and malware exchanges. Samples collected are then put into an isolated environment for analysis. Any sample that is believed to be malicious will be sent to the signature generation phase, where invariant characteristics will be extracted from the binary. Finally, the generated signature will be dispatched to individual antivirus clients to detect the malware instance, or better, other variants of the same family.

The problem with this model is that *malware samples are captured and analyzed in two different environments*. Therefore, instead of trying to detect a particular analysis environment or popular virtualization container used to create automated malware analysis systems, we assume that any environment other than the original *is* an analysis environment. In other words, malware that possesses the protections we propose will behave maliciously only on the original host that was infected; any attempt to execute a given sample in another environment will result in incorrect execution. This goal is achieved through two techniques.

Host Identity-based Encryption (HIE). Before deploying a malware instance on a given system, information (based on system hardware and software) that can uniquely identify this system is collected. This information is then used to derive a key (or host ID) that will be used to encrypt certain portions of the malware instance. At runtime, the malware instance will gather the same set of information again and use it to derive a decryption key. Thus, if the instance is put into a different execution environment, decryption will fail and the sample will not exhibit malicious behavior.

A straightforward implementation of HIE involves encrypting the entire malware binary (e.g., encryption key-based packing). While such an approach could provide ideal protection for a program, this method may actually benefit the defender. As an example, once an analysis system is made aware that the sample leverages HIE, execution failure could be used to determine whether the host ID is correct, which can ease the process of brute-forcing the decryption key. A more appropriate use of this technique may involve encryption of only mechanisms critical to the malware. As an example, portions of code or data associated with the sample's domain name generation algorithm (used to contact C&C servers) could be encrypted. Then, even if decryption fails, the sample will attempt to resolve or connect to the wrong C&C server. The malware analysis system would in turn treat this information as real.

HIE has two major advantages. First, it uses modern cryptography, which means that knowledge of how a key is derived does not affect the integrity of the protection; unless the defender can guess the same decryption key, they cannot unlock the sample. Second, any two instances of malware will possess different decryption keys; intelligence gathered from successfully analyzing one malware instance provides no advantage in analyzing the second. As an example, some malware instances attempt to detect whether their execution is inside a virtual environment (e.g., by detecting the emulated hard drive). Once a defender discovers this particular method, they can modify their environment to mitigate attempted detections by all samples that use it. However, with HIE, even if the analyst knows sample a expects environment α, this result cannot be used to run a sample b that expects environment β.

Although it shares similarities with conventional DRM techniques (i.e., locking software to a specific environment), HIE is different in several ways. First, to prevent protection bypass, DRM systems often assume the highest privilege level on a system or utilize special hardware. Without precluding itself from portions of its target audience, HIE cannot (and does not) make the same assumptions. Second, the goal of DRM is to prevent the protections of even a single instance of copyrighted material from being broken (because the unprotected copy could then be widely distributed). In contrast, to be effective, HIE only needs to prevent the successful processing of large volumes of malware in an acceptably small time period. Finally, although it is possible to enumerate all possible host configurations and brute force a decryption key, use of a sufficiently large key space will make this process unacceptably inefficient.

Despite its advantages, host identity-based encryption is not considered sufficiently resistant to forgery. Thus, we also propose a network-based identifier that is derived at the C&C server. The combination of host and network-based keys are used by instruction set localization (ISL), a second technique that provides a malware instance running on an infected system with its actual malicious behavior.

Instruction Set Localization. Instruction set virtualization (e.g., as used in VMProtect and Code Virtualizer) is an obfuscation technique that protects software by transforming the source code, intermediate representation, or native machine code of a program into bytecode for an arbitrarily chosen instruction set architecture. At runtime, the execution semantics of the original program are fulfilled by a native interpreter bundled with the bytecode. Malware obfuscated with this technique is thus not vulnerable to memory dump or unpacking strategies, as the executable code present in the binary is bytecode representing an unknown machine language. This property makes instruction set virtualization a successor to more traditional transformations that encrypt, compress, or otherwise reversibly transform natively executable code. However, as mentioned above, this technique does not pose a significant challenge to dynamic analysis, as researchers have already developed techniques [13, 4] capable of automatically reverse engineering bytecode execution semantics.

To fix the weaknesses associated with instruction set virtualization, we propose a technique similar to HIE, whereby a virtualized instruction set is bound (or *localized*) to a specific environment. In this scenario, a malware instance deployed on a given system represents only an interpreter of bytecode for a virtualized instruction set. All malicious tasks, which will be requested from and provided by the C&C server, represent bytecode to be interpreted.

