Segurança de Sistemas Operacionais

Roberto Sadao Yokoyama

UTFPR-CP

Agosto, 2016

Aula de hoje: [1]

1. Segurança de sistemas de arquivos
2. Segurança de programa de aplicação

[1] Slides baseados no material do livro: Goodrich [1]

Segurança de sistemas de arquivos

Princípios de segurança de arquivos

- Arquivos e pastas são gerenciadas pelo sistema operacional
- Aplicações, incluindo *shell*, acessam os arquivos através de API
- Controle de acesso: permitir ou negar um certo tipo de acesso a um arquivo/pasta por usuário/grupo
- Operações de arquivo
  - leitura/gravação/execução
  - Abrir arquivo: retorna identificador de arquivo
  - Fechar arquivo: invalida o identificador de arquivo
- Pastas: permissão de listagem/execução

Controle de acesso de arquivo Linux

Controle de acesso para:

- Arquivos
- Diretórios
- Além disso...
  - \dev\..: dispositivos
  - \mnt\..: sistemas de arquivos montados

O que acontece se um usuário tem permissão de escrita para um arquivo, mas não para o diretório onde reside o arquivo?

Sistema de arquivos Linux

- Árvore de diretórios (pastas)
- Cada diretório possui zero ou mais arquivos ou diretórios
- *Hard Link*
  - Cria links de diretório ou arquivo para um arquivo
  - O mesmo arquivo (link) pode ter ligações diretas de vários arquivos, cada um com seu próprio nome, mas todos possuem o mesmo proprietário, grupo e permissões
  - Arquivos são excluídos quando não há mais *hard link* para eles
- Link simbólico (*symlink*)
  - De um arquivo/diretório para um arquivo de destino ou diretório
  - Armazena o caminho para o alvo, que é "atravessado por cada acesso"
  - O mesmo arquivo ou diretório pode ter vários *links* simbólicos para ele
  - Deletar o *link* não afeta o alvo (arquivo/diretório)
  - Deletar o alvo inválida (mas não remove) o link simbólico
  - É análogo ao "atalho"do Windows

Permissões de arquivos

- Cada arquivo pertence a um usuário e tem um grupo associado
- Para acessar um arquivo, cada pasta ancestral deve ter permissão de execução e o próprio arquivo deve ter permissão de leitura
- O proprietário do arquivo recebem o poder de alterar permissões nesses arquivos (arbitrário)
- Permissões normalmente são mostradas na notação de 10 caractere (d: diretório, r: leitura, w:escrita e x:execução)
- Para ver as permissões use: *ls -l*

jk@sphere: /test$ ls -l
total 0
-rw-r- - - - - 1 jk ugrad 0 2005-10-13 07:18 file1
-rwxrwxrwx 1 jk ugrad 0 2005-10-13 07:18 file2

Exercício

Permissões	Owner	Group	Others
-rw - r - - r - -			
-rw - r - - - - -			
-rwx - - - - - -			
-r- - r - - r - -			
-rwxrwxrwx			

Exercício - resposta

Permissões	Owner	Group	Others
-rw-r- - r - -	Leitura/escrita	Somente leitura	Somente leitura
-rw-r- - - - -	Leitura/escrita	Somente leitura	Nenhuma permissão
-rwx- - - - - -	Leitura/escrita /execução	Nenhuma permissão	Nenhuma permissão
-r- -r- -r - -	Somente leitura	Somente leitura	Somente leitura
-rwxrwxrwx	Leitura/escrita /execução	Leitura/escrita /execução	Leitura/escrita /execução

Permissões para diretórios

- Os bits de permissão interpretados de maneira diferente para diretórios
- Bit *leitura* permite listar nomes de arquivos do diretório, mas não propriedades como tamanho e permissões
- Bit *escrita* permite criar e apagar os arquivos dentro do diretório
- Bit *execução* permite "entrar"no diretório e obter as propriedades dos arquivos
- As linhas dos diretórios da saída do *ls -l* iniciam com *d*, como abaixo:
  jk@sphere: /test$ ls –l
  Total 4
  drwxr-xr-x 2 jk ugrad 4096 2005-10-13 07:37 dir1
  -rw - r - - r - - 1 jk ugrad 0 2005-10-13 07:18 file1

