8- Invasão da Pilha de Execução

Nesse laboratório você vai usar seus conhecimentos sobre a pilha de execução para entender como são realizados ataques do tipo *buffer overflow*. Este tipo de ataque se aproveita da ausência de verificação de "final de array" em C para sobrescrever a pilha de execução, alterando o endereço de retorno lá presente, e possivelmente outras informações. Ainda que com muitas simplificações, esse laboratório mostra como é importante escrever código seguro...

Ao longo do tempo, os sistemas foram ficando mais protegidos contra esse tipo de ataque. Neste laboratório, vamos precisar desabilitar algumas dessas proteções. Uma delas é feita pelo compilador `gcc`: a inserção de um código que verifica a integridade da pilha após uma chamada de função. Para desabilitar essa proteção, executamos o `gcc` com a opção **`-fno-stack-protector`**.

Outro mecanismo de segurança presente nos sistemas atuais é impedir que um programa execute código que "resida" na área de dados, ou na pilha. Para desabilitar essa proteção, passamos ao `gcc` a opção **`-Wa,--execstack`**.

A última proteção que precisamos desabilitar é a *randomização* de endereços, que impede que os endereços dos dados de um programa (e de sua pilha) sejam sempre os mesmos, em qualquer execução desse programa. Para realizar esse laboratório sem essa proteção, execute na linha de comando

```
setarch x86_64 -R /bin/bash
```

para entrar em um ambiente de execução sem randomização de endereços, e execute todos os comandos do laboratório dentro desse ambiente.

Vamos utilizar o programa bufbomb.c para testar alguns exemplos. Para isso, busque aqui o arquivo 'bufferbomb.tar'. Coloque-o em uma pasta onde possa criar uma subpasta para este laboratório e em seguida execute:

```
tar xvf bufferbomb.tar
```

Isso deve criar uma pasta `bufferbomb` com os arquivos:

- `bufbomb.c` - código a ser "atacado"
- `buf.c` - manipulação do buffer
- `hex2raw.c` - função auxiliar para criar as strings de entrada
- `Makefile` - configuração da compilação e ligação

Para criar o programa executável bufbomb, escreva:

```
make bufbomb
```

O utilitário **`make`** é uma ferramenta que constrói programas e bibliotecas a partir de regras definidas em um arquivo denominado **`Makefile`** (ou **`makefile`**).

O programa bufbomb é bastante simples, e basicamente chama a função `getbuf`, definida no arquivo `buf.c`. Execute `./bufbomb` escrevendo um texto qualquer quando o programa ficar a espera de entrada, e veja o que acontece.

Agora vamos interferir no funcionamento de `bufbomb` fornecendo como entrada uma string de valores "apropriados". Veja que a função `getbuf` declara um array de char e chama `Gets` para preencher esse array.

Para obtermos essa entrada "apropriada", usaremos o programa auxiliar `hex2raw`. Esse programa lê, de um arquivo texto, uma sequência de valores em formato hexadecimal e gera um arquivo *binário* com esses valores. Isto é, se o arquivo de entrada `se` contiver:

```
00 00 00 00
00 00 00 01
```

e escrevermos:

```
./hex2raw < se > se.raw
```

o programa `hex2raw` vai ler o arquivo de entrada **`se`** e gerar o arquivo de saída **`se.raw`**.

Podemos depois executar:

```
./bufbomb < se.raw
```

para fornecer a `bufbomb` a sequência de bytes equivalente ao conteúdo do arquivo mostrado. Como o programa não testa o tamanho dessa sequência, podemos dar como entrada uma string arbitrariamente grande.

Essa é a base do ataque de *buffer overflow*: sobrescrever a pilha de execução, alterando os endereços de retorno que estão empilhados para apontarem para um outro código que o intruso deseja que seja executado, ou para forçar a execução de código inserido "maliciosamente" na pilha.