The interpreter's request for a task includes the host identifier of the infected system. The C&C server combines the host identifier with a network identifier and uses this information as part of virtualizing the native code representing the malicious task. The bytecode given to the infected host will thus only run on that specific host, as determined by forgery-resistant host and network-based identifiers.

Similar to HIE, this obfuscation technique offers several guarantees. First, unless the interpreter deployment-time (or infection-time) signature matches the runtime signature, the task cannot be executed correctly due to incorrect bytecode interpretation. Second, the only way to understand the task is to correctly determine its interpretation, such as by brute-forcing the combination of host and network identifiers.

Malware instances that use ISL hold several advantages. First, they will (in general) be more extensible, in a manner similar to a Platform-as-a-Service (PaaS) model. In addition, a given instance will contain no information about its actual malicious tasks, which can complicate the identification of behaviors. Finally, botnets assembled using these instances will offer better resistance to analysis and tracking, as without access to the original infected host, researchers and security practitioners will need to forge both host and network-level identifiers.

2 Prototype Design

In this section, we present additional details on the design of host identity-based encryption (HIE) and instruction set localization (ISL).

2.1 Host Identity-based Encryption

The operational particulars of host identity-based encryption present two major challenges. First, the specifics of generating a sufficiently unique ID for the host must be decided. Second, deployment logistics must be worked out, which will include the use of intermediate code agents that determine the host ID to which the delivered malware instance will be bound.

Host ID Generation. As mentioned previously, identifier generation is a critical part of a DRM system. If a DRM scheme fails to generate an appropriately unique ID, the protections otherwise afforded are easily bypassed. For this reason, many schemes require execution at the highest privilege levels (e.g., the OS kernel or hypervisor), or special hardware capable of providing trusted information (e.g., TPM).

Identifier generation is likewise a critical part of HIE and ISL, but requires additional considerations. First, a malware author cannot assume availability of the highest privilege level, as in contrast to legitimate DRM systems, malware may execute as an unprivileged user. For similar reasons, attackers also cannot assume there will be hardware support. Thus, the information used must be retrievable without any privilege.

Second, information for identifier generation cannot be collected from a single source, as unlike a legitimate DRM system, this information may not be trusted. Thus, to prevent brute force attacks, host identifier generation must use multiple sources to increase key space. However, the information must also be as stable as possible to minimize the introduction of false positives. As the stability of information usually decreases with size, a tradeoff must be made.

To the benefit of the proposed system, on current commodity systems, there exists sufficient information that is unique, invariant, and requires no privilege to obtain. As an example, on Windows systems, the following could be used to create the host-based identifier:

- **The Environment Block**. When a process is created, Windows stores environment information in the process' address space. In our design we use the process owner's username, computer name, and CPU identifier. As the environment block is directly accessible by code that executes inside a given process, this information can be obtained without API or system calls.

- **MAC address**. The MAC address of the NIC can be obtained from the `GetAdaptersInfo` API.

- **GPU info**. GPU information can be obtained from the `GetAdapterIdentifier` method of `IDirect3D9Ex` interface. In our design, we use the device description.

- **SID of the user**. Using the token of a process, the `GetTokenInformation` API can be used to obtain the SID of the process' owner. This identifier is unique across a Windows domain.

Once collected, this information can be concatenated and used as an input to a cryptographic hash function (e.g. bcrypt) to generate the host identifier, which will then be used as the encryption/decryption key. Due to time constraints, we did not examine the viability of file system or registry information to supplement the above design, but believe they are likewise a rich source of stable, uniquely identifiable information.

Malware Deployment. As mentioned previously, an additional challenge is introduced by the need to obtain a host identifier prior to deploying a malware instance bound to that identifier. Solutions to this problem are complicated by exploit reliability concerns that mandate shellcode be as small as possible. Thus, gathering information used to produce the host identifier must instead be deferred to an intermediate downloader agent.

The use of multiple, intermediate deployment agents creates an additional hurdle. That is, if security practitioners capture exploit shellcode or the intermediate downloader, they could use these agents to obtain a malware instance bound to their analysis environment. To solve this problem, we propose using one-time URLs similar to those offered in password reset procedures. More specifically, before the shellcode or downloader is sent to the infected host, the server will assign it a unique path to download the next stage. Although the operational specifics will vary based on the attack vector (e.g., drive-by download versus email attachment), in all cases the one-time URL will also be short-lived.

