Prevenção de Ataques de Não-Terminação baseados em Estouros de Precisão

Raphael Ernani Rodrigues and Fernando Magno Quintão Pereira

Departamento de Ciência da Computação – UFMG – Brasil
{raphael,fernando}@dcc.ufmg.br

Resumo Dizemos que um programa é vulnerável a um ataque de não terminação quando um adversário pode lhe fornecer valores de entrada que façam algum de seus laços iterar para sempre. A prevenção de ataques desse tipo é difícil, pois eles não se originam de bugs que infringem a semântica da linguagem em que o programa foi feito. Ao contrário, essas vulnerabilidades têm origem na aritmética modular inteira de linguagens como C, C++ e Java, a qual possui semântica bem definida. Neste artigo nós apresentamos uma ferramenta que detecta tais problemas, e que saneia código vulnerável. A detecção da vulnerabilidade dá-se via uma análise de fluxo de informação; a sua cura decorre de guardas que nosso compilador insere no código vulnerável. Nós implementamos esse arcabouço em LLVM, um compilador de qualidade industrial, e testa-mo-no em um conjunto de programas que compraz mais de 2.5 milhões de linhas de código escrito em C. Descobrimos que, em média, caminhos em que informação perigosa trafega são pequenos, sendo compostos por não mais que 10 instruções assembly. A instrumentação que inserimos para prevenir ataques de não terminação aumenta o tamanho do programa saneado em cerca de 5% em média, e torna-os menos que 2% mais lentos.

Resumo We say that a program is vulnerable to a non-termination attack if (i) it contains a loop that is bounded by values coming from public inputs, and (ii) an adversary can manipulate these values to force this loop to iterate forever. Preventing this kind of attack is difficult because they do not originate from bugs that break the semantics of the programming language, such as buffer overflows. Instead, they usually are made possible by the wrapping integer arithmetics used by C, C++ and Java-like languages, which have well-defined semantics. In this paper we present the diagnosis and the cure for this type of attack. Firstly, we describe a tainted-flow analysis that detects non-termination vulnerabilities. Secondly, we provide a compiler transformation that inserts arithmetic guards on loop conditions that may not terminate due to integer overflows. We have implemented our framework in the LLVM compiler, and have tested it on a benchmark suite containing over 2.5 million lines of C code. We have found out that the typical path from inputs to loop conditions is, on the average, less than 10 instructions long. Our instrumentation that prevents this kind of attacks adds less than 5% extra code on the secured program. The final, protected code, is, on the average, less than 2% slower than the original, unprotected program.

1 Introdução

Um ataque de Negação de Serviços (*Denial-of-Service – DoS*) consiste em sobrecarregar um servidor com uma quantidade de falsas requisições grande o suficiente para lhe comprometer a capacidade de atender contatos legítimos. Existem hoje diversas maneiras diferentes de detectar e reduzir a efetividade desse tipo de ataque [**?**]. Neste artigo, contudo, descreveremos uma forma de negação de serviço que é de difícil detecção: *os ataques de não-terminação*. Um adversário realiza um ataque desse tipo fornecendo ao programa alvo entradas cuidadosamente produzidas para forçar iterações eternas sobre um laço vulnerável. Um ataque de não terminação demanda conhecimento do código fonte do sistema a ser abordado. Não obstante tal limitação, esse tipo de ataque pode ser muito efetivo, pois bastam algumas requisições para comprometer o sistema alvo. Uma vez que essa quantidade de incursões é pequena, os métodos tradicionais de detecção de negação de serviço não podem ser usados para reconhecer ataques de não-terminação. Além disso, dada a vasta quantidade de código aberto usado nos mais diversos ramos da indústria de *software*, usuários maliciosos têm a sua disposição um vasto campo de ação.

A detecção de código vulnerável a ataques de não-terminação é difícil. Tal dificuldade existe, sobretudo, porque esse tipo de ataque não decorre de deficiências de tipagem fraca, normalmente presentes em programas escritos em C ou C++. Programas escritos em linguagens fortemente tipadas, como Java, por exemplo, também apresentam a principal fonte de vulnerabilidades a ataques de não terminação: a *aritmética modular inteira*. Em outras palavras, uma operação como `a + 1`, em Java, C ou C++, pode resultar em um valor menor que aquele inicialmente encontrado na variável `a`. Esse fenômeno ocorrerá quando a variável `a` guardar o maior inteiro presente em cada uma dessas linguagens. Nesse caso, ao fim da operação, `a + 1` representará o menor inteiro possível em complemento de dois. Em outras palavras, um laço como `for (i = 0; i <= N; i++)` nunca terminará se `N` for `MAX_INT`, o maior inteiro da linguagem.

