Code obfuscation

Abstract

Code obfuscation has a number of legitimate uses, including improved software security, tamper prevention, and intellectual property protection. Unfortunately, it also helps malware authors increase the survivability of their code and its ability to avoid detection. Combined with novel delivery methods, self-altering malware has rendered reverse-engineering exceedingly difficult and become an increasingly sophisticated challenge to intrusion detection systems (IDS) and anti-virus (AV) software.

As obfuscation becomes more complex it becomes harder to conceal and so detection and mitigation are not impossible. There is however a time-lag between the malware being deployed and IDS/AV companies issuing detection signatures. Modern IDS and AV software versions increasingly employ heuristic detection. There are therefore multiple layers of detection that must be avoided and a good defender will cover as many of these as possible.

This paper introduces some of the key concepts in code obfuscation and highlights some of the problems faced by IDS and AV software providers in defending against a constantly evolving threat. Also included is a general section on detection and mitigation.

Contents

Introduction

Code obfuscation is a set of program transformations that make program code and/or program execution difficult to analyse by hindering both manual and automated inspection.[1] There are a number of legitimate reasons why code is obfuscated;[2] to improve the security of the code through obscurity, to prevent tampering, or to protect intellectual property. Code obfuscation however, also presents a number of opportunities to maximize the potential for the survival of malware on an infected machine. On top of the various methods of obfuscation, the manipulation of encoding and decoding provide added layers of complexity which complicate detection. These methods have been used in combination with techniques like code fragmentation to malware of increasing sophistication and present a considerable challenge to IDS and AV software.

Legitimate reasons for code obfuscation

Obfuscated code contributes to heightened security by preventing code modifications or 'application hijacking' (the insertion of malicious code). In order to do this the code must be reverse-engineered and so obfuscated code offers increased security by obscuring its structure and functions.[3] Software developers may also employ obfuscation techniques to conceal flaws and vulnerabilities, or protect intellectual property. Code obfuscation also protects against malicious modifications to a program and software piracy because an attacker must first understand the software before they can make specified modifications.[4]

Methods of code obfuscation

Basic obfuscation commonly employs a simple mathematical function called 'exclusive OR operation' (XOR)[5]. The function is easy to implement and hides data from user inspection by using a constantly repeating key to encrypt the code. This is typically very easy to defeat because programs exist that systematically apply every possible single-byte XOR key in search of a particular string.[6] Other manual obfuscation techniques include renaming variables and functions, breaking down code structures (for delivery in separate data packets), or

[1] C. S. Collberg and C. D. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. volume 28, 2002.
[2] Techniques for Automating Obfuscation http://msdn.microsoft.com/en-us/library/hh977082(v=vs.107).aspx
[3] C. S. Collberg and C. D. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. volume 28, 2002.
[4] Code Obfuscation Techniques for Software Protection https://www.cosic.esat.kuleuven.be/publications/thesis-199.pdf
[5] Nowhere to Hide: Three methods of XOR obfuscation https://blog.malwarebytes.org/intelligence/2013/05/nowhere-to-hide-three-methods-of-xor-obfuscation/
[6] Obfuscation: Malware's best friend https://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/

translating the code through functions such as Hex, ROT, Base64 or a Caesar cipher. Often these translations are then incorporated into six basic obfuscation methods: [7]

- **Dead-code-insertion –** is the insertion of No Operation Performed (NOP) code; this code serves no function but is written in a way that complicates analysis
- **Subroutine reordering -** randomly changes the order of subroutines in the program, creating different malware signatures for every variation of subroutines
- **Code transposition –** changes the order of instructions by using statements which alters the code from its native form; this is achieved in two ways: by using unconditional branch statements, or by reordering the independent instructions, which is difficult to implement and harder to identify the malware
- **Instruction substitution –** replaces some of the code statements with the equivalent statements
- **Code integration –** inserts a new brief into the benign source code from a program in order to run the code malicious
- **Register reassignment –** replaces the unused registers with malware code registers is; the program code and its behaviour remains the same.

Most of the obfuscation discussed so far is still robust to memory dumps as they will still be calculated at run time and do not prevent static code analysis through normal reverse engineering techniques. This still presents a challenge for manual analysis, but automated processes are available to understand the concealed functionality. Often however, an entire program is obfuscated, preventing code analysis until it is placed in memory. This type of obfuscation is achieved with the use of a piece of software called a packer. Packers have legitimate purposes, some of which include reducing file sizes and protecting against piracy; they also help conceal vital program components and deter novice program crackers. [8] It compresses the original malware file and makes the original code and data unreadable, then, they will unpack to an executable before running in memory.