2.2 Instruction Set Localization

The design of ISL requires overcoming three major challenges: network identifier generation, transformation rule generation, and fault tolerance.

Network Identifier Generation. Network identifiers are already used by companies like Google and Facebook to prevent session hijacking. Similar to their selections, the following could be used as the network identifier:

- **Geo-location**. The IP address is the most straightforward candidate for the network identifier, but is not sufficiently stable. For this reason the geo-location of the IP address should be used instead at the granularity of state or province. This choice of granularity permits a certain level of mobility while maintaining the objectives of ISL.

- **Autonomous System Number (ASN)**. In general, geo-location alone comprises a sufficient network identifier. However, as the publication of this information is not mandatory, geo-location databases can contain outdated or incorrect data. For this reason, the ASN should be used as well.

As it is collected at the C&C server, we consider information used to create the network identifier unforgeable.

Transformation Rule Generation. Ideally, an execution environment's *unique ID* (the combination of the host and network identifiers) would be mapped to a unique virtual instruction set. However, as designing such a system exceeds the scope of this paper, we instead propose an alternative but equally effective approach. That is, only one virtual instruction set is used, but each malware instance receives a *task decryption key* derived from the unique ID when deployed. When responding to a task request, the C&C server will encrypt the task using a key derived from the malware instance's unique ID. If there is a mismatch (e.g., in the network identifier used to create the unique ID), the decryption routine will generate invalid or incorrect bytecode that does not reveal the malicious task.

As the proposed design replaces per-host unique virtual instruction sets with task decryption keys, generation of this component merits additional discussion. Although straightforward, creating a task decryption key by simply hashing the unique ID could compromise the protections provided by ISL. In this scenario, security practitioners could leverage knowledge of the features used in unique ID generation to reproduce a key for a given execution environment, or worse, search a malware analysis database to identify samples that use the same key and successfully decrypt tasks sent to them. To prevent this potential attack, a keyed hash (e.g., HMAC) should be used instead; the corresponding key would be kept on the C&C server and known only to the botnet operator.

3 Discussion

Operational Security. Both HIE and ISL are implemented using modern cryptography and thus are immune to knowledge of how keys are generated– the only way to break their protections is to derive the correct keys. As security organizations automatically analyze malware in environments separate from those originally infected, derivation of the correct keys requires searching through the entire key space, which is of non-trivial size. Moreover, some configuration information (i.e., that used to derive the network identifier) may be impossible to duplicate.

Another advantage of HIE and ISL is that they are insensitive to analysis techniques. That is, regardless of the employed analysis granularity (e.g., fine-grained dataflow analysis used in [16] or high-level, blackbox network intelligence collection), the resistance offered by HIE and ISL can be broken only if the configuration parameters of the original execution environment are successfully matched.

Potential Countermeasures. One straightforward idea for bypassing the protections provided by HIE and ISL is to analyze samples in the original environments they infected. While such an approach may work for samples collected by high-interaction honeypots, for a variety of practical reasons the use of this method is not feasible for other sources. Challenges include monitoring system capability limitations (e.g., of low-interaction honeypots), legal and privacy considerations and impact on business operations and continuity (e.g., for client submissions). As samples collected by high-interaction honeypots represent only a small portion of all collected samples, the effectiveness of this approach is limited.

An alternative to analyzing malware on the systems originally infected is the collection and duplication of host and network-level environment information. However, for similar (though perhaps less significant) policy and privacy reasons, the implementation of this idea would face significant hurdles. Moreover, even if the host identifier can be successfully forged, duplication of the correct network identifier would require analysis system deployment on an unprecedented, globally cooperative scale.

Another potential countermeasure is to record and collect the network activity between an infected host and the C&C server, then replay that communication during analysis. Without one additional protection, this approach would bypass ISL and could be combined with attacks or cooperative efforts to forge the host identifier. However, the use of SSL/TLS for C&C communication mitigates the successful use of this response.

Finally, the very manner by which HIE and ISL protect a malware instance could be leveraged by the security community to create instability in a set of host or network identifiers and thus prevent successful or correct execution (i.e., the allergy attack [3]). However, this countermeasure could also make legitimate software systems that use the same information equally unreliable. As such, the success of this response may depend on the willingness of users to accept security over usability.

4 Concept Integration by Malware

At the inception of this paper, concerns were raised about the extent to which malware authors would adopt techniques like HIE and ISL; the rise of the Flashback botnet supports the practicability of their use in real-world malware.

