Escola Superior de Tecnologia e Gestão
Instituto Politécnico da Guarda

Escola Superior de Tecnologia e Gestão
Instituto Politécnico da Guarda

PROGRAMAÇÃO E SEGURANÇA
BUFFER OVERFLOW PROTECTION

José Carlos Fonseca, 2018

OBJECTIVOS

1. Programar em C ou C++ seguindo as boas práticas de forma a evitar problemas de Buffer Overflow
2. Definir formas de combater o Buffer Overflow
3. Explicar as protecções dos SO e dos compiladores para combater o Buffer Overflow e as suas limitações
4. Utilizar o gcc para compilar um programa
5. Utilizar o gdb para fazer debug a um programa

EVITAR O BUFFER OVERFLOW

- Verificar todos os limites e condições fronteira na utilização de Buffers
  - Auditar os locais onde são feitos os cálculos dos valores de alocação dos Buffers
  - Verificar as condições de fim dos ciclos
- Evitar usar as funções que não validam os limites dos Buffers
  - Usar o STL (Standard Template Library) Strings do C++
  - Substituir os Buffers de Strings do C pelas Strings do C++
- Usar analisadores de código, tais como Coverity, PREfast, e Klocwork
- Usar linguagens de programação de mais alto nível que validem os limites dos buffers e impeçam o acesso directo à memória como o C#, Java, etc. em vez de C, C++ ou Assembly

EVITAR O BUFFER OVERFLOW

Função de manipulação de strings	Como usar com segurança
strcpy char *strcpy(char *strDestination, const char *strSource);	É muito difícil de usar com segurança. Verificar se o buffer fonte não é nulo e se o tamanho do buffer de destino é maior que o tamanho do buffer de origem e se o buffer de origem termina com nulo
strcat char * strcat (char * destination, const char * source);	É muito difícil de usar com segurança, tal como o strcpy.
strncpy char *strncpy(char *strDest, const char *strSource, size_t *count*);	Verificar se o buffer fonte não é nulo e se não é um ponteiro ilegal, e calcular correctamente o size com sizeof(buf)-1. Não esquecer de usar o -1 pois caso contrário poderá permitir a exploração do EBP com o ataque off-by-one
strncat char * strncat (char * destination, char * source, size_t num);	É difícil de usar, pois o tamanho especificado é o número de caracteres a serem concatenados e não o tamanho final do buffer.

EVITAR O BUFFER OVERFLOW

Escola Superior de Tecnologia e Gestão
Instituto Politécnico da Guarda

Função de manipulação de strings	Como usar com segurança
sprintf int sprintf(char *buffer, const char *format [, *argument]* ...);	É muito difícil de usar com segurança. As strings têm de ser terminadas com nulo, o tamanho tem de ser correcto, as formatações dos caracteres têm de estar correctas e não ultrapassarem o tamanho máximo permitido, etc.
_snprintf int _snprintf(char *buffer, size_t *count,* const char *format [, *argument]* ...);	O buffer tem de terminar em nulo. Como não é uma função standard as diferentes implementações podem dar resultados diferentes. Não esquecer de usar o sizeof(buf)-1 pois caso contrário poderá permitir a exploração do EBP com o ataque off-by-one. É preferível usar esta função para fazer concatenações em vez das strcat e strncat.
gets char *gets(char *buffer);	É muito difícil, senão impossível, de usar com segurança. Deve-se usar o fgets ou um objecto stream do C++

PROTECÇÕES PARA BUFFER OVERFLOW

- Stack não executável
- Address Space Layout Randomization (ASLR)
- Canário
  - ProPolice - gcc
  - /GS - Windows
  - Stackguard - gcc

PROTECÇÕES PARA BUFFER OVERFLOW

- **Stack não executável** – Não se conseguem executar instruções existentes no Stack
  - Evita a grande maioria dos ataques que requerem a execução de código que é colocado no Stack
  - Consegue-se contornar mandando executar funções conhecidas, nomeadamente da biblioteca libc que é carregada em quase todos os programas, ou colocando o código no Heap
    - O System() é uma das funções que podem ser usadas. Pode ser chamado com o parâmetro /bin/sh para executar a Shell. Se for seguida de exit(), executa a Shell e sai sem erros

