Self Modifying Code

Malware Analysis and Reverse Engineering Proposal, Spring 2020

Jack Foltz, Andrew Park, Brian Qi

February 2020

1 Problem Statement

Self modifying code covers broad category of all types of code that are modified in some way, before running on the target platform. This definition includes code that modifies machine code directly or modifies code during runtime. Often, this technique is employed for optimization purposes: code can replace conditional branch statements with unconditional branch statements, or patch dynamic library loads. However, we are primarily intetrested in its security applications, in which code modifies itself at runtime in order to evade different methods of malware detection.

Basic malware typically has some sort of identifiable signature, such as a unique pattern of instructions present in the binary discovered through static analysis. By using self modifying code, static analysis can be rendered completely useless, since only a small bit of generic code that modifies itself is present in the binary. While traditional malware authors have used packing to encrypt the payload of a virus to evade detection, the unpacker itself must be left exposed. However, it is possible to even obsucure this unpacker part as well [1].

We seek to create a piece of software that modifies itself at runtime in order to evade such detection vectors, testing it out with both self made static analysis tools, and real world anti-malware software.

2 Motivation

Effective detection and mitigation of malware is an important security goal for modern operating systems and anti-malware software.

Understanding the methods by which malware can obscure itself through usage of self modifying code may lead to new insights into the characteristics of a good solution for detecting such malware, or how this technique could be even further enhanced.

References

[1] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, 2007.