Programação Segura em GNU/Linux

José Ricardo Simões Rodrigues*

Julho de 2003.

Resumo

Este breve texto procura falar acerca da programação segura em GNU/Linux, principalmente quanto à linguagem `C`, vez que o código fonte de grande parte dos aplicativos críticos do sistema é escrita em `C`.

Sumário

1 Condição de Corrida

Quando o resultado de uma operação computacional varia de acordo com as velocidades relativas dos processos diz-se que existe uma condição de corrida (*race condition*). É necessário evitar condições de corrida para garantir que o resultado de uma computação não varie entre uma execução e outra. Condições de corrida resultam em computações paralelas errôneas, pois cada vez que o programa for executado (com os mesmos dados) resultados diferentes poderão ser obtidos. A programação de computações paralelas exige mecanismos de sincronização entre processos, e por isso sua programação e depuração é bem mais difícil do que em programas tradicionais.

`Race condition` ou condição de corrida é mais comum no Unix e no Linux. Na prática, ataques que explorem tal erro consistem em fazer algum programa que rode como `root` (super-usuário) executar alguma falha que possa lhe enviar para o `shell` do sistema. O programa com maiores problemas de *race condition* até hoje é o `sendmail`, serviço de e-mail padrão do Unix. É possível encontrar falhas até em versões mais recentes.

Para evitar condições de corrida, devemos:

- Definir uma seção crítica: dados compartilhados ou ações críticas que levem à condição de corrida;
- Garantir exclusão mútua: se um processo está na seção crítica, todos os outros devem ficar impossibilitados de fazer a mesma coisa;
- Empregar sincronização: um mecanismo para garantir a correta execução de processos cooperantes.

Na figura **??**, um pseudo-código exemplificando a seção crítica de um processo.

*O autor é estudante do curso de Pós Graduação *Lato Sensu* em Administração de Redes Linux da Universidade Federal de Lavras. O presente artigo foi escrito como atividade da disciplina Segurança em Redes e Criptografia ministrada pelo Prof. Joaquim Quinteiro Uchôa. Endereço para correspondência eletrônica: `simoes@uni.de`

```
repete
     entra seção
          seção crítica
     sai seção
          seção restante
até falso;
```

Figura 1: A seção crítica de um processo

Assim, temos que a solução para condição de corrida deve ter os seguintes requisitos:

- Exclusão mútua: se o processo P está executando sua SC, então nenhum outro processo pode estar;
- Imposições: nenhuma imposição deverá ser feita quanto à velocidade de execução dos processos;
- Progresso: nenhum processo executando fora da SC poderá bloquear outro processo; e
- Espera limitada: não deve haver postergação infinita.

No pequeno código da figura **??** demonstraremos uma *race condition* devido à falta de permissões/testes durante a abertura de um arquivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#define ARQ "/tmp/arquivo"
int main(void)
{
    FILE *fd;
       if (access(ARQ, F_OK) < 0)
       {
          fprintf(stderr,"O arquivo temporário não existe.\n");
          fd = fopen(ARQ,"a");
          fprintf(fd, "Arquivo criado.\n");
       }
       else
       {
          fprintf(stdout,"O arquivo já existe\n");
          fprintf(stdout,"Adicionando \"hello world\" no final...\n");
          fd = fopen(ARQ,"a");
          fprintf(fd,"hello world\n");
 }
       fclose(fd);
       return(0);
}
```

Figura 2: Código fonte do programa `vulneravel.c`

Compilamos e rodamos o comando `chmod +s vulneravel` para dar permissão de root. Executando este programa, vemos que ele não verifica as permissões do do arquivo temporário quando o cria nem faz um teste para ver se está realmente escrevendo em um arquivo do qual é proprietário.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
int main(int argc, char *argv[4])
{
       if (argc < 2)
       {
         fprintf(stderr,"Sintaxe errada: %s <\"programa\">\n",argv[0]);
         exit(-1);
       }
       while (1)
       {
        system("rm -rf /tmp/ arquivo;ln -s /etc/passwd arquivo");
        system(argv[1]);
        break;
       }
       return(0);
}
```

Figura 3: Código fonte do programa `exploit.c`

Quando executássemos o programa `exploit.c` compilado, vide código na figura **??**, este iria criar um link simbólico para o `/etc/passwd` e o programa iria escrever para esse mesmo arquivo. Como o programa tinha a flag `+s` ativada e o seu dono era o administrador do sistema nada o poderia parar de modificar qualquer arquivo no sistema.

