Identificação de código suspeito por meio de depuração supervisionada

Davidson R. Boccardo[1], Raphael C. S. Machado[1], Vagner P. da Silva[1], Luiz Fernando Rust da Costa Carmo[1]

[1]Instituto Nacional de Metrologia, Qualidade e Tecnologia (INMETRO)
Duque de Caxias – RJ – Brasil

{drboccardo,rcmachado,vpraia,lfrust}@inmetro.gov.br

Abstract. *The identification of hidden features in an embedded software by analyzing its binary code is a challenging task. This challenge is due to the inherent complexity of binary code, which hinders readability and understanding of all the features that the software may contain. This fact is particularly important in the area of information security, as hidden features can compromise the proper behavior of the software. In the proposed work we propose a method for locating suspect code snippets that may contain hidden features by debugging the embedded software in a supervised manner, assisted with the reconstruction of control flow graphs and call graph.*

Resumo. *A identificação de funcionalidades ocultas pela análise do binário de um software embarcado é uma tarefa desafiadora. Este desafio se deve ao fato da própria complexidade em se analisar um binário. É de particular interesse da área da segurança da informação garantir a inexistência de tais funcionalidades uma vez que elas podem comprometer o comportamento de um software. Este trabalho propõe um método para localizar trechos de código suspeitos por meio de uma depuração supervisionada do software embarcado, auxiliada pela análise estática dos grafos de fluxo de controle e de chamadas.*

1. Introdução

Com o avanço da tecnologia, os softwares passaram a ser uma constante em diversas áreas da sociedade. Este fato pode ser observado pelo grande número de sistemas embarcados, atualmente utilizados para diversos fins. Nesse cenário é observada a possibilidade de inúmeras novas formas de falhas e fraudes decorrentes de características inerentes aos softwares e aos dispositivos que o executam, o que introduz uma série de preocupações relacionadas à confiabilidade desses sistemas.

Essas preocupações se mostram especialmente importantes em áreas onde a tolerância a falhas é pequena ou onde o interesse por fraudes é grande. Por exemplo, em softwares utilizados na área médica, pequenas falhas podem ter consequências drásticas. Outro exemplo, na área automotiva, uma pessoa mal-intencionada pode querer reduzir a quilometragem do veículo para obter um maior valor de revenda. Outro caso seria o do proprietário de uma balança, o qual pode modificar o software de tal forma para que as novas medições (erradas) possam beneficiá-lo.

Este trabalho propõe um método para identificar trechos de código que podem ter uma funcionalidade escondida, reduzindo assim a complexidade de uma avaliação

de segurança de software. Para isso o método emprega técnicas de análise estática para mapear todo o conjunto de instruções contidos no executável e de análise dinâmica para executar o código e mapear as instruções realmente executadas durante o funcionamento do dispositivo. A diferença entre o conjunto completo de instruções obtido na análise estática e o conjunto de instruções executado durante a análise dinâmica resulta em um conjunto de instruções, que será criteriosamente avaliado a fim de verificar a presença de códigos suspeitos.

2. Proposta para identificação de código suspeito

No proposto trabalho apresentamos um método de análise de software para identificação de trechos de código suspeitos. O método parte do princípio que funcionalidades não-declaradas são suspeitas, e necessitam de uma análise mais cuidadosa. Assim, o método realiza a identificação por meio de análises estática e dinâmica. Na etapa estática, grafos de fluxo de controle e o grafo de chamadas são analisados; e na etapa dinâmica, chamada de depuração supervisionada, permite identificar as instruções que são realmente executadas pelas funcionalidades declaradas. As instruções não executadas são classificadas como suspeitas, por não terem seus propósitos documentados.

A identificação do código suspeito auxilia no processo de validação de software [Dowd et al. 2006], ao passo que reduz a área de busca por possíveis funcionalidades escondidas que o software possa ter, podendo até mesmo eliminar esta busca no caso em que todas as instruções encontradas no código binário são executadas durante a depuração supervisionada. Caso, existam instruções não executadas, estas serão apuradas a fim de verificar se comprometem a segurança ou confiabilidade do software, ou se são instruções pertencentes a bibliotecas ou interrupções não utilizadas.