Exercício

Permissões	Descrição
drwxr-xr-x	
drwxrwx- - -	
drwx- -x- - -	
drwxrwxrwx	

Exercício – resposta

Permissões	Descrição
drwxr-xr-x	todos podem entrar e listar arquivos do diretório, mas somente o proprietário pode adicionar/apagar arquivos
drwxrwx- - -	acesso total ao proprietário e grupo, nenhuma permissão aos outros
drwx- -x- - -	acesso total ao proprietário, o grupo pode acessar e ler os nomes de arquivos no diretório, nenhuma permissão aos outros
drwxrwxrwx	acesso total para todos

Bits especiais de permissão

- **Set-user-ID** (setuid bit): em arquivos executáveis, faz com que o programa seja executado com permissões do proprietário do arquivo, independentemente de quem executá-lo.
- **Set-group-ID** (setgid bit):
  - em arquivos executáveis, faz com que o programa seja executado com permissões do grupo do arquivo, independentemente do usuário que executá-lo nesse grupo
  - em diretórios, faz com que arquivos criados dentro do diretório tenha a mesma permissão do grupo. É útil para diretórios compartilhados por vários usuários com grupos diferentes do grupo padrão
- **Stick bit**: em diretórios, impede que os usuários deletem ou renomeiem arquivos que eles não são proprietários.

Alterar permissões

- As permissões são alteradas com **chmod**. Somente o proprietário do arquivo ou *root* pode alterar permissões
- Se um usuário possui um arquivo, o usuário pode usar **chgrp** para definir o grupo do arquivo
- *root* pode alterar a propriedade de arquivo com **chown** (opcionalmente, pode alterar o grupo com o mesmo comando)
- A opção de -R de recursivo, pode ser usado para alterar permissões de subdiretórios

Alterar permissões: exemplos

Comando	Descrição
chown -R root dir1	Altera a propriedade de dir1 e tudo de dentro para *root*
chmod g+w,o-rwx file1 file2	Adiciona a permissão de gravação grupo de file1 e file2, negando todo o acesso a outros
chmod -R g=rwX dir1	Adiciona a permissão de leitura/gravação de grupo para dir1 e tudo dentro.
chgrp testgrp file1	Define o grupo do file1 para testgrp, se o usuário for um membro desse grupo
chmod u+s file1	Define o bit setuid na file1.

Notação numérica

- 644: leitura/escrita para o proprietário e somente leitura para outros
- 775: leitura/escrita/execução para o proprietário e grupo, leitura e execução para outros
- 640: leitura/escrita para o proprietário, somente leitura para grupo e sem permissão para outros
- 777: leitura/escrita/execução para todos

Introdução

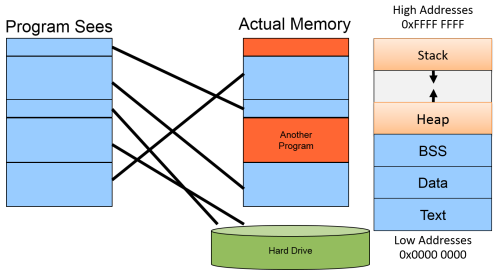
- Muitos ataques não exploram diretamente alguma falha do núcleo do SO, mas sim atacam programas inseguros
- *Overflow* de *buffer* baseado em pilha
- *Overflow* de *buffer* baseado em *heap*

Ataque de *overflow* de *buffer*

- Um dos bugs mais comuns de SO é um estouro de *buffer*
  - O desenvolvedor falha em implementar um código que não verifica se uma sequência de entrada se encaixa em sua matriz de alocada de memória
  - Uma entrada para o processo de execução excede o comprimento do *buffer* ("matriz alocada")
  - A sequência de entrada substitui uma parte dos dados do processo na memória
  - Faz com que o aplicativo se comporte incorretamente e inesperadamente
- Efeito de um estouro de *buffer*
  - O processo pode operar em dados mal-intencionados ou executar código malicioso, passado pelo invasor
  - Se o processo é executado como *root*, o código malicioso irá executar com privilégios de *root*