Exercícios

1. Uma forma de ataque que se tornou bastante comum é desviar o controle para alguma função já existente no próprio código "atacado", mas que não seria chamada normalmente. Examine o código de `bufbomb.c`. Veja que existe uma função `danger`, normalmente chamada pela função `protect`, que determina se o usuário tem as credenciais apropriadas para executar a chamada. Você irá criar uma string de bytes que desviará o controle para `danger` sem passar por `protect`. Para isto, a string dada como entrada para `bufbomb` deve sobrescrever a pilha de execução, **colocando o endereço de `danger` no lugar do endereço de retorno de `getbuf`**.

   Você vai ter que criar uma sequência de bytes com tamanho suficiente para ocupar desde o início do espaço ocupado pelo array local **`buf`** até o endereço de retorno de `getbuf`. Os valores nas posições que não correspondem ao endereço de retorno de `getbuf` podem ser preenchidos com qualquer valor.

   Desenhe a pilha a partir da chamada a `getbuf`, na `main`. O código de `getbuf` está em `buf.c`. Para descobrir o endereço de `danger` vamos usar o utilitário `objdump`, que permite a inspeção de arquivos objeto e executáveis. A opção **-d** deste utilitário produz como saída um *desassembly* do código de máquina armazenado no arquivo. Com o comando

   ```
   objdump -d bufbomb > bufbomb.d
   ```

   redirecionamos a saída do `objdump` para um arquivo chamado `bufbomb.d`. Inspecione esse arquivo: o endereço de `danger` estará ao lado do *label* **danger**. Lembre-se que esse endereço deve aparecer na pilha em little-endian!

   Verifique também no "desassembly" da função `getbuf` onde começa, na pilha de execução, o array `buf`. Como o endereço do array é fornecido como parâmetro para `Gets`, verifique como é calculado o valor desse parâmetro (isto é, veja qual é o offset em relação ao `%rbp` correspondente ao endereço de `buf`).

   Crie um arquivo `stringinvasora` contendo uma sequência de valores hexa, por exemplo (quebras de linha não fazem diferença):

   ```
   00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00
   ....
   86 04 80 00 00 00 00 00
   ```

   Lembre-se que a sequência tem que ter um tamanho adequado para que o endereço de danger sobrescreva o endereço de retorno de getbuf!

   Utilize agora o programa `hex2raw` para gerar sua string e depois forneça o arquivo resultante a bufbomb:

   ```
   ./hex2raw < stringinvasora > stringinvasora.raw
   ./bufbomb < stringinvasora.raw
   ```

   Você verá se seu programa fez o que você queria pelas mensagens exibidas: se você conseguiu desviar o controle para `danger`, a mensagem exibida por essa função deverá estar no terminal. Mas para observar o efeito "danoso" da execução de `danger`, crie um arquivo chamado **`vitima`** dentro da pasta `bufferbomb`. Repare que a função `danger`, ao executar, remove esse arquivo!

   obs: Caso sua sequência de bytes esteja bem construída, seu programa irá chamar `danger` e depois deve gerar uma *segmentation fault*. Iremos mudar isso agora.

2. No item anterior, o programa gera um segmentation fault ao tentar retornar da função `danger`, pois não há um endereço de retorno no lugar adequado na pilha. Corrija isto, criando um novo arquivo `stringinvasora2`. Nesse arquivo, estenda a sequência de bytes de maneira a preencher a pilha com um endereço de retorno para `danger`. Use o endereço da função `smoke`, fazendo com que o controle vá para ela depois da execução de `danger`. (Como `smoke` chama `exit()`, o programa agora deve terminar elegantemente.)

3. Vamos experimentar agora ativar uma função passando um argumento para ela. Nosso novo objetivo é fazer `getbuf` "retornar" para a função `fizz`, passando como argumento um inteiro com valor `0x01020304`. Veja o código de `fizz`; se você conseguir invocá-la com esse argumento, ela imprimirá a linha: *fizz! You called fizz....*

   Neste exercício, vamos precisar do endereço do array `buf`. Repare que a função `getbuf` imprime este endereço antes de ler sua entrada. (Esse endereço deve ser sempre o mesmo, em todas as execuções. Se não for, provavelmente você se esqueceu de desabilitar a randomização de endereços...)

