MC504 - Sistemas Operacionais

Buffer Overflow

Islene Calciolari Garcia
Instituto de Computação - Unicamp
Primeiro Semestre de 2017

Sumário

Buffer Overflow

- Ataque que insere mais dados do que o espaço previamente alocado. Estes dados extras na verdade são trechos de códigos que serão executados pelo programa-vítima.
- Esta aula foi baseada na série Buffer Overflow Primer, de Vivek Ramachandran

Registradores Intel 32-bits

- `EBP`: *base pointer*
- `ESP`: *stack pointer*

- `EAX`: acumulador
- `ECX`: contador
- `EDX`: dados
- `ESI`: *source index*
- `EDI`: *destination index*

Vamos examinar a pilha de execução

- Componentes do frame (não necessariamente nesta ordem):
  - valor de retorno
  - endereço de retorno
  - registradores
  - argumentos para a função
  - variáveis locais da função
- Veja os códigos `pilha.c` e `pilha2.c`

Funções vulneráveis

- Veja `man gets()`, `man fgets()` e `man scanf()`
  - Você já utilizou `scanf("%<max>s, s");`?
- Veja o código `buffer.c`
- Insira um buffer maior do que o esperado e veja o resultado
- Veja o código `buffer-strcpy.c`

Como colocar o novo ponto de execução via entrada padrão?

- Como colocar encontrar e inserir o endereço na string?
- Veja `man printf()`

```
Endereço: 0x4005f6
$ printf "abcdef... \xf6\x05\x40\x00" | ./buffer
```

Vítima e armadilhas

Loop infinito

- **Shellcode:** Código malicioso a ser inserido tem este nome porque, na maioria das vezes, o código adicionado iria abrir uma nova shell.
- Vamos escrever armadilhas. A vítima vai colaborar nos testes executando código via `mmap`.
- Veja os códigos `map-armadilha-loop` e `map-vitima`
- Como modificar `map-armadilha-loop` para chamar uma função tipo `printf` ou outra função qualquer?