Espaço de endereçamento

- Cada programa precisa acessar a memória para executar
- Por simplicidade, seria bom permitir que cada processo (ou seja, cada programa em execução), agir como se ele possui toda memória
- O modelo de espaço de endereço é usado para realizar essa tarefa
- Cada processo pode alocar espaço em qualquer lugar na memória
- A maioria dos *kernels* gerencia alocação de cada processo de memória por meio do modelo de memória virtual
- Como a memória é gerenciada é irrelevante para o processo



Vulnerabilidades e método de ataque

- Cenários de vulnerabilidade
  - O programa tem privilégios de root (setuid) e executa um *shell*
- Método de ataque típico
  1. Encontrar a vulnerabilidade
  2. Engenharia reversa do programa
  3. Construir um *exploit*[2]

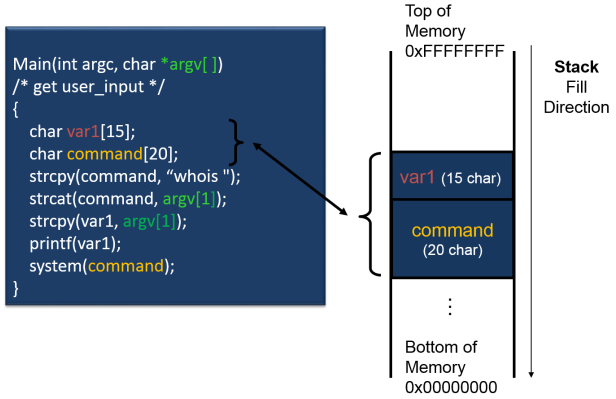
[2] Um exploit é qualquer entrada (ou seja, uma pedaço de software, uma sequência do argumento ou a sequência de comandos) que se aproveita de um bug, falha ou vulnerabilidade a fim de provocar um ataque

Ataque de estouro de *buffer*

- O invasor explora um *buffer* não verificado para executar um ataque de estouro de *buffer*
- O objetivo final para o atacante é conseguir um *shell* para permitir executar comandos arbitrários com privilégios elevados
- Tipos de ataques de estouro de *buffer*:
  - Esmagamento de *heap* ou estouro de heap
  - Esmagamento de pilha ou estouro de pilha

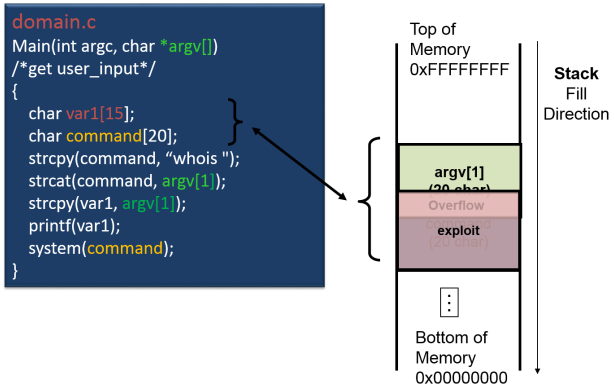
Buffer overflow

- Recupera a informação de registo de domínio
- ex. domínio brown.edu



Vulnerabilidade *strcpy()*

- argv[1] é a entrada do usuário
- strcpy(dest, src) não verifica o buffer
- strcat(d, s) concatena strings

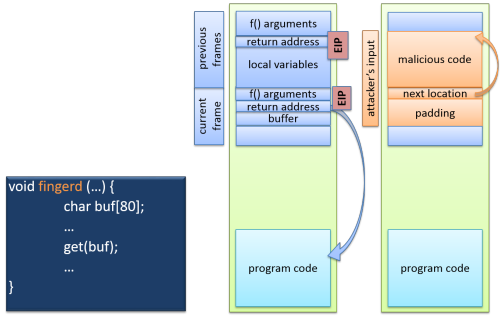


strcpy() vs. strncpy()