Crie um arquivo `codigo.s` com as instruções

```
movl $0x01020304, %edi
ret
```

e gere o arquivo objeto correspondente com o comando

```
gcc -c codigo.s
```

Se você executar, agora,

```
objdump -d codigo.o
```

você poderá obter o código de máquina correspondente a essas instruções (repare que elas preparam um argumento em `%edi`).

Prepare agora um arquivo `stringinvasora3` que sobrescreva a pilha da seguinte forma:

- as primeiras posições do array `buf` devem ser preenchidas com um código "malicioso" (as instruções de `codigo.s`);
- o endereço de retorno de `getbuf` deve ser sobrescrito com o endereço de `buf`, para forçar a execução desse código;
- a posição apropriada da pilha deve ser preenchida com o endereço de `fizz`, para que a instrução `ret` do código inserido na pilha force a invocação dessa função.

9- Código de Máquina

1. Traduza a função abaixo para *assembly*, criando um arquivo `foo.s`:

   ```
   int foo (int x) {
     return x+1;
   }
   ```

2. Use **`gcc -c foo.s`** para traduzir seu programa para linguagem de máquina (o gcc vai gerar um arquivo **`foo.o`**).

   Veja qual o código de máquina que seu programa gera, usando o comando **`objdump -d foo.o`** (a opção -d do objdump faz um "disassembly" do arquivo objeto).

3. Escreva agora um programa em C como descrito a seguir. Declare um array de bytes **global** (`unsigned char codigo[]`) preenchido com o código de máquina visto no item anterior (não declare esse array como uma variável local). Lembre-se que a saída do **objdump** mostra os códigos das instruções em hexadecimal. Use esses valores para preencher o array.

   Seu programa deve converter o endereço do array para um endereço de função. Para isso, declare o tipo "*ponteiro para uma função que recebe um `int` e retorna um `int`*", conforme abaixo:

   ```
   typedef int (*funcp) (int x);
   ```

   Na sua função `main` atribua o endereço do array a uma variável desse tipo:

   ```
   funcp f = (funcp)codigo;
   ```

   O ponteiro `f` armazena agora o endereço da função, ou seja, o endereço inicial do código da função, armazenado na memória. Você pode então usar `f` para chamar essa função como se fosse uma função C, fazendo, por exemplo:

   ```
   i = (*f)(10);
   ```

   Faça isso no seu programa, imprimindo a seguir o valor da variável `i` para poder verificar se o seu código de máquina foi realmente executado.

   Deve ser necessário compilar seu programa com

   ```
   gcc -Wall -Wa,--execstack -o seuprograma seuprograma.c
   ```

   para permitir a execução do seu código de máquina (sem a opção `-Wa,--execstack`, o sistema operacional abortará o seu programa, por tentar executar um código armazenado na área de dados). Execute o programa resultante e verifique a sua saída.

4. Traduza agora a função abaixo para assembler,

   ```
   int foo (int x) {
     return add(x);
   }
   ```

   Observe o código de máquina da nova função `foo` com o `objdump`. Note que o código gerado para a instrução `call` é

e8 00 00 00 00

Nessa instrução, o byte **e8** representa o código de *call*, e os quatro bytes seguintes (os bytes 00 neste exemplo), representam o deslocamento da função chamada (add) em relação à instrução seguinte ao *call* (isto é, a diferença entre os dois endereços: o endereço de add e o endereço da instrução seguinte ao call).

Esse deslocamento é armazenado em little endian, e pode ser um valor negativo ou positivo, dependendo do endereço da função chamada ser "menor" ou "maior" que o da instrução seguinte ao *call*.

Note que o deslocamento NÃO está correto no arquivo .o, pois o endereço da função chamada somente será conhecido no passo de *linkedição* (a função não está definida no módulo). O montador então preenche a posição do deslocamento com um valor "default", que será depois "corrigido" pelo *linkeditor*.

5. Declare agora a função add no seu arquivo C (o arquivo que contém a main):

```
int add (int x) {
  return x+1;
}
```

e modifique o programa para que ele preencha no array **codigo** o código de máquina da nova função `foo`.