Este artigo traz duas contribuições relacionadas a ataques de não-terminação. Em primeiro lugar, ele descreve uma técnica que descobre vulnerabilidades relacionadas a esse tipo de ataque. Em segundo lugar, o artigo mostra como código pode ser protegido contra tais ataques. A nossa técnica de detecção de vulnerabilidades é baseada em análise de fluxo contaminado. Tal análise é parte do arcabouço teórico de rastreamento de informação inicialmente proposto por Denning e Denning nos anos setenta [**?**]. Um ataque de fluxo contaminado pode ser efetivo somente em programas cujas operações críticas dependam de dados de entrada. Em nosso contexto, uma operação crítica é o teste de controle de laço. Em conjunto com o algoritmo de detecção de vulnerabilidades, nós propomos também uma técnica para sanear programas contra ataques de não-terminação. Nossa estratégia consiste na inserção de verificações sobre as operações aritméticas realizadas pelo programa alvo. Essas verificações ocorrem durante a execução do programa, e invocam código de segurança sempre que estouros de precisão em variáveis inteira são percebidos. Nós instrumentamos somente código que controla o número de iterações em laços. Consequentemente, o

arcabouço que propomos incorre em uma perda de desempenho muito pequena, e, em nossa opinião, completamente justificável em decurso do benefício que assegura.

Nós implementamos todas as idéias que discutimos neste artigo em LLVM, um compilador de qualidade industrial [**?**]. Na seção **??** descreveremos uma série de experimentos que validam nossa análise. Ao analisar os programas presentes na coleção SPEC CPU 2006, fomos capazes de descobrir XX laços que são influenciados por dados provenientes de entradas públicas, isto é, que podem ser manipuladas por um adversário. Dentre esses laços, pelo menos XX estão sujeitos à não terminação. Para obter esse número, utilizamos um padrão muito simples: procuramos por laços cujo teste de parada é do tipo `i <= N`, sendo `N` dependente da entrada. Uma vez que nos atemos a esse tipo de condição de parada, especulamos que a quantidade de laços vulneráveis presente nos programas de SPEC CPU 2006 seja bem maior que o valor que apuramos. A nossa instrumentação – usada para impedir os ataques de não-terminação – mostra-se extremamente eficiente. Os testes que inserimos antes de cada operação aritmética que pode levar a um ataque de não terminação custa-nos uma perda de desempenho de menos que XX%.

2 Ataques de Não-Terminação

De acordo com Appel e Palsberg [**?**, p.376], um laço natural é um conjunto de nodos S do grafo de fluxo de controle (CFG) de um programa, includindo um nodo cabeçalho H, com as seguintes três propriedades:

- a partir de qualquer nodo em S existe um caminho que chega a H;
- existe um caminho de H até qualquer nodo que faz parte de S;
- qualquer caminho de um nodo fora de S para um nodo em S contém H.

A *condição de parada* de um laço é uma expressão booleana $E = f(e_1, e_2, \ldots, e_n)$, sendo cada $e_j, 1 \leq j \leq n$ um valor que contribui para a computação de E. Seja P um programa que possui um laço L limitado por uma condição de parada $E = f(e_1, e_2, \ldots, e_n)$. Dizemos que P é vulnerável a um ataque de não terminação quando as duas condições abaixo são verdadeiras sobre ele:

1. existe um subconjunto $E' \subseteq \{e_1, e_2, \ldots, e_n\}$ de valores que dependem de um conjunto $I = \{i_1, i_2, \ldots, i_m\}$ de dados lidos a partir da entrada do programa.
2. existe uma atribuição de valores $i_1 \mapsto v_1, i_2 \mapsto v_2, \ldots, i_m \mapsto v_m$ que, ao influenciar E', faz com que o laço L não termine.

Note que a nossa definição de vulnerabilidade de não-terminação requer a noção de dependência de dados. Se um programa P possui uma instrução que usa a variável u e define a variável v, então v *depende* de u. Dependências de dados são transitivas, e embora possam ser cíclicas, não são necessariamente comutativas.

Ilustraremos ataques de não terminação via o programa mostrado na Figura **??**(a). Esse programa calcula o fatorial de um número inteiro na linguagem C. O padrão que rege essa linguagem de programação não determina

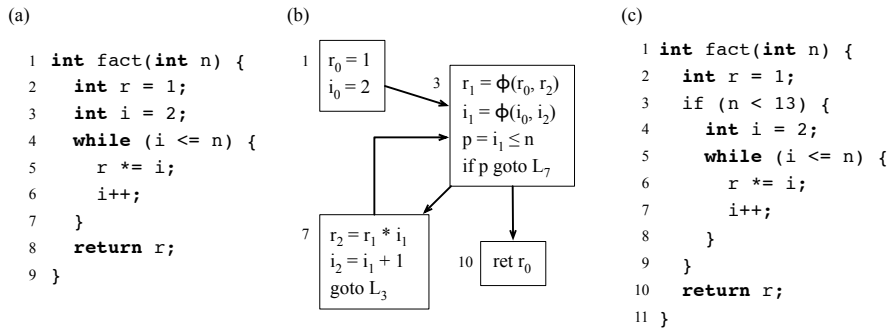

Figura 1. (a) Uma função em C, que calcula o fatorial de um número inteiro. (b) O CFG da função `fact`. (c) Exemplo de laço cuja condição de parada depende de valores de entrada mas que sempre termina.

o tamanho do tipo `int`. Essa informação depende da implementação do compilador usado. Entretanto, é usual que inteiros sejam representados como números de 32 bits em complemento de dois. Nesse caso, o maior inteiro representável é $MAX\_INT = 2^{31} - 1 = 2,147,483,647$. Se o parâmetro `n` for igual a MAX_INT, então a condição da linha 4 sempre será verdadeira, e o laço nunca terminará. A não-terminação ocorre porque quando `i` finalmente chega a MAX_INT, a soma $i + 1$ nos retorna o menor inteiro possível, isto é, -2^{31}.