PROTECÇÕES PARA BUFFER OVERFLOW

- **Address Space Layout Randomization (ASLR)** – O SO altera a localização de certas partes do espaço virtual de endereçamento (virtual address space) cada vez que é invocada uma aplicação. Isto inclui o código da aplicação, as bibliotecas, o Stack e o Heap
- Não sabendo o local para onde uma subrotina deve saltar torna muito difícil explorar o Buffer Overflow

PROTECÇÕES PARA BUFFER OVERFLOW

- Address Space Layout Randomization (ASLR)
  - Para que um ataque tenha sucesso o hacker tem de conseguir saber essa localização. Se conseguir injectar na memória executável um número significativo do NOPs, bastará um endereço aproximado para conseguir os seus intentos
  - Há código cuja localização não pode ser aleatória
  - Há formas de obter as tabelas de alocação
  - Pode-se explorar com o ret2reg (retorno para registo)

PROTECÇÕES PARA BUFFER OVERFLOW

Canário

- Nas minas de carvão eram usados canários para detectar se o ar era respirável, já que os canários morrem mais rapidamente do que os humanos na presença de gases nocivos como metano e o monóxido de carbono
- A morte do canário era um aviso para os mineiros abandonarem rapidamente o local onde se encontravam

PROTECÇÕES PARA BUFFER OVERFLOW

- **Canário** – É um valor pseudo-aleatório que é verificado se sofreu alterações quando a rotina termina a sua execução. Tradicionalmente é colocado entre o RET e o EBP e impede a exploração do RET, mas não do EBP. As implementações modernas colocam o Canário após o EBP e não após o RET protegendo também o EBP.
  - ProPolice - gcc
  - /GS - Windows
  - Stackguard - gcc

Memória baixa	Buffer	16	Buffer
	↓		
	ebp	4	Onde é guardado o EBP
	Canary	4	Canário
	ret	4	EIP de retorno
Memória alta	*str	4	Argumento *str

PROTECÇÕES PARA BUFFER OVERFLOW

- **Canário**, formas de o explorar

Forma de explorar	O que é
Pointer subterfuge	Alterar um ponteiro local para posteriormente colocar dados nessa localização
Register attack	Alterar o valor armazenado num registo (como o EBP) para ganhar controlo sobre ele
Vtable hijacking	Alterar um ponteiro local de um objecto de modo a que uma chamada para o vtable execute um payload
Exception handler clobbering	Alterar um registo da excepção para desviar o handler para o payload
index ou of range	Explorar um índice de um array cujos limites não são verificados
Heap overruns	Buffer Overflow no heap

GCC – ALGUMAS OPÇÕES

Escola Superior de Tecnologia e Gestão
Instituto Politécnico da Guarda

Comando	Função
`-fno-stack-protector`	Não usar Canário
`-z execstack`	Permitir que o Stack seja executável
`-mpreferred-stack-boundary=n`	Para colocar o stack boundary a 2^n, já que o gcc aloca um espaço múltiplo do seu valor a cada variável. Nós usamos o valor n=2, pelo que dá 2^2=4
`-ggdb`	Gerar informação de debug
`-static`	Para que o código das funções incluídas nos #include seja incluído no código fonte final, em vez de ficar só uma referência a esse código

DESACTIVAR AS PROTECÇÕES RECENTES

- Para fazermos os nossos exemplos, simplificamos o problema desactivando as protecções recentes e definimos algumas variáveis. Para tal vamos:
  - Usar o **`-static`** no gcc para que o código das funções dos #include seja incluído no código fonte final, em vez de ficar só uma referência a esse código
  - Usar o **`-fno-stack-protector`** no gcc para desactivar o ProPolice
  - Usar o **`-mpreferred-stack-boundary=2`** no gcc para colocar o stack boundary a 4, já que o gcc aloca um espaço múltiplo do seu valor do stack boundary a cada variável
  - Usar o **`-z execstack`** para que o stack possa ser executável. O bit NX impede que certas regiões da memória sejam executáveis e o Stack é uma delas
- Desactivar o ASLR no Ubuntu executando como root:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

- Activar o core dump em caso de crash do programa executando como user:

```
ulimit -c unlimited
```