Vítima e armadilhas

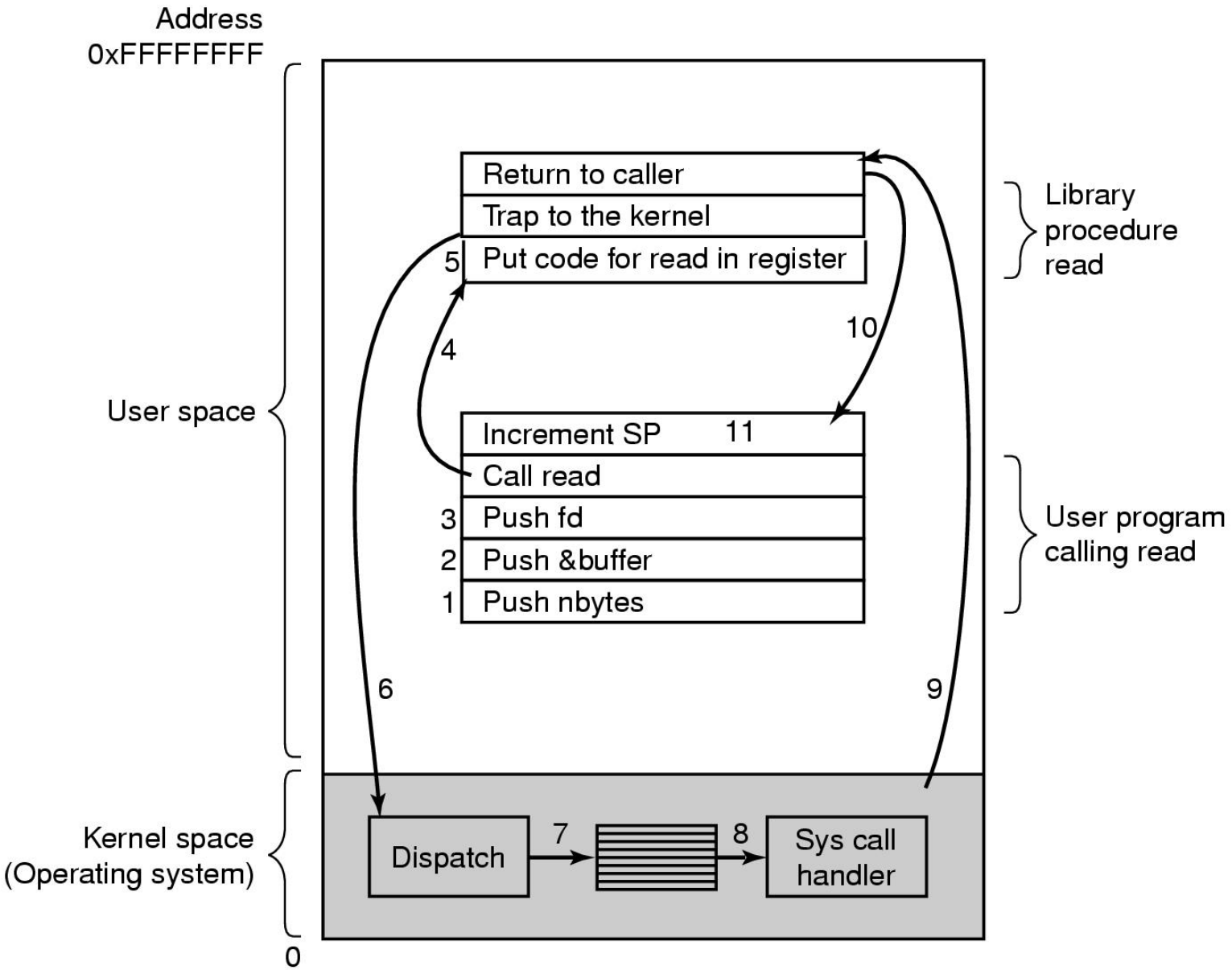
Chamada de outras funções

- Para fazer outros testes, não será preciso recompilar o código da vítima. Por quê?
- Tente colocar algo como `printf("Armadilha!!!!");` no código da vítima. Por que não funciona?
  - Veja o código `map-armadilha-printf.c`
  - No gdb, utilize `disassemble f`

Chamadas de sistema

- Fazem a fronteira entre o modo usuário e o modo kernel (protegido)
- Exemplos: fork, waitpid, execve, open, close, read, write, mkdir, link, unlink, ...
- O kernel deve ter uma tabela com as várias chamadas.

Chamada de sistema



Tanenbaum: Figura 1.17

execve

- Veja o exemplo `execve.c`

```
int main() {
  char *args[2]  = {"/bin/bash", NULL};

  execve(args[0], args, NULL);
  return 0;
}
```

Veja `map-armadilha-execve.c`

Todos os argumentos devem estar na pilha

- Argumentos
  - `filename`
  - `argv`
  - `envp`
- Veja o código `map-armadilha-execve2.c`
- Não produz um shellcode perfeito porque a string de entrada ainda contém 0s (NULLs). :(

Explorando a vítima

- Considere um shellcode sem NULLs como o disponível em demystifying execve shellcode
- Como saber em qual endereço o shellcode deve ser posicionado?
- NOP Sled antes do shell code permite a recuperação de pequenas falhas no cálculo do endereço.
- Ataque força-bruta?
- Veja o código `buffer-shellcode.c`

Proteção das páginas

- Controlar quais páginas são executáveis poderia resolver o problema?
- Return to `libc`
- Veja o código `buffer-libc.c`

Como funciona o Stack Canary

- Termo canário inspirado nos pássaros utilizados nas minas de carvão.
- Valores chave são inseridos na pilha
- Caso esses valores sejam alterados, o programa deve interromper sua execução.
- Veja as opções de compilação do gcc 6.2:

```
-fstack-protector: Proteção para funções que invocam
   alloca ou funções com buffers maiores do que 8 bytes.
-fstack-protector-all: Proteção para todas as funções
-fno-stack-protector: Desliga a proteção
```