Ferramenta de Geração Automática de Códigos Maliciosos Distribuídos

Victória Serra de Lima Moraes[1], **Paulo Lício de Geus**[1]

[1]Instituto de Computação (IC) – Universidade Estadual de Campinas (UNICAMP) – Campinas – SP – Brasil

victoriamoraes42@gmail.com, pgeus@unicamp.br

Resumo. *Códigos maliciosos tornam-se cada vez mais perigosos dia após dia, com novas arquiteturas ou melhores formas de processamento de dados. Surge, então, a necessidade de desenvolver ferramentas que impeçam a proliferação desses códigos. Com esse objetivo, é proposto o projeto e implementação de uma ferramenta de geração automática de códigos maliciosos distribuídos a fim de testar ferramentas de detecção de malware no contexto de computação de múltiplos núcleos.*

1. Introdução

Desenvolvedores de malware elaboram técnicas sofisticadas para burlar as ferramentas de detecção baseadas em assinatura existentes. Com a migração de CPUs de único núcleo para processadores multi-core, e a mudança de sistemas de 32 bits para sistemas de 64 bits, a realidade dos códigos maliciosos acompanha tais alterações. Assim, desenvolvedores de malware podem começar a usar essas informações para detectar se o código está sendo executado em uma solução *sandbox* ou em uma máquina real, pois a maioria das soluções de segurança ainda é de núcleo único. Ao verificar o número de núcleos disponíveis para determinada execução, o código malicioso pode evitar não apenas uma *sandbox*, mas também emuladores de antivírus, evitando assim a detecção.

Contudo, ferramentas de detecção de softwares nocivos não realizaram muitas atualizações que fossem capazes de detectar esse modo de ataque[Botacin et al. 2019]. Neste cenário, este trabalho propõe uma ferramenta de geração automática de malwares distribuídos a partir de amostras reais, para testar essas ferramentas e assegurar que os esforços feitos para reparar as falhas encontradas sejam definitivos, visto que não seria apenas um malware específico a ser detectado, mas inúmeros malwares gerados da mesma maneira.

2. Reescrita Binária

A reescrita binária pode ser dividida em quatro etapas[Wenzl et al. 2019] – análise sintática, análise, transformação, e geração de código. Executáveis consistem em dados administrativos e de *payload*. O foco da reescrita é obter e manipular os dados de *payload*. No entanto, essa informação geralmente se encontra dispersa ao longo do arquivo; instruções (em arquiteturas CISC) e variáveis não são bem delimitadas; binários não possuem informações de tipo de variável; e o tipo do endereço deve ser recuperado separadamente. Assim, o propósito da primeira etapa é obter o fluxo bruto de instruções e passá-lo ao *disassembler*.

A segunda etapa recupera a estrutura do código fonte do programa. Primeiramente, o fluxo bruto binário é avaliado a fim de que um grafo de controle de fluxo seja gerado. Então, algoritmos de recuperação de funções têm a tarefa de achar e agrupar séries de instruções conectadas por condições a blocos de funções, assim como determinar os pontos de entrada e saída da função. Após obtidas essas informações, o binário poderá ser alterado em pontos de instrumentação. Esses pontos são definidos como locais especificados pelo usuário onde o fluxo de controle muda; ou mudanças de instruções podem ser aplicadas. Finalmente, as mudanças planejadas são integradas ao binário de tal forma que ele se mantenha executável.

3. Metodologia

A princípio, construímos, estaticamente, um grafo de dependências dos binários, de modo a entender quais partes podem ser distribuídas. O próximo passo é realizar uma análise de dados do programa, colocando-o na forma de atribuição estática única (SSA)[Pradelle et al. 2011]. Isso fornece um número de versão exclusivo para cada definição de registro e fornece um link direto para cada uso de registro.

Partimos, então, para a reescrita do binário. A aplicação é carregada em um programa similar a um *debugger*, com o intuito de monitorar cada instrução executada e variável acessada. Isso pode ser realizado com a API `PTRACE` (Linux) ou a API de aplicação de *debug* (Windows). Durante a execução, o binário é "desmontado"(*disassembled*) ao longo dos caminhos cobertos por seus dados de entrada, obtendo, assim, as estruturas de interesse.

Com essas estruturas, obtemos um código com todos os acessos à memória executados no código binário, mas onde a semântica está oculta. Finalmente, com esse código e o grafo de dependências previamente obtido, podemos realizar a divisão do malware em múltiplos núcleos com o intuito de evitar detecção.

4. Conclusões

A ferramenta proposta estende o trabalho realizado por [Botacin et al. 2019], buscando um sistema de geração automática de códigos maliciosos distribuídos paralelizados de diferentes formas a fim de avaliar diferentes ferramentas em contextos variados. Ao fim, será gerado um possível alerta destinado aos fabricantes de softwares de detecção de malware para que haja uma atualização de suas ferramentas.

Referências

Botacin, M., de Geus, P. L., and Grégio, A. (2019). "VANILLA" malware: vanishing antiviruses by interleaving layers and layers of attacks. *Journal of Computer Virology and Hacking Techniques*.

Pradelle, B., Ketterlin, A., and Clauss, P. (2011). Transparent Parallelization of Binary Code. In *First International Workshop on Polyhedral Compilation Techniques, IMPACT 2011, in conjunction with CGO 2011*, Chamonix, France. Christophe Alias, Cédric Bastoul.

Wenzl, M., Merzdovnik, G., Ullrich, J., and Weippl, E. (2019). From hack to elaborate technique—a survey on binary rewriting. *ACM Comput. Surv.*, 52(3):49:1–49:37.