Versão: A Nota mínima: **7.5/20 valores** / Duração: 120 minutos

Número:_____ **Nome:**_____

Responda aos grupos II, III, IV e V em folhas A4 separadas.

[8v] **Grupo I - Assinale no seguinte grupo se as frases são verdadeiras ou falsas (uma resposta errada desconta 50% de uma correta).**

V F

1) Em C, se tivermos uma variável "`short x = 0x1234;`" o valor "`-x`" também pode ser obtido através de "`~x + 1`" ☐☐

2) Em C, admita as variáveis "`char *a;`" e "`long *b;`". A comparação "`if(sizeof(a) == sizeof(b))`" é verdadeira ☐☐

3) Em C, admita as variáveis "`int x=0x01234567;`" e "`short *ptr=(short*)&x;`". Logo, "`*ptr`" equivale ao valor `0x4567` ☐☐

4) Em C, o maior valor positivo que é possível atribuir a uma variável do tipo `unsigned int` é $2^{32} - 1$ ☐☐

5) Em C, o operador lógico && (AND) termina a avaliação da expressão se encontrar uma condição que seja avaliada como verdade ☐☐

6) Em C, admita a variável "`char *ptr;`". A expressão "`(int*)ptr + 7`" avança, relativamente a `ptr`, 28 bytes na memória ☐☐

7) Em C, tentar reservar um bloco na *heap* pode falhar mesmo existindo um total de memória livre superior ao tamanho desse bloco ☐☐

8) Em C, podemos redimensionar com `realloc` o tamanho de um vetor de inteiros declarado estaticamente como "`int vec[20];`" ☐☐

9) Em x86-64, a *stack* é sempre usada na gestão da invocação de funções, independentemente do número dos seus parâmetros ☐☐

10) Em x86-64, "`idivq %rcx`" efetua a divisão (com sinal) entre `%rax` e `%rcx` colocando o quociente em `%rax` e o resto em `%rdx` ☐☐

11) Em x86-64, à semelhança das operações de deslocamento de bits, as operações de rotação também perdem os bits da informação original ☐☐

12) Em x86-64, a instrução "`leaq (%rax,%rax,4),%rax`" pode ser usada para multiplicar por quatro o valor presente em `%rax` ☐☐

13) Em x86-64, o registo `%rax` é local a cada uma das funções, o que dispensa qualquer cuidado no seu uso entre invocações de funções ☐☐

14) Em x86-64, admita que o valor de `%rsp` é `0x1008`. A execução da instrução `ret` coloca o valor de `%rsp` em `0x1010` ☐☐

15) Em x86-64, reservar 8 bytes para variáveis locais de uma função pode ser conseguido através de "`addq $8, %rsp`" ☐☐

16) Em x86-64, o endereço inicial de uma estrutura sujeita a alinhamento depende dos tipos de dados dos seus campos ☐☐

17) Em x86-64, imediatamente após o prólogo de uma função, o valor anterior de `%rbp` pode ser obtido através de `(%rsp)` ☐☐

18) Em x86-64, a adição de dois bytes 0x3A e 0x1C deixa as *flags* do registo EFLAGS com os valores ZF=1, SF=1, CF=0, OF=0 ☐☐

19) Na hierarquia de memória, à medida que nos afastamos do CPU abdicamos da performance em favor do custo por byte ☐☐

20) A possibilidade de existirem várias referências para a mesma posição de memória em C dificulta as otimizações efetuadas pelo compilador ☐☐

[3v] **Grupo II – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

[1,5v] **a)** Admita os seguintes endereços