Como você pode obter o endereço de **add** "programaticamente" (isto é, usando o operador &), seu programa C pode calcular qual deve ser o deslocamento correspondente à chamada de `add` no seu código de máquina.

Lembre-se que o seu código está armazenado no espaço reservado para o array e, portanto, o endereço do início do código é o próprio endereço do array. Se, por exemplo, a instrução `call` começa na posição **n** do array, a próxima instrução começará na posição **n+5** (pois a instrução `call` tem 5 bytes).

Modifique agora seu programa C, fazendo com que ele "corrija" a instrução *call*, preenchendo os quatro bytes após o *opcode* **e8** com o deslocamento calculado. Lembre-se que esse valor deve estar em *little-endian*!

Execute agora o seu programa, e verifique se tudo funciona corretamente!

6. Substitua agora no seu arquivo *assembly* a instrução **call add** por **jmp add**. Observe que o código de máquina dessa instrução é

e9 00 00 00 00

Assim como num `call`, o byte `e9` representa o código de **jmp** e os quatro bytes seguintes são o deslocamento, ou diferença, entre o endereço de destino do "jump" e o endereço da próxima instrução. Isto vale para todos os tipos de "jump" (incondicional e condicional).

10- Chamadas ao Sistema Operacional em Assembly

Nesse laboratório vamos explorar a interface com o sistema operacional.

1. Para começar, pegue o 'copia.c'. Compile e execute esse programa, testando-o com algum arquivos de entrada (pode ser o próprio arquivo `copia.c`):

```
> gcc -Wall -o copia copia.c
> ./copia copia.c
```

2. Observe o código de `copia.c`. A rotina `main` chama quatro funções definidas neste mesmo arquivo: `myopen`, `myread`, `mywrite` e `myclose`. Cada uma dessas funções é implementada por uma única linha, que contém uma chamada ao sistema operacional (na verdade, uma chamada a uma função ***wrapper***, provida pela biblioteca padrão de C).

As chamadas ao sistema operacional são documentadas na seção 2 do manual do Linux. Para ver a documentação de uma delas, basta escrever, por exemplo:

```
> man 2 open
```

3. Agora você deve substituir cada uma das funções (`myopen`, `myread`, `mywrite` e `myclose`) por funções equivalentes a elas, escritas em assembly.

Atenção: suas funções assembly não devem chamar as funções da biblioteca de C! Elas devem

usar uma interface direta com o SO.

Crie um arquivo `chamadas.s`, e comece substituindo a função `myopen`. Comente a definição dessa função no arquivo `copia.c`, mantendo o seu protótipo. Implemente `myopen` em assembly (ver instruções abaixo), e compile tudo agora com:

```
> gcc -Wall -o copia chamadas.s copia.c
```

Para descobrir o código das chamadas ao sistema, consulte esta tabela de chamadas.

Observe os seguinte pontos:

- Para ativar o sistema você vai usar a instrução **`syscall`**
- O código da chamada (open, read, etc) deve ser passado em %rax.
- Os parâmetros da são passados em %rdi, %rsi, %rdx, %r10, %r8 e %r9, nesta ordem.
- Não se esqueça de observar as convenções de C

4. Vá comentando, uma a uma, a definição das demais funções no arquivo `copia.c` e substituindo-as pela implementação em assembly em `chamadas.s`.

   A implementação em assembly de chamadas a serviços do sistema operacional que você acabou de fazer é semelhante à implementação das funções *wrapper* disponíveis na biblioteca padrão de C.

5. Agora escreva em assembly uma função `myotherwrite`, que receba um descritor de arquivo e uma string, e escreva essa string string no arquivo fornecido, usando uma chamada direta ao sistema operacional:

   ```
   int myotherwrite (int fd, char *s);
   ```

   Dica: você deve calcular o tamanho da string para passar como terceiro parâmetro para a syscall! (não use uma função pronta, esse cálculo é muito simples...)

