Aula TP - 29/Abr/2019

Cada grupo deve colocar a resposta às perguntas dos seguintes exercícios na área do seu grupo no Github até ao final do dia 10/Mai/2019. Por cada dia de atraso será descontado 0,15 valores à nota desse trabalho.

Note que estes exercícios devem ser feitos na máquina virtual disponibilizada.

Exercícios

1. *Buffer Overflow*

Instruções para quem não esteja a utilizar a máquina virtual disponibilizada:

1. Para este ponto necessita de instalar o gdb e o GDB-Peda na conta do utilizador *user* na máquina virtual. Sugere-se que efetue a seguinte sequência de comandos:

```
sudo apt-get install gdb

cd ~/

git clone https://github.com/longld/peda.git ~/bin/peda

echo 'source ~/bin/peda/peda.py' | sudo tee --append ~/.gdbinit > /dev/null
```

2. Adicionalmente, grave os ficheiros na diretoria Aula 11.a para a sua máquina local na diretoria /home/user/Aulas/Aula11.a .

Experiência 1.1 - Organização da memória do programa

Utilize o comando Unix `size` para ver o tamanho, em bytes, do segmento de texto, dados e bss dos programas size1.c, size2.c, size3.c, size4.c e size5.c, seguindo os seguintes passos:

1. Compile os vários ficheiros. Ex.: `gcc -o size1 size1.c`
2. Efetue o comando `size` para cada um dos ficheiros executáveis. Ex.: `size size1`
3. Verifique e explique as diferenças de valores nos vários segmentos de texto, dados e bss.

Experiência 1.2 - Organização da memória do programa

Utilize o programa memoryLayout.c para verificar o layout da memória do programa. Verifique se os resultados são os expectáveis (de notar que as optimizações do compilador e a arquitectura da máquina podem levar a resultados ligeiramente diferentes).

Nota: Terá que compilar e executar o programa.

Experiência 1.3 - Buffer overflow em várias linguagens

Verifique o que ocorre no mesmo programa escrito em Java (LOverflow.java), Python (LOverflow.py) e C++ (LOverflow.cpp), executando-os.

Explique o comportamento dos programas.

Pergunta P1.1 - Buffer overflow em várias linguagens

Verifique o que ocorre no mesmo programa escrito em Java (LOverflow2.java), Python (LOverflow2.py) e C++ (LOverflow2.cpp), executando-os.

Explique o comportamento dos programas.

Pergunta P1.2 - Buffer overflow em várias linguagens

Verifique o que ocorre no mesmo programa escrito em Java (LOverflow3.java), Python (LOverflow3.py) e C++ (LOverflow3.cpp), executando-os.

Explique o comportamento dos programas.

Experiência 1.4 - Buffer overflow em várias linguagens

Verifique o que ocorre no mesmo programa escrito em Java (ReadTemps.java) e Python (ReadTemps.py), executando-os.

- Teste com temps.txt com os valores: 30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9
- Teste com temps.txt com os valores: 30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9 31.0 31.2
- Altere os programas para resolver o problemas SEM alterar o tamanho do array.

Pergunta P1.3 - Buffer overflow

Analise e teste os programs escritos em C RootExploit.c e 0-simple.c .

- Indique qual a vulnerabilidade de *Buffer Overflow* existente e o que tem de fazer (e porquê) para a explorar e (i) obter a confirmação de que lhe foram atribuídas permissões de root/admin, sem utilizar a *password* correta, (ii) obter a mensagem "YOU WIN!!!".

Experiência 1.5 - Formatted I/O

A formatação do I/O, em C, pode fazer toda a diferença.

Analise e teste o programa IOformatado.c e veja qual a diferença entre a função segura() e vulneravel().

Nota: Relembre-se que em C existem diretivas de formatação, tais como %d, %s, ...

Pergunta P1.4 - Read overflow

Analise e teste o program escrito em C ReadOverflow.c .

- O que pode concluir?

Experiência 1.6 - Buffer overflow na Heap

O exemplo de *Buffer overflow* na *heap* visto na aula teórica, encontra-se nos ficheiros overflowHeap.1.c e overflowHeap.2.c.

- Teste e altere o output da variável *readonly*.

Experiência 1.7 - Buffer overflow na Stack

As vulnerabilidades de buffer overflow ocorrem quando é possível escrever e/ou executar código em áreas de memória que normalmente não seria possível, e deriva usualmente da utilização de funções não seguras.

A maioria dos sistemas operativos e compiladores actuais já incorporam algumas características de segurança que previnem esse tipo de ataques, tais como:

- *canaries* (ou canários) - valores colocados na *stack* (pelo compilador) entre um buffer e dados de controlo, de modo a monitorizar *buffer overflows* durante a execução do programa. Se houver um *buffer overflow*, o primeiro dado a ser corrompido/alterado é o canário, e uma verificação falhada dos dados do canário são um alerta para a existência de um *overflow*;
- *Executable space protection* (XP) / *Data Execution Prevention* (DEP) - impede que alguns sectores de memória (por exemplo, a *stack*) possam ser executados (i.e., o programa não consegue executar código que estiver nesses sectores de memória);
- *Address Space Layout Randomisation* (ASLR) - técnica utilizada para impedir que seja executado código injetado no programa, através da colocação randomizada de módulos e estruturas de dados no espaço da memória.

