Computadores de Programação (DCC/UFRJ)

Aula 15: Referências a Memória fora dos limites e estouro de buffer

Prof. Paulo Aguiar

Problemas potenciais de corrupção da pilha

- C não faz restrições a referências a vetores
- Variáveis locais são salvas na pilha junto com informação de estado (como valores de registradores e endereço de retorno)
- Referências fora de limites podem causar corrupção da pilha

Exemplo de estouro de buffer

- Função **gets** lê da entrada padrão, copia caracteres para vetor apontado por ponteiro **s** e termina com caracter NULL (0x00)
- Se o buffer alocado para receber a string tiver apenas 8 posições e forem lidos mais de 7 caracteres, haverá estouro do buffer

Função echo com uso de gets

```
/* Lê uma linha de entrada e escreve de volta */
void echo () {
    char buf[8]; /* Pequeno demais!  /*
    gets(buf);
    puts(buf);
}
```

Estouro de buffer

Assembly da função echo

```
1 echo:
2         pushl   %ebp              Salva %ebp na pilha
3         movl    %esp, %ebp
4         pushl   %ebx              Salva %ebx
5         subl    $20, %esp         Aloca 20 bytes na pilha
6         leal    -12(%ebp), %ebx   Computa buf como %ebp-12
7         movl    %ebx, (%esp)      Armazena buf no topo da pilha
8         call    gets              chama gets
9         movl    %ebx, (%esp)      Armazena buf no topo da pilha
10        call    puts              chama puts
11        addl    $20, %esp         Desaloca pilha
12        popl    %ebx              restaura %ebx
13        popl    %ebp              restaura %ebp
14        ret     retorna
```

Estouro de buffer

Pilha na função echo

	end. retorno	
%ebp →	%ebp salvo	
%ebp-4 →	%ebx salvo	
%ebp-8 →	[7][6][5][4]	
%ebp-12 →	[3][2][1][0]	← buf

Caracteres lidos	Corrupção adicional
0-7	nenhuma
8-11	%ebx salvo
12-15	%ebp salvo
16-19	endereço de retorno
20+	estado salvo no registro do chamador

Estouro de Buffer

Contra-medidas

- Usar função **fgets** que inclui como argumento o número máximo de caracteres a serem lidos
- Deteção de corrupção da pilha

Ataques com uso de estouro de buffer

- Normalmente é inserida uma string que insere código de ataque mais bytes extras para corromper o endereço de retorno e desviar para o código pirata ao executar `ret`
- Em geral o código pirata executa uma chamada de sistema para iniciar um shell, dando amplos poderes ao intruso
- Em outras situações o código de ataque pode realizar tarefas indevidas, reparar a stack, executar `ret` uma segunda vez e retornar sem erro aparente
  - Worm nov 88, com uso de bufffer overflow no site remoto através do comando FINGER com uma string apropriada

Vulnerabilidades

CVE

- Para vulnerabilidades de sistema consultar Common Vulnerabilities and Exposures (CVE®) http://nvd.nist.gov/nvd.cfm?advancedsearch
  - Common Vulnerabilities and Exposures (CVE®) is a dictionary of common names (i.e., CVE Identifiers) for publicly known information security vulnerabilities, while its Common Configuration Enumeration (CCETM) provides identifiers for security configuration issues and exposures.

Inibindo ataques de estouro de buffer em Linux

Randomização da pilha

- Inserir código pirata e ponteiro para este código implica em saber o endereço em que a string será localizada
- Para sistemas rodando a mesma combinação de SO e programas, as localizações das pilhas são as mesmas, facilitando, por exemplo, ataque semelhante a vários servidores
- A idéia de randomização é variar a posição da pilha entre execuções de programas

Randomização da pilha

Implementação

- Aloca-se um espaço aleatório entre 0 e n bytes na pilha no início de um programa (pode ser usada a função `alloca`)
- Este espaço não é utilizado pelo programa, mas varia a posição da pilha entre execuções de um mesmo programa

Ação de ataque

- Para contornar a randomização, intruso pode inserir uma sequência de NOPs (*NOP sled*) (que apenas incrementam o PC, sem executar nada) antes do código de ataque
  - Necessário apenas desviar para um endereço na sequência de NOPs, para levar à execução do código malicioso
- A randomização não impede que um ataque seja bem sucedido com uso de força bruta

Randomização da pilha

Exercício

Listando 10.000 vezes o endereço de uma variável local na memória, obtém-se valores entre 0xffffb754 e 0xffffd754.