- Função strcpy () copia a sequência do segundo argumento para o primeiro argumento
  - strcpy(dest, src)
  - Se cadeia de caracteres da fonte > destino, os caracteres podem ocupar o espaço de memória usado por outras variáveis
  - O caractere nulo é anexado ao final automaticamente
- Função strncpy() copia a sequência de caracteres especificando o número *n* de caracteres para copiar
  - strncpy(dest, src, n); dest[n] = '\0'
  - Se a sequência de caracteres de fonte é mais do que a sequência de caracteres de destino, os caracteres excedentes serão descartados automaticamente
  - Deve ser colocado o caractere nulo manualmente

Retorno de endereço

- A chamada de sistema Unix fingerd(), que é executado como *root* (ele precisa acessar arquivos restritos), costumava ser vulnerável a estouro de *buffer*
- A ideia é escrever o código malicioso em *buffer* e substituir o endereço de retorno para apontar para o código malicioso
- Quando o endereço de retorno for atingido, ele agora irá executar o código malicioso com os plenos direitos e privilégios de *root*



Injeção de código *shell*

- Um atacante toma controle de computador atacado, então insere um código "shellcode"
  - Um shellcode é:
  - Código de máquina com um conjunto de instrução nativos da CPU (por exemplo, x86, x86-64, braço, sparc, risc, etc.)
  - Injetado como uma parte do *buffer* estourado
- Injetamos o código diretamente no *buffer* que enviamos para o ataque
- Um *buffer* que contém o código *shell* é um "*payload*"

Mitigação de estouro de *buffer*

- Nós sabemos como acontece um estouro de *buffer*, mas por que acontece?
- Esse problema pode não ocorre em Java; é um problema de C
  - Em Java, objetos são alocados dinamicamente na heap (exceto ints, etc.)
  - Java também não é possível fazer aritmética de ponteiro
  - Em C, no entanto, você pode declarar coisas diretamente na pilha
- Aleatorização de espaço de endereço: altera o endereço de modo aleatória, no qual a pilha está localizada, para cada processo.

Detecção de estouro do *buffer* baseado em pilha usando um canário aleatório

O canário é colocado na pilha antes o endereço do remetente, para que qualquer tentativa de sobre-escrever o endereço do remetente também excesso escreve o canário.

Normal (safe) stack configuration:

Buffer overflow attack attempt:

Exercício-1

Exemplo de um programa setuid

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

static uid_t euid, uid;

int main(int argc, char* argv[])
{
        FILE *file;
        /*Armazena IDs de usuario verdadeiras e efetivas */
        uid = getuid();
        ueuid = geteuid();
        /*Rebaixa privilegio */
        seteuid(uid);
        /*Faz algo util*/
        /*...*/
```

Exercício-1

Exemplo de um programa setuid – continuação

```
        /*Aumenta privilegios*/
        seteuid(euid);
        /*Abre arquivo*/
        file=fopen("/home/admin/log","a");
        /*Rebaixa privilegios novamente*/
        seteuid(uid);
        /*Escreve no arquivo*/
        fprintf(file,"Alguem usou este programa\n");
        /*Fecha o fluxo de arquivo  e retorna*/
        fclose(file);
        return 0;
}
```

Questão:

Qual o problema deste programa com relação as permissões de arquivo?

Exercício-2

Exemplo de um programa com descritor de arquivos

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        /*Abre o arquivo de senhas para leitura*/
        FILE *passwords;
        passwords=fopen("/home/admin/passwords","r");
        /*Le as senhas e faz algo util*/
        /*...*/
        execl("/home/joe/shell","shell", NULL);
}
```

Questão:

Qual o problema deste programa com relação as permissões de arquivo? (pesquise: funções *fcntl()*)

Atividade

Programa C simples sujeito à exploração

```
#include <string.h>
#include <stdio.h>
void overflowed(){
        printf("%s\n", "Execution Hijacked");
}
void function1(char *str){
        char buffer[5];
        strcpy(buffer, str);
}
void main(int argc, char *argv[]) {
        function1(argv[1]);
        printf("%s\n", "Executed normally");
}
```

Atividade

Observe que, em circunstâncias normais, *overflowed* jamais é chamado. Como você provocaria um *buffer overflow* do programa e assumiria o controle do programa, fazendo com que a função *overflowed* seja executada.

Referências

[1] M. T. Goodrich and R. Tamassina. *Introdução à Segurança de Computadores*. Bookman, 2013.