The Flashback Botnet. In September 2011, Flashback emerged as malware that targeted Mac OS X. By April 2012, the botnet representing Flashback-infected systems had grown to over 600,000 Macintosh systems. This size makes Flashback the most considerable threat built using Mac OS X thus far.

While its size and propagation activities are interesting, the operational properties of Flashback malware reinforce the concerns expressed in this paper. To elaborate, the initial Flashback agent connects to its C&C server and downloads one or more additional payloads (e.g., those that can illegally monetize the victim's use of search engines). When requesting a payload, the agent submits the hardware UUID of the infected system. This value is then hashed (via MD5) to create a key that, in combination with the RC4 stream cipher, binds the payload to that system. Like HIE, unless the hardware UUID of the system matches the one used to create the payload, it will not execute successfully.

While Flashback's use of a system's hardware UUID bears some similarities to HIE, other aspects of the malware's operation are less sophisticated. For example, Flashback's use of encryption does not optimally protect its domain name generation algorithm, which is thus easier to reverse engineer. In addition, communication between a Flashback-infected system and the C&C server is in plaintext, which permits impersonation of the C&C.

Despite its other shortcomings, Flashback demonstrates that malware authors have already begun using protections like those described in this paper. Security researchers must therefore prepare for a future of malicious software that will only run on the systems it originally infects.

5 Related Work

Previous security research has included the design of analysis-resistant malware. As an example, Sharif, et al introduced the design and implementation of a technique called *conditional code obfuscation* [12], which encrypts trigger-based code with a key derived from an input that would activate the trigger. In a manner somewhat conceptually similar to HIE and ISL, code protected by this mechanism cannot be decrypted unless the threat analyst can derive an input that would activate the trigger.

More recently, Dunn et. al proposed using TPM to cloak the execution of malware [6]. Like HIE, this work also proposes the idea of encrypting a payload with a key specific to a particular host. In contrast to our work (which ensures that malware is not executed in an analysis environment), the goal of Dunn's research is to make host-level *monitoring* of malware infeasible.

6 Conclusion

In this paper we proposed two obfuscation techniques–host identity-based encryption (HIE) and instruction set localization (ISL)–that make the successful execution of a malware sample dependent on the unique properties of the original host it infects. Going forward, researchers must include ways to mitigate these protections or examine alternatives to threat detection and analysis. To highlight the current and future importance of the associated concerns, we discussed the Flashback botnet's use of a similar technique to prevent the automated analysis of its samples.

7 Acknowledgements

The authors would like to thank the anonymous reviewers for their feedback. This material is based upon work supported in part by the National Science Foundation grant 0831300, the Department of Homeland Security contract FA8750-08-2-0141, and the Office of Naval Research grants N000140710907 and N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Department of Homeland Security, or the Office of Naval Research.

References

[1] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. Automatically identifying trigger-based behavior in malware. *Botnet Detection* (2008), 65–88.

[2] Caballero, J., Poosankam, P., Kreibich, C., and Song, D. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security* (2009), CCS '09.

[3] Chung, S., and Mok, A. Allergy attack against automatic signature generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection* (2006).

[4] Coogan, K., Lu, G., and Debray, S. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011).

[5] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and Communications Security* (2008).

[6] Dunn, A. M., Hofmann, O. S., Waters, B., and Witchel, E. Cloaking malware with the trusted platform module. In *Proceedings of the 20th USENIX conference on Security* (2011).

[7] Kirda, E., Kruegel, C., Banks, G., Vigna, G., and Kemmerer, R. Behavior-based spyware detection. In *Proceedings of the USENIX Security Symposium* (2006).

[8] Kolbitsch, C., Comparetti, P., Kruegel, C., Kirda, E., Zhou, X., and Wang, X. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium* (2009).

[9] Paleari, R., Martignoni, L., Roglia, G., and Bruschi, D. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX conference on Offensive technologies* (2009).

[10] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Computer Security Applications Conference (ACSAC)* (2006).

[11] Saito, Y. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (2005).

[12] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Impeding malware analysis using conditional code obfuscation. In *Network and Distributed System Security (NDSS)* (2008).

[13] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009).

[14] Vasudevan, A., and Yerraballi, R. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Security and Privacy, 2006 IEEE Symposium on* (2006).

[15] Yin, H., Liang, Z., and Song, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (2008).

[16] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security* (2007), CCS '07.