GDB – ALGUMAS OPÇÕES

Comando	Função
`-q`	(--quiet) opção da linha de comandos. Não mostra o número da versão quando o gdb é iniciado
`--core core`	Para fazer o debug do core dump. Mosta nomeadamente o valor do endereço de retorno onde se deu o crash do programa
`disas main`	(disassemble) desassembla o main (ou outra qualquer função) do programa e mostra o seu código máquina. Com o modificador /m mostra também o código fonte, se existir informação. Com o modificador /r mostra o assembly
`b *0x08048588`	(break) coloca um breakpoint no endereço 0x08048588 do programa. Colocando o nome de uma função coloca um breakpoint no início da função, por exemplo: break main. Colocando o número da linha do programa, coloca um breakpoint nessa linha, por exemplo para a linha 5 fazer: break 5.
`del break`	(delete breakpoints) elimina os breakpoints criados

GDB – ALGUMAS OPÇÕES

Comando	Função
`r `perl -e "print 'A'x30";``	(run) executa o programa com o parâmetro `perl -e "print 'A'x30";`. Este parâmetro executa um código em perl que mostra 30 As
`attach 4978`	(attach) anexa um processo que está a correr de modo a poder fazer-se o debug deste
`x/20xw $esp`	Mostra o conteúdo de 20 posições de memória (de 4 bytes cada) em hexadecimal a iniciar em $esp. Como tem um w vai mostrar words. Se tivesse um b iria mostrar bytes. Caso não tenha nenhum deles mostra o resultado com o formato usado da última vez
`x/20d 0xbffff620`	Mostra o conteúdo de 20 posições de memória (de 4 bytes cada) em decimal a iniciar em 0xbffff620
`x/20i $eip`	Mostra o conteúdo de 20 posições de memória desassembladas a partir do endereço $eip

GDB – ALGUMAS OPÇÕES

Comando	Função
`mai i sec`	(maintenance info sections) mostra as secções de memória alocadas ao programa (.data, .text, .bss, etc.)
`cont`	(continue) continua a execução do programa
`q`	(quit) sai do gdb
`p/d 0xc`	(print) mostra o valor 0xc em decimal. Podia-se ter usado o /x para hexadecimal, ou o /t para binário
`p/t $esp`	(print) mostra o valor de $esp em binário. Podia-se ter usado o /x para hexadecimal, ou o /d para decimal
`p system`	(print) mostra a localização em memória da função system(). Também usamos este comando para saber a localização de outras funções como o exit().
`display`	Mostra expressões sempre que o programa para. Por exemplo display/i $pc para mostrar o Program Counter (instrução que vai ser executada)

GDB – ALGUMAS OPÇÕES

Comando	Função
`find $esp, 0xbfffffff, "/bin/bash"`	(find) procura na memória a partir de $esp até 0xbfffffff a string "/bin/bash"
`l main`	(list) Lista o código do main do programa. Também se pode listar o programa a partir de uma dada linha, por exemplo a 5, com: list 5
`stepi`	(stepi) Executa a instrução de código máquina seguinte
`step`	(step) Executa a instrução de código do programa seguinte
`i r ebp eip`	(info registers) mostra o conteúdo dos registos ebp e eip. Se não se colocarem parâmetros mosta o conteúdo de todos os registos
`set *(char*)0x08048e3a = 0x74`	(set) modificar directamente um byte que se encontra na memória. Neste caso a instrução de um programa.

BIBLIOGRAFIA

- Aleph One (Elias Levy) (1996), Smashing the stack for fun and profit, Phrack Magazine
- Chris Anley, John Heasman, Felix Linder, Gerardo Richarte, (2007), The Shellcoder's Handbook, Wiley Publishing, Inc.
- Michael Howard, David LeBlanc, (2003), Writing Secure Code, 2nd edition, Microsoft Press
- Michael Howard, David LeBlanc, (2005), 19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them, McGraw-Hill/Osborne
- Dethy (2000), http://biblio.l0t3k.net/b0f/en/htce.txt
- Wenliang Du, (2010), Return-to-libc Attack Lab, http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Return_to_libc/Return_to_libc.pdf
- Tenouk, http://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html
- Securiteam, Using GDB for Vulnerability Developement, http://www.securiteam.com/securityreviews/5OP0B2KCKI.html