- Qual o intervalo aproximado de endereços?
  R:
- Se queremos obter buffer overrun e é usada uma seq. de NOPs com 128 bytes, quantas tentativas teriam que ser feitas para testar todos os endereços?
  R:

Randomização da pilha

Exercício

Listando 10.000 o endereço de uma variável local na memória, obtem-se valores entre 0xffffb754 e 0xffffd754.

- Qual o intervalo aproximado de endereços?
  R: 0xffffd754 - 0xffffb754 = 0x2000 $= 2x16^3 = 2^{13}$.
- Se queremos obter buffer overrun e é usada uma seq. de NOPs com 128 bytes, quantas tentativas teriam que ser feitas para testar todos os endereços?
  R:

Randomização da pilha

Exercício

Listando 10.000 o endereço de uma variável local na memória, obtem-se valores entre 0xffffb754 e 0xffffd754.

- Qual o intervalo aproximado de endereços?
  R: 0xffffd754 - 0xffffb754 = 0x2000 $= 2x16^3 = 2^{13}$.
- Se queremos obter buffer overrun e é usada uma seq. de NOPs com 128 bytes, quantas tentativas teriam que ser feitas para testar todos os endereços?
  R: A cada tentativa são testados $128 = 2^7$ *endereços.*
  Logo serão necessárias da ordem de $2^6 = 64$ *tentativas.*

Deteção de corrupção da pilha

Protetor da pilha

- Gerar um valor aleatório (*canary* ou valor de guarda) e inserir entre o buffer local e o restante do registro da pilha
- Antes de restaurar registros salvos e retornar da função, testa-se se este valor foi preservado ou não
- Se não foi preservado, retorna com erro

GCC e proteção da pilha

- GCC tenta advinhar se função é vulnerável a este tipo de ataque e insere a proteção automaticamente
- Para obter código sem o protetor é preciso executar o GCC com a opção `-fno-stack-protector`

Deteção de corrupção da pilha

Função echo com uso de gets

```
/* Le uma linha de entrada e escreve de volta */
void echo () {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

Deteção de corrupção da pilha

```
echo:
    pushl  %ebx              salva %ebx
    subl   $40, %esp         aloca 40 bytes de espação
    movl   %gs:20, %eax      prepara teste de corrupção
    movl   %eax, 28(%esp)    salva na pilha
    xorl   %eax, %eax        zera %eax
    leal   20(%esp), %ebx    buf computado como %esp+20
    movl   %ebx, (%esp)      armazena buf no topo da pilha
    call   gets              chama gets
    movl   %ebx, (%esp)      armazena buf no topo da pilha
    call   puts              chama puts
    movl   28(%esp), %eax    restaura valor do teste de corrupção
    xorl   %gs:20, %eax      compara com valor original
    je  .L2                  se for igual (não houve corrupção),
                             vai para retornar
    call   __stack_chk_fail  rotina para tratar corrupção da pilha
.L2:                         retornar
    addl   $40, %esp         libera pilha
    popl   %ebx              restaura %ebx
    ret                      retorna
```

Protetor da pilha

Canary: %gs:20

- Armazenado no offset 20 do segmento %gs (%gs:20), que é de leitura apenas (parte de arquitetura i286 antiga) e não pode ser alterado por um atacante

Rotina de inicialização do sistema operacional carrega o valor aleatório no segmento %gs

Referências bibliográficas

- *Computer Systems–A Programmer's Perspective* (**Cap.3**, seção 3.12)