Heavily obfuscated code will implement both methods so even when unpacked the code is difficult to read, but could still be statically analysed. Once run, malware is decompressed in memory, revealing the program's original code. Packers ensure that the code can only be

[7] Basic survey on Malware Analysis, Tools and Techniques http://airccse.org/journal/ijcsa/papers/4114ijcsa10.pdf
[8] Obfuscation: Malware's best friend https://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/

analysed dynamically as it is being run. Malware authors prefer to create custom packers, decreasing the likelihood that IDS and AV software will detect it, and prevents malware from being reverse engineered. This approach often defeats modern unpacking scripts, and forces reversers to manually unpack the file and see what the program is doing when the malware variant is first deployed.[9] In addition to packers, a number of automatic code obfuscators are readily available such as the BitBoost Python Code Obfuscator[10] which can convert the same variable name or function parameter name into several different names within a piece of obfuscated code, without changing code functionality or increasing the number of code instructions.

Obfuscation examples

Below are a few simple examples of code obfuscation to show how the technique complicates malware detection which can be employed to conceal code.

Obfuscation Example	Explanation

`public class HelloWorld {`
 	
`  public static void main(String[] args) {`
`  System.out.println("Hello World!");`
`  }`
`}`
	Normal code for “Hello World!”

`public class HelloWorld {`
 	
`  public static void main(String[] args) {`
`  System.out.println("48656c6c6f20576f726c6421");`
`  }`
`}`
	Data Obfuscation with Hex
	Hex encoding turns “Hello World!” into 48656c6c6f20576f726c6421.

`public class HelloWorld {`
 	
`  public static void main(String[] args) {`
`System.out.println(“48”,”65en”,”6c”,”6c(fd”,”6f”,”2054”,”57g”,”6f5h”,”72__t”,”6c”,”64’h”,”21”`
`);`
`  }`
`}`
	Data fragmentation
	Adding “” around each digit and then packing it with other additional characters - in decoding only the first two bytes are read.

`public class HelloWorld {`
 	
`  cHVibGljIHN0YXRpYyB2b2lk main(String[] args) {`
`System.out.println(“48”,”65en”,”6c”,”6c(fd”,”6f”,”2054”,”57g”,”6f5h”,”72__t”,”6c”,”64’h”,”21”`
`);`
`  }`
`}`
	Code Obfuscation with Base64
	Using Base64 to encode ’public static void’ into cHVibGljIHN0YXRpYyB2b2lk hides the variables that determine how the “Hello World!” script is run.

[9] Obfuscation: Malware’s best friend https://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/
[10] BitBoost Python Code Obfuscator http://www.bitboost.com/python-obfuscator/

Malicious code obfuscation

All of these techniques can be used separately or in combination to obfuscate code, however for those scripting malware, code obfuscation greatly enhances two imperatives: the malware must evade detection, and survive long enough to complete its tasks. In order to do so it must:

- Ensure entry point obfuscation – hiding the initial security breach on the host machine or system by inserting code at an unlikely point in the infected file
- Resist manual and automated analysis – conceal suspicious signatures or behaviour
- Obfuscate the communication of instructions between the malware and the command and control server
- Ensure information exfiltration – hide what data has been compromised and where it was sent

Encryption and encoding/decoding issues

Fragmentation is simply deconstructing code for transmission in multiple packets and relies on the ability of the host to receive these, potentially in the wrong order, and reconstitute functioning code.[11] As the malware is broken down and transmitted in discrete packets at the TCP/IP layer, obfuscating the code can be used to evade IDS, and more commonly to hide specific signature strings. This fragmentation does not evade AV however, as packets are reassembled at the TCP/IP layer before being run on the host. It is therefore necessary to combine the obfuscation with other methods of concealment, such as encryption, the use of packers, or virtualization obfuscation. Although encryption is considered a distinct aspect of coding from obfuscation, malware encryption has been used to aid obfuscation[12] in addition to the manipulation of encoders and decoders which change as the code propagates creating malware that can be described as:

- **Oligomorphic -** where the decoder is changed for every instance of infection. It can still be detected by its signature, as there is a limit to the number of replications a decoder can make of itself.
- **Polymorphic -** an advancement on oligomorphic malware, this generates infinite number of decoders by using different obfuscation techniques. The basic function of polymorphic malware remains the same each time it is decoded, only the obfuscation

[11] Intrusion detection evasion: How Attackers get past the burglar alarm http://www.sans.org/reading-room/whitepapers/detection/intrusion-detection-evasion-attackers-burglar-alarm-1284
[12] A Brief History of Malware Obfuscation http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2/

changes. Depending on the conditions, polymorphic code also has the ability to re-write itself, further complicating detection.