   Para testar sua função, você pode usar o código abaixo:

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

   #define MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH)
   ssize_t myotherwrite(int fd, const void *buf);

   int main (int argc, char** argv) {
     ssize_t tam;
     int arq;
     char buf[] = "testando escrita direta\n";

     if (argc != 2) {
       fprintf(stderr,"forma correta: %s \n", argv[0]);
       return 1;
     }

     arq = open (argv[1], O_CREAT|O_RDWR, MODE);
     if (arq<0) { perror("abertura de arquivo"); return 1;}

     if ((tam = myotherwrite (arq, buf)) != sizeof(buf)-1) {
       perror("escrita:");
       return 1;
     }
     close (arq);
     return 0;
   }
   ```

11- Compilação e Ligação

1. Considere o programa formado pelos arquivos abaixo:

   - arquivo `temp1.c`:

     ```
     #include <stdio.h>
     #include "temp1.h"
     #include "temp2.h"

     int a = 1024;

     int main (void) {
       foo();
       printf("%d\n", a);
       return 0;
     }
     ```

   - arquivo `temp2.c`:

     ```
     #include "temp1.h"
     #include "temp2.h"
     ```

```
int b = 10;

void foo (void) {
  a = -3;
}
```

- arquivo `temp1.h`:

```
extern int a;
```

- arquivo `temp2.h`:

```
extern int b;
void foo(void);
```

Note que os arquivos `temp1.h` e `temp2.h` contém declarações dos simbolos globais exportados, respectivamente, por `temp1.c` e `temp2.c` A inclusão desses arquivos garante a consistência dessas declarações entre os módulos, tanto para os arquivos que definem os símbolos quanto para os que os importam.

Compile, ligue e execute esse programa, seguindo os passos abaixo:

```
> gcc -c -Wall temp1.c
> gcc -c -Wall temp2.c
> gcc -o prog  temp1.o temp2.o
```

(Você poderia fazer tudo de uma só vez chamando `gcc -Wall temp1.c temp2.c`, mas, neste caso, o `gcc` não gera os arquivos `.o`.)

2. Use o programa `nm` para inspecionar os símbolos usados por cada módulo (chame `nm temp1.o` e `nm temp2.o`). Você consegue inferir o significado das letras que aparecem na saída deste programa (`U`, `T`, etc.)? (Dica: "man nm".)

3. Troque no 1o arquivo (`temp1.c`) a linha `int a = 1024;` por `char a = 1;` e recompile esse módulo. O que acontece?

   Você percebe como a inclusão dos arquivos de header **inclusive pelos módulos que definem os símbolos** garante a consistência das declarações?

4. Restabeleça no arquivo `temp1.c` declaração `int a = 1024;`

   a. Troque no 2o arquivo (`temp2.c`) a linha `#include "temp1.h"` por `extern char a;`

      Recompile os dois módulos, gere um novo executável e execute o programa. O que aconteceu? Você consegue explicar?

   b. Agora troque no arquivo `temp2.c` a declaração `extern char a;` por `char a = 1;` e recompile este módulo.

      Tente gerar um novo executável. O que acontece agora? Inspecione novamente os símbolos dos módulos objeto usando o `nm`.

   c. E se a troca for por `static char a = 1;`, o que acontece? Por que?

5. Restaure a inclusão de `"temp1.h"` no 2o arquivo (`temp2.c`), e remova a declaração `static char a = 1;`.

   a. Agora remova **no primeiro arquivo** (`temp1.c`) a linha `#include "temp2.h"` (isto é, a declaração dos símbolos exportados por temp2).

      Troque em `temp1.c` a linha `foo();` por `b();`

      Recompile os dois módulos e gere um novo executável com

      ```
      gcc -Wa,--execstack -Wall -c temp1.c
      gcc -Wa,--execstack -Wall -c temp2.c
      gcc -o prog -Wa,--execstack temp1.o temp2.o
      ```

      Verifique com o `nm` as tabelas de símbolo de cada módulo (em especial, o símbolo `b`).

      Execute o programa. O que aconteceu? Por que?

   b. Troque agora o valor inicial de `b` em `temp2.c` por 0xC3. Recompile este módulo, gere um novo executável, e execute-o. O que acontece agora? Por que?