Uma boa maneira de evitar estes problemas, além de fazer testes exaustivos e não tomar nada como garantido no nosso software, poderíamos criar um arquivo temporário com um nome aleatório. Isto iria dificultar bastante a tarefa de qualquer indivíduo que tentasse aproveitar-se do nosso programa.

2 Buffer overflow

Ataques de Buffer overflow são um problema de segurança permanente. Chegam a contabilizar aproximadamente metade dos problemas de vulnerabilidade de segurança. Segundo Brian Snow da NSA, ataques de buffer overflow ainda serão um problema em vinte anos.

Programas escritos em C são particularmente suscetíveis de *buffer overflow*. C possui muitas funções inseguras se não usadas de modo apropriado, sem levar em conta que o C permite manipulação direta de ponteiros sem uma checagem de segurança. Apesar disso, muitos programas críticos são escritos em C.

O mais simples ataque de buffer overflow é o *stack smashing* que sobrescreve um buffer de uma pilha para substituir o endereço de retorno. Quando a função retorna, o controle pulará para o endereço que foi colocado na pilha pelo cracker, possibilitando-o executar código arbitrário.

O *buffer overflow* é um ataque usado há muito tempo e que ainda será muito usado. Como já dissemos, compreende em lotar os *buffers* (memória disponível para aplicativos) de um servidor e incluir na sua lista de processos algum programa tal como um `keylogger` ou um `trojan`. Todos os sistemas são vulneráveis a buffer overflows e a solução é a mesma, procurar se já existem correções existentes. Novos erros desse tipo surgem todo dia. Atualização do servidor é o único remédio. Um dos usos famosos

do *buffer overflow* é o `telnet` reverso. Ele consiste em fazer a máquina alvo conectar-se a um servidor no computador do cracker, fornecendo-lhe um shell (prompt) de comando. O `netcat`, chamado de "canivete suíço do TCP/IP", é uma espécie de "super-telnet", pois realiza conexões por UDP, serve como servidor, entre outras tarefas. Ele é o mais utilizado para a realização do `telnet` reverso, e pode ser usado tanto na arquitetura Windows quanto nos unices.

3 Erros de formatação

São erros que ocorrem com o uso de funções como sprintf(), geralmente *buffer overflows*. Veja a figura **??**.

```
# Possível brecha para buffer overflows
sprintf(buffer, variable);
# sintaxe ok
sprintf(buffer, "%s", variable);
```

Figura 4: Excerto de código-fonte C

Um pequeno programa de nome `PScan` ajuda a evitar esse problema varrendo o código em busca de possíveis problemas com funções tipo `printf()`. Cheque a url `http://www.striker.ottawa.on.ca/~aland/pscan/`.

Existe um tipo de erro parecido. Trata-se não validação de formulários web. Vamos supor que temos um sítio web com um simples formulário com dois campos, como a figura **??**.

```
NOME: _______________
MAIL: _______________
```

Figura 5: Formulário web

O formulário iria enviar os dados para um CGI que, por sua vez, iria inserir estes dados numa base de dados SQL.

O CGI, em pseudo código, seria algo do gênero da figura **??**

```
# partimos do pressuposto que o $NOME e o $EMAIL são os
# dados enviados pelo formulário
$PEDIDO="insert into tabela values('','$NOME','$EMAIL');"
executar_pedido_em_servidor_sql($PEDIDO);
```

Figura 6: Pseudo-código CGI

Suponhamos que um usuário enviasse para o CGI as variáveis com o conteúdo da figura **??**.