Esta experiência e as próximas perguntas vão necessitar do ASLR desativado, pelo que execute o seguinte comando:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

O exemplo de *Buffer overflow* na *stack* visto na aula teórica, encontra-se nos ficheiros overflowStack.1.c e overflowStack.2.c.

Vamos utilizar o Gnu debugger (GDB) para obtermos a informação necessária para explorar a vulnerabilidade, sem alterar o primeiro programa overflowStack.1.c.

Para isso execute os seguintes comandos:

- compilar overflowStack.1.c com informação de debug, a ser utilizada pelo GDB;

```
gcc overflowStack.1.c -g -o overflowStack.1
```

- iniciar o debug do programa com o gdb

```
gdb overflowStack.1
```

- no gdb, colocar *breakpoint* na função store()

```
b store
```

- no gdb, executar o programa até ao *breakpoint*

```
r
```

- no gdb, obtenha informação sobre a *frame* actual de execução do programa

```
info f
```

- na última linha é-lhe indicado em que endereço de memória está guardado o base pointer (%rbp) e o endereço de retorno (%rip). Como vamos querer reescrever o conteúdo do endereço de retorno, de modo a direcioná-lo para a função debug(), guarde o endereço de memória em que está guardado o %rip (emr).
- o próximo passo é obter o endereço de memória onde está guardada a função debug(). Para isso, no gdb efetue o comando

```
p debug
```

- aponte o endereço que lhe é apresentado, já que é esse endereço que quer reescrever no emr.
- de seguida, para ter a certeza quantos bytes tem de escrever no array buf para reescrever o %rip, veja em que endereço se inicia o array buf (eia)

```
p &buf
```

- O número de bytes a escrever até ao inicio do emr é de emr - eia. Utilize um calculador hexadecimal para obter o resultado (por exemplo, utilize http://www.calculator.net/hex-calculator.html)
- já tem todos os dados que necessita do gdb, pelo que pode sair do gdb com

```
quit
```

- Execute o programa overflowStack.1 com o parâmetro necessário para obter os dados escondidos na função debug().

Pergunta P1.5

Agora que já tem experiência em efetuar o *overflow* a um *buffer* (cf. pergunta P1.3), consegue fazer o mesmo se for necessário um valor exato?

Compile e execute o programa 1-match.c, e obtenha a mensagem de "Congratulations" no ecrã. Notas:

- já ouviu falar de *little-endian* e *big-endian*?

Indique os passos que efetuou para explorar esta vulnerabilidade.

Experiência 1.8

Compile e execute o programa 2-functions.c, e obtenha a mensagem de "Congratulations" no ecrã. Notas:

- relembre que o nome de uma função em C equivale ao endereço onde esta é escrita em memória;
- poderá fp ser win?

Indique os passos que efetuou para explorar a vulnerabilidade.

Experiência 1.9

Compile e execute o programa 3-return.c, e obtenha a mensagem de "Congratulations" no ecrã. Notas:

Indique os passos que efetuou para explorar a vulnerabilidade.

Nota final: Para voltar a ativar o ASLR que desativou na Experiência 1.7, execute o seguinte comando:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

---

2. Vulnerabilidade de inteiros

Grave os ficheiros na diretoria Aula 11.c para a sua máquina local na diretoria /home/user/Aulas/Aula11.c .

Experiência 2.1

Utilize o programa IntegerCheck.java para verificar os valores máximos e mínimos dos vários inteiros na sua máquina.

- Quantos bits tem cada um dos inteiros?

Experiência 2.2

Teste o programa IntegerError.java e verifique o que ocorre em situação de *integer overflow*.

- Teste com outros tipos de inteiro: byte, short, long
- E se precisar de utilizar inteiros maiores?

Experiência 2.3

Teste o programa IntegerCheck2.java e insira valores correctos e incorrectos.

- O que acontece quando lê um long e o atribuí a outro tipo de inteiro?
- O que acontecia se utilizasse o scan para o tipo correcto em cada um dos casos (nextByte(), nextInt(), ...)?

Pergunta P2.1

Analise o programa overflow.c.

1. Qual a vulnerabilidade que existe na função *vulneravel()* e quais os efeitos da mesma?
2. Complete o *main()* de modo a demonstrar essa vulnerabilidade.
3. Ao executar dá algum erro? Qual?

Pergunta P2.2

Analise o programa underflow.c.

1. Qual a vulnerabilidade que existe na função *vulneravel()* e quais os efeitos da mesma?
2. Complete o *main()* de modo a demonstrar essa vulnerabilidade.
3. Ao executar dá algum erro? Qual?

Experiência 2.4

Analise o programa erro_sinal.c.

1. Qual a vulnerabilidade que existe na função *vulneravel()* e quais os efeitos da mesma?
2. Complete o *main()* de modo a demonstrar essa vulnerabilidade.
3. Ao executar dá algum erro? Qual?