O programa da figura **??**(a) é vulnerável a ataques de não-terminação. Para explicitar tal fato, a figura **??**(b) mostra o grafo de fluxo de controle do programa. Esse CFG está convertido para o formato de atribuição estática única (SSA) [**?**]. Usaremos essa representação de programas porque ela facilita a nossa análise de dependência de dados. Os blocos básicos que começam nos rótulos 3 e 7 formam um laço natural, segundo a definição de Appel e Palsberg. Esse laço é controlado pela condição de parada $i_1 \leq n$. A variável n, o limite do laço, é dependente da entrada. Existe um valor de n, a saber MAX_INT, que força o laço a não-terminar.

3 Detecção e Prevenção de Não-Terminações

Nesta seção descreveremos nossa técnica para detectar vulnerabilidades de não-terminação. Esse algoritmo de detecção fornece os subsídios necessários a uma segunda técnica que introduzimos neste artigo: o *saneamento de laços*.

3.1 Detecção Automática de Não-Terminação

Dizemos que um laço é *alcançável* quando as condições que o controlam usam valores que dependem de dados de entrada do programa. Note que um laço alcançável não é necessariamente vulnerável. A título de exemplo, o programa

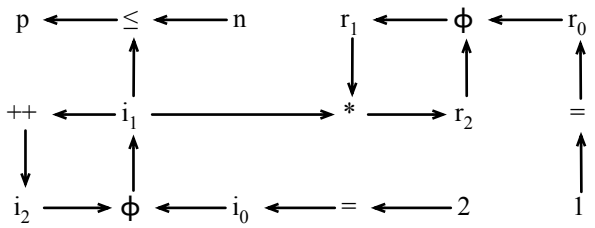

Figura 2. (a) Grafo de dependências da função `fact`, construído a partir do CFG visto na figura **??**(b).

da Figura **??**(c), uma sutil alteração da função `fact` inicialmente vista na figura **??**(a), termina para qualquer entrada, embora ele contenha um laço alcançável. Utilizamos o *grafo de dependências de dados* para determinar laços alcançáveis. Esse grafo é definido da seguinte forma: para cada variável v no programa, nós criamos um nodo n_v, e para cada instrução i no programa, nós criamos um nodo n_i. Para cada instrução $i : v = f(\ldots, u, \ldots)$ que define uma variável v e usa uma variável u nós criamos duas areastas: $n_u \leftarrow n_i$ e $n_i \leftarrow n_v$. O grafo de dependências que extraímos a partir d CFG visto na figura **??**(b) é mostrado na figura **??**.

Um caminho entre uma entrada do programa, e o predicado que controla o laço é uma condição necessária para um ataque de não-terminação. O grafo de dependências de nosso exemplo apresenta tal condição: existe um caminho que une o nodo correspondente ao parâmetro `n`, uma entrada, ao nodo que corresponde a `p`, o predicado de controle do laço. Esse tipo de caminho, uma vez contruído o grafo, pode ser encontrado em tempo linear no número de arestas do grafo - normalmente proporcional ao número de nodos - via a simples busca em profundidade ou largura.

Diremos que um laço é *vulnerável* quando ele é alcançável, e, além disso, sua condição de parada é dependente de alguma operação *cíclica* passível de estouro de precisão. Seguindo a definição de laços de Appel e Palsberg, uma operação cíclica é qualquer instrução que ocorre no corpo S do laço. Por exemplo, no CFG da figura **??**(b), as instruções $i_2 = i_1 + 1$ e $r_2 = r_1 \times i_1$ são cíclicas. O laço daquele exemplo encaixa-se em nossa definição de vulnerabilidade, pois sua condição de parada é alcançável a partir da entrada, e depende de uma instrução cíclica passível de estouro de precisão: $i_2 = i_1 + 1$.

A nossa definição de vulnerabilidade inclui muitos laços que não são concretamente vulneráveis, tais como aquele visto na figura **??**(c). Seria possível utilizar técnicas computacionalmente intensivas, tais como algoritmos de satisfabilidade, para refinar a nossa definição, eliminando alguns desses falsos positivos. Tal abordagem já foi utilizada em trabalhos anteriores ao nosso [**?**,**?**,**?**,**?**,**?**]. Por outro lado, os próprios autores desses trabalhos reportam que dificilmente suas