O método possui o seguinte fluxo: a partir da obtenção do conjunto de instruções da imagem do código binário (*disassembly*) é iniciada a etapa de análise estática. Esta etapa consiste na análise dos grafos de fluxo de controle individualizados para cada função e do grafo de chamadas. Após a etapa de análise estática, inicia-se a etapa de análise dinâmica chamada de depuração supervisionada, na qual é realizada a instrumentação do código através de pontos de parada (*breakpoints)* em pontos específicos, determinados durante a etapa de análise estática, para obter o conjunto de instruções executadas pelas funcionalidades declaradas. A última etapa consiste na identificação de código suspeito na qual é realizada a diferença entre as instruções extraídas da imagem do código binário e as instruções executadas durante a depuração supervisionada. A Figura 1 representa graficamente esse fluxo.

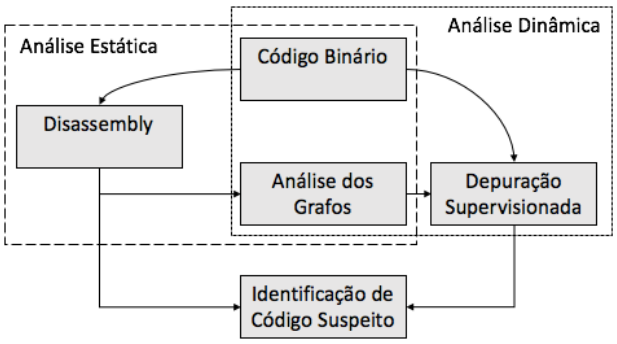

Figura 1. Fluxo de trabalho para identificação de código suspeito.

Etapa de análise estática. A partir do *disassembly* do binário, são gerados o grafo de chamadas e os grafos de controle de fluxo para cada uma das funções [Collberg and Nagra 2009]. O grafo de chamadas representa o relacionamento entre as funções de um programa. Os nós deste grafo são as funções do programa e as arestas representam as chamadas realizadas. Uma aresta de F1 → F2 de um nó F1 para um nó F2 indica que a função F1 possui uma instrução de chamada cujo destino é a função F2. O grafo de fluxo de controle representa a computação intra-procedural de uma função. Os nós de um grafo de fluxo de controle são chamados de blocos básicos, que contêm um fluxo contínuo de instruções e, possivelmente, terminam em uma instrução de salto. O fluxo de controle sempre inicia no começo de um bloco, executa todas as instruções daquele bloco e saí no final do bloco. Uma aresta de A → B da entrada de um bloco A para um bloco B indica que o fluxo de controle poderia, possivelmente, fluir do bloco A para o bloco B durante a execução do programa. Nesta etapa, os endereços das instruções iniciais das funções são armazenados para a etapa de análise dinâmica.

Etapa de análise dinâmica. O método de depuração supervisionada visa identificar quais instruções são executadas durante a execução do software, tendo como base a documentação detalhada das funcionalidades do código.

O método estabelece pontos de parada (*breakpoints*) nos endereços das instruções iniciais obtidas pelo grafo de chamadas. Quando o primeiro ponto de parada é atingido, o método adiciona novos pontos de parada nas instruções de transferência de controle (salto) e no endereço da instrução final do bloco básico correspondente, na qual o bloco é obtido a partir do grafo de fluxo de controle daquela função. O método vai então removendo e adicionando novos pontos de parada (devido a limitação do número de *breakpoints*) de acordo com a execução da aplicação de software. Após um limiar de instruções executadas ou de um tempo pré-definidos, os endereços das instruções executadas são armazenados, e serão utilizados para realizar a correspondência das instruções obtidas na etapa estática com as da etapa dinâmica.

Identificação de código suspeito. Possíveis trechos de código suspeitos são identificados pela diferença entre as instruções obtidas no *disassembly* do binário e as instruções executadas durante a depuração supervisionada. Para redução do número de falsos positivos, considera-se a utilização de assinaturas de bibliotecas externas e de instruções inseridas pelo ambiente de compilação para fins de identificação e remoção dessas instruções do conjunto de código suspeito.

3. Experimentos

Para validar o método, um protótipo que reproduz o funcionamento de um taxímetro foi desenvolvido, contendo funcionalidades para iniciar, parar e zerar o contador do valor de uma viagem, "alterar a bandeira" e um *backdoor* malicioso que adiciona 10% ao valor do contador. O software foi embarcado em uma placa de desenvolvimento NXP ARM7-LPC 2368 com interface JTAG (padrão IEEE), utilizada para fins de gravação e depuração, e disponível em grande parte dos dispositivos.

A extração do código binário do software embarcado é feita pelo dispositivo J-Link V8 e do software J-Link Commander. O J-Link V8 é um dispositivo para conexão com o sistema embarcado por meio da interface JTAG. O J-Link Commander é um software de interface de usuário que faz parte do kit do J-Link V8.

Os processos de *disassembly* do código, geração do grafo de chamadas e dos grafos de fluxo de controle foram realizados pela ferramenta IDA Pro. O IDA Pro é um software de engenharia reversa que contém *disassembler* e depurador para dezenas de arquiteturas, e oferece diversos recursos para análise de software. Por meio do *disassembly* foram encontradas 2945 instruções. Após esta etapa, foram localizadas as instruções de transferência de fluxo de controle a fim de estabelecer os pontos de parada durante a depuração supervisionada. No caso da arquitetura ARM, utilizada neste trabalho, as instruções de desvio são as instruções de desvio (B, BL, BX, BLX e BXJ).

A depuração supervisionada foi realizada por meio do J-Link V8, J-Link GDB Server e GDB. O J-Link GDB Server é um software que funciona como um servidor de depuração para o J-Link V8, e o GDB é um software de depuração, utilizado para enviar comandos de depuração para o J-Link GDB Server.

Foi pré-estabelecido um limiar de 5 mil instruções para a etapa de análise dinâmica, na qual 890 instruções foram executadas diante de todas as funcionalidades declaradas. As 2055 (2945-890) instruções remanescentes podem ser enquadradas como códigos para tratamento de erro, bibliotecas do próprio ambiente de compilação, bibliotecas externas, ou até mesmo funcionalidades escondidas, necessitando de uma avaliação mais criteriosa. Entre estas instruções está contido o *backdoor* malicioso que adiciona 10% ao valor do contador.

4. Considerações finais

Neste trabalho, propomos um método de análise de software para a identificação de possíveis funcionalidades escondidas, por meio de depuração supervisionada dos grafos estáticos do software. O resultado é obtido pela identificação das instruções contidas no código binário, mas que durante a execução das funcionalidades declaradas do software não são executadas. Resultados preliminares de validação mostram que o método é eficaz para identificar as instruções legítimas, que pertencem as funcionalidades declaradas, e as instruções não executadas, que incluem a função maliciosa de alterar um taxímetro, reduzindo a complexidade de análise durante uma avaliação.

O emprego deste método de validação de software depende da análise da documentação do software, na qual especifica todas suas funcionalidades, possibilitando a reprodução das funcionalidades e a verificação de trechos de código que possam conter funcionalidades ocultas. Este cenário é extremamente comum para órgãos regulatórios que demandam a abertura de detalhes de implementação de um software, como exemplo, a abertura de código-fonte como requisito para avaliação de medidores inteligentes. Como trabalhos futuros, integraremos ao método uma forma de identificar instruções de bibliotecas e do ambiente de compilação para refinar os resultados.

Referências

Dowd, M., McDonald, J., and Schuh, J. (2006). The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. *Addison-Wesley Professional*.

Collberg, C. and Nagra, J. (2009). Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. *Addison-Wesley Professional*, 1st edition.