- **Metamorphic -** re-written every time it is replicated, making each instance different from its previous once. This prevents detection by removing the potential for common signatures within a particular malware variant.

Virtualization obfuscation

During run-time, some malware authors make use of 'virtual machines' (VMs). It must be noted that in this context VMs refer to a virtual processor that executes virtual instructions (byte code) in an area of memory sectioned off from the rest of the host machine. This method was derived from digital rights management (DRM) schemes used to deter copy-writing of commercial media. While it is similar to runtime executable packers that decompress a file at runtime, exposing obfuscated code before it is executed, these VMs not only compress the target code, but also virtualize it. This renders analysis of its internal structure, if not impossible, extremely difficult to do.[13]

Because a VM translates portions of the malware's original code into a custom language (chosen at random when the malware is compiled and then interpreted at run-time) the malware is never restored to its original form.[14] This inhibits reverse-engineering for AV signatures and complicates real-time heuristic monitoring of the malware by IDS and AV software, because much of the code remains obfuscated in a custom language while running within a VM.

Detection and mitigation

Malicious code is difficult to detect when obfuscated and in the case of compromised code, has identical observable behaviour to the original. Often the functions appear legitimate to the user, although the program is performing entirely different functions. The calibre of obfuscation is therefore measured in three categories: potency (how well are the functions hidden from manual analysis), resilience (how well the obfuscation avoids automated

[13] Inside the Jaws of Trojan.Clampi
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/inside_trojan_clampi.pdf
[14] Unpacking Virtualization Obfuscators http://static.usenix.org/event/woot09/tech/full_papers/rolles.pdf

analysis), and cost (what resource overhead is added by including the obfuscated functions).[15] Although code obfuscation is a powerful tool in maximizing the survival of malware, its use may have an upside in detection as the presence of hash functions and encryption routines coupled with an unusual number of conditions utilizing them may indicate that suspicious code is a malware.[16] Mitigation measures focus on rule based checks that look for specific decoding commands during automated analysis. Malware programs often incorporate trigger-based behaviour to initiate routines based on specific conditions, and so advanced malware analysers are able to discover code guarded by triggers without triggering the malicious code.[17] Conducted in a 'sandbox' (a virtual machine used for running suspicious code in a tightly controlled environment), the malware can be triggered and analysed while isolated from a physical machine or network.

Obfuscated code stands out from normal code and so the more malware authors employ these techniques, the easier malware is to detect. As obfuscation becomes more complex, it requires more hashing or unusual processes, and monitoring the use of those unusual (or disabling them entirely) would further constrain obfuscated malware. Finally, whilst malware can be highly customised and complex to run on its own, as soon as it has to talk to another machine it needs to behave normally. Consequently there are multiple layers of detection that must be avoided and a good defender will cover as many as possible. While there are still signatures that detect the usage of some forms of obfuscation and encoding which are worth investigating, many of the methods you highlight are designed to defeat these.

Summary

Code obfuscation has a number of legitimate reasons that improve security, prevent tampering and protect intellectual property. It has also been used by malware authors to increase the survival of their code, and its ability to avoid detection. This has been achieved by employing a number of obfuscation methods in concert, complicating analysis of the code and increasing the difficulty of generating signatures or recognizable patterns of behaviour. Combined with novel delivery methods, self-altering malware has become an increasingly sophisticated challenge to IDS and AV software. The use of DRM schemes for virtualization

[15] A Taxonomy of Obfuscating Transformations https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf
[16] Impeding Malware Analysis Using Conditional Code Obfuscation http://www.iseclab.org/people/andrew/download/NDSS08.pdf
[17] ibid

obfuscation has rendered reverse-engineering exceedingly difficult, and continues the latest escalation in the malware arms race.

IDS and AV software has evolved in response to this and modern versions increasingly employ heuristic detection. This behavioural monitoring increases the chance that heavily obfuscated malware, which often require native functions such as hashing, encoding, and packers can be identified. It allows benign programs to be identified and white-listed, as their behaviour will change should malware compromise them. Modern AV software also often includes as standard virtual 'sandbox' tools that allow users to run suspicious programs, isolated from the operating system in order to prevent malicious code compromising the host machine or network. The trend towards a more holistic, behaviour, based protection looks set to continue as the proliferation of new malware variants renders signature based scanning inadequate.

www.cert.gov.uk

@CERT_UK

A CERT-UK PUBLICATION