Detecção Estática e Consistente de Potenciais Estouros de Arranjos

Bruno Rodrigues Silva[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

brunors@dcc.ufmg.br

Resumo. *Estouros de arranjos, uma vulnerabilidade de software bastante conhecida na literatura especializada em segurança, são frequentemente utilizados por adversários que têm o objetivo de corromper o fluxo de controle do programa. Isto é possível em linguagens fracamente tipadas como C e C++ onde os acessos à arranjos não são verificados. Uma possível solução é a inserção de código para verificação de todos os acessos à arranjos de forma a evitar aqueles fora dos limites. Entretanto o custo em tempo de execução seria proibitivo. Este trabalho propõe a detecção de potenciais estouros de arranjos via análise estática de código, o que permitiria ao desenvolvedor, a inserção de código de verificação apenas nos traços estáticos possivelmente vulneráveis. Verificou-se que cerca de 41% do código fonte dos benchmarks SPEC CPUINT 2006 estão vulneráveis à este tipo de ataque.*

1. Introdução

Um estouro de arranjo acontece quando este é preenchido com dados que ultrapassam os seus limites. Isso pode acontecer de forma proposital ou não, principalmente em linguagens fracamente tipadas e bastante utilizadas como C e C++ que não fazem verificação dos acessos ao arranjo. O que possibilita a existência de uma grande quantidade de *worms* e vírus que contaminaram e contaminam milhões de dispositivos computacionais em todo o mundo. O estouro de arranjo pode sobreescrever os valores de variáveis locais e endereço de retorno de função, o que compromete todo o fluxo de dados e controle.

Usuários maliciosos de posse do código fonte podem manipular os dados de entrada que são públicos a fim de estourar um arranjo e sobreescrever o endereço de retorno de uma função com informações que direcionem o fluxo de controle para funções de sistema tais como *telnet* e *shell* com os mesmos privilégios de sistema do programa atacado. Este redirecionamento pode causar desde a interrupção do serviço até a obtenção do controle total do sistema.

Uma solução empregada por muitos compiladores é a inserção de canários, que são valores aleatórios inseridos antes do valor de retorno de uma função. Além disso, é inserido uma pequena seção de código de verificação antes do retorno, capaz de gerar uma exceção e encerrar o programa caso o canário tenha sido modificado. Portanto, caso um adversário tente sobreescrever o valor de retorno, inevitavelmente ele também sobrescreverá o canário que por sua vez será detectado no momento do retorno da função.

Entretanto, mesmo funções protegidas por canários, estão vulneráveis ao ataque de estouro de arranjo [Maffra et al. 2013]. Isto acontece porque algumas variáveis locais

podem ser alojadas na pilha da função após o espaço alocado para o arranjo e portanto, elas podem ser sobreescritas em um possível estouro. Um adversário pode, através de um estudo cauteloso do fluxo de dados/controle da função, escolher quais valores tais variáveis receberão e assim comprometer a execução de todo o programa.

O trabalho de Quadros *et. al*. [Quadros et al. 2012] demonstra como um adversário pode assumir o controle total de um sistema executando Ubuntu Linux, por meio de um ataque de estouro de arranjo. Em [Maffra et al. 2013] Maffra *et al.* propõem uma análise estática de código para a detecção de vulnerabilidade de ataque por estouro de arranjo em código compilado com canários. Entretanto, por não considerar as dependências de controle entre as variáveis e os predicados de instruções de desvio, definidas na próxima Seção, sua análise não é consistente. Uma análise é considerada consistente se ela não gera falsos negativos, isto é, a análise não pode reportar a inexistência de vulnerabilidade, quando na verdade ela existe.

Nesse trabalho, propõe-se uma análise estática de código capaz de detectar de forma consistente os possíveis traços de código vulneráveis ao ataque de estouro de arranjo. Tal análise foi implementada como um módulo para o compilador LLVM [Lattner and Adve 2004] e executada sobre o conjunto de benchmarks SPEC CPU INT 2006[1].

2. Análise Estática

Um grafo de dependências $G = (V, E)$ tal como proposto por Ferrante [Ferrante et al. 1987] é utilizado na implementação dessa análise estática para detecção consistente de vulnerabilidade de estouro de arranjo. O conjunto V de vértices contém variáveis mapeadas em registradores, conjuntos de posições de memória[2] e operações. Já o conjunto E representa as relações de dependências de dados e de controle entre os vértices contidos em V. Para cada variável u definida com a informação contida em outra variável v, temos uma aresta direcional conectando v à u, capturando assim a relação de dependência de dados entre essas duas variáveis.

Por outro lado, informação também flui implicitamente de um predicado p, que controla um teste condicional, para toda variável atribuída no escopo desse teste. Por exemplo, a sequência `p = (a > b); v = p?0:1;` determina uma dependência de controle de p para v e portanto uma aresta direcional conectando p à v também é inserida no conjunto E do grafo de dependências. Não considerar os fluxos implícitos em uma análise de fluxo de informação é um erro, conforme descrito em Silva *et. al* [Silva 2013].

O algoritmo para construção do grafo de dependências pode ser encontrado em Silva [Silva 2013] e uma versão linear sobre o número de variáveis do programa pode se encontrada em [Silva 2014]. Portanto, por limitações de espaço, ele não será descrito nesse texto. Uma vez que o grafo de dependência tenha sido construído, é correto afirmar que todas as relações de dependência de controle e de dados entre as variáveis e posições de memória do programa estão devidamente presentes nesta nova representação intermediária. Tal representação é então utilizada na busca por traços estáticos de código que possibilitem estouro de arranjo, isto é, caminhos contaminados no fluxo de informação.

[1]Código fonte disponível na página do projeto E-CoSoC - https://code.google.com/p/ecosoc/

[2]Uma análise de ponteiros é utilizada na construção de tais conjuntos

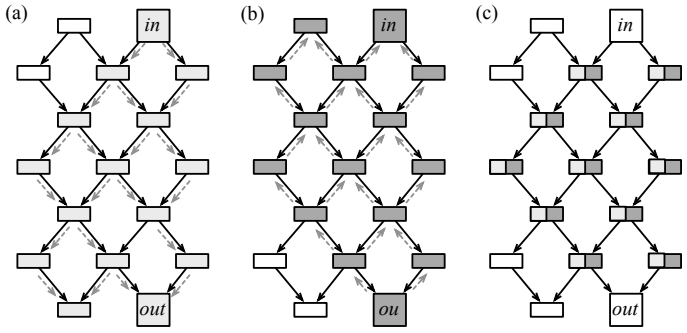

Figura 1. a) Busca em profundidade a partir de um vértice do conjunto de entradas. b) Busca em profundidade invertida à partir de um vértice do conjunto de sorvedouros. c) Vértices marcados em ambas as buscas

A busca por caminhos contaminados é baseada na localização de arranjos que são dependentes de entradas que um adversário pode manipular. Para tanto, é definido um conjunto G_e de entradas públicas que, seguindo o estudo realizado por Mafra *et. al* [Maffra et al. 2013], será constituído pelos seguintes elementos:

1. os argumentos do método *main*, isto é, as variáveis *argc* e *argv*;
2. o resultado retornado por funções externas de entrada tais como *scanf*, *fgets*, *read*;
3. ponteiros passados como argumento de funções externas;

Um conjunto de sorvedouros G_s que contém todos as ponteiros para início de arranjos locais também é construído. Em seguida é realizada uma busca em profundidade a partir de cada vértice do conjunto G_e, onde cada vértice alcançável é marcado. Uma segunda busca é realizada a partir de cada vértice do conjunto G_s seguindo o sentido contrário das arestas. Novamente todo os vértices alcançáveis por cada sorvedouro são marcados. A Figura 1a) esboça a busca e marcação de vértices a partir de um vértice de entrada. A busca invertida a partir de um sorvedouro pode ser visualizada na Figura 1b). Finalmente a Figura 1c) exibe os vértices presentes na interseção das duas buscas. Estes são, portanto, os possíveis caminhos que conectam uma entrada pública que pode ser manipulada pelo adversário, à uma acesso de arranjo que pode ser estourado e consequentemente deve ser sanitizado ou evitado pelo desenvolvedor.

3. Resultados

A análise estática descrita na seção anterior foi implementada como um módulo para o compilador LLVM e executada sobre o conjunto de *benchmarks SPEC CPU INT 2006*. A linha de base utilizada para comparação do resultado foi a solução proposta por Mafra [Maffra et al. 2013], que não leva em consideração as dependências de controle e portanto não pode ser considerada consistente. A Figura 2 mostra o número de vértices contaminados em cada *benchmark*. Um vértice está contaminado se ele faz parte de um caminho que leva um vértice do conjunto G_e à um vértice do conjunto G_s. Claramente o número de vértices em caminhos vulneráveis encontrados pela análise de linha de base é significativamente inferior ao número encontrado pela análise consistente proposta neste trabalho. Isso se justifica porque a linha de base não considera os fluxos implícitos de informação determinados pelas dependências de controle. Além disso, a análise consistente revelou que do total de 3,559,715 vértices, 1,475,574 estão inseridos em caminhos vulneráveis. Isto é, 41% do código fonte da coleção de *benchmarks*.

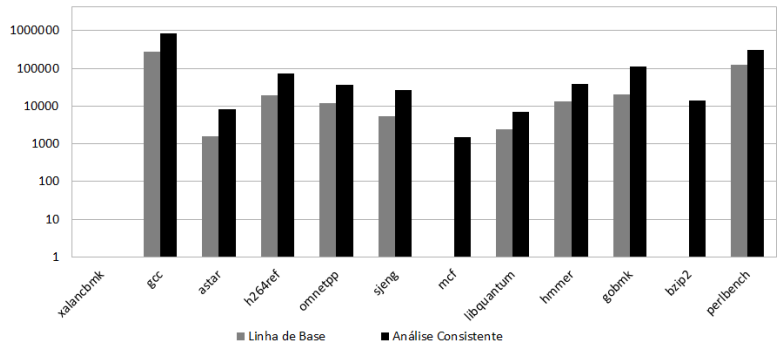

Figura 2. Comparação entre o número de vértices vulneráveis da solução de linha de base e da análise estática consistente.

4. Conclusão e Trabalhos Futuros

Está claro que desconsiderar os fluxos implícitos em qualquer análise de fluxo de informação é um erro que pode custar a consistência da análise. Este trabalho mostrou que isso é particularmente verdade para o problema de estouro de arranjos. Como trabalho futuro pretende-se criar um instrumentador de código. Assim, as aplicações serão analisadas e aqueles traços de instruções em caminhos detectados como vulneráveis, serão instrumentados com novas instruções capazes de impedir um estouro de arranjo. Além disso, busca-se utilizar a análise estática aqui proposta, na solução de outros importantes problemas de segurança computacional, tais como o vazamento de informação por canais laterais em sistemas criptográficos [Kocher 1996] .

Referências

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK. Springer-Verlag.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Maffra, I. K. T., Pereira, F. M. Q., and Oliveira, L. B. (2013). Detecção automática de vulnerabilidades em código protegido por canários. In *SBSeg*, pages 184 –197.

Quadros, G. S., Souza, R. M., and Pereira, F. M. Q. (2012). Dynamic detection of address leaks. In *SBSeg*, pages 61–75.

Silva, B. R. (2014). Um algoritmo linear para a construção de program slices. In *Anais do XVIII Simpósio Brasileiro Linguagens de Programação*.

Silva, Bruno Rodrigues, P. F. M. Q. O. L. B. (2013). Uma representação intermendiária para a detecção de vazamentos implícitos de informação. In *Anais do XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 212–225.