O CGI iria então executar o pedido da figura **??** no servidor.

Visto que o SQL ignora o que estiver para a frente dos caracteres "–" num pedido, teríamos neste momento completo acesso à base de dados de SQL.

Este tipo de problema é bastante comum hoje em dia com o crescente número de ferramentas baseadas em tecnologias para a WEB.

```
$NOME="José Silva";
$EMAIL="email@ficticio.') and INSERIR_INSTRUÇÃO_SQL_AQUI--";
```

Figura 7: Código SQL malicioso

```
insert into tabela values('','José Silva','email@ficticio') and
INSERIR_INSTRUÇÃO_SQL_AQUI--");
```

Figura 8: Código SQL malicioso sendo executado

Para solucionar este problema bastaria filtrar os `escape chars` (caracteres de escape) como, por exemplo, o ', & e qualquer outro caracter que possa ser utilizado para escapar ao nosso pedido normal ao servidor.

4 Programação segura com `C`

Abaixo, temos algumas falhas usuais que devem ser evitadas quando programamos em `C`.

4.1 Chamadas a funções como a `system()`

Conforme o `ANSI C`, podemos ver que o cabeçalho da função `system()` é o seguinte:

```
int system (const char * string);
```

A função aceita como argumento uma string que será executada no shell e retorna 1 em caso de erro (por exemplo, se o `fork()` tiver falhado) ou o valor retornado pelo programa executado.

Como exemplo de utilização desta função podemos ver algo bastante usual como o da figura

Existe em Linux uma variável ambiente da shell chamada PATH.

Essa variável contém os caminhos para os diretórios onde o shell procura os comandos inseridos na mesma.

Não será necessário pensar muito para arranjar então maneira de nos aproveitarmos do programa para executar código nosso: basta, por exemplo, instruir o shell a procurar pelos binários (no caso clear) primeiramente no diretório corrente.

Se o programa vulnerável tivesse privilégios de administrador teríamos neste momento acesso completo sobre o sistema.

4.2 O *Internal Field Separator*

A variável IFS, acrônimo de *Internal Field Separator* serve para especificar ao shell qual o caracter delimitador dos argumentos.

Se o IFS é igual a " "e executarmos `ls -al` o shell sabe que o `ls` é o nome do binário que queremos executar e o `-al` é o primeiro e único argumento.

Se o IFS é igual a "/"e executarmos `ls/-al`, o shell sabe que o `ls` é o nome do binário que queremos executar e o `-al` é o primeiro e único argumento.

O problema com esta função é que todas as variáveis ambiente do shell são passadas para o ambiente do programa que é executado pelo `system()`.

Este problema é resolvido através do uso de funções da família `exec()`. Estas funções permitem um controle e filtragem sobre as variáveis de ambiente e argumentos do programa a executar.

4.3 *Dynamic Link Libraries*

O sistema operacional Linux utiliza bibliotecas compartilhadas.

Este conceito é bastante parecido com o uso de bibliotecas *.dll* no Microsoft Windows.

Esta idéia possibilita algumas vantagens, como por exemplo uma compilação mais rápida do código e menos espaço ocupado em disco. Como desvantagens temos o tempo extra de carregamento das bibliotecas durante a execução do programa e a possibilidade de enganar o binário a executar as nossas próprias funções através de métodos que geralmente têm como finalidade um *debugging* das aplicações mais simples.

Podemos comparar os prós e contras das bibliotecas dinâmicas às vantagens e desvantagens da alocação dinâmica de memória num programa.

É possível controlar as operações do *dynamic loader* através de variáveis ambiente do shell. A mais importante é a `LD_PRELOAD`.

Tal como o que foi expresso na sua definição, é possível enganar o programa induzindo-o a executar funções nossas. Isto é muito simples de alcançar interceptando funções usadas no programa que queremos enganar carregando funções nossas através do `LD_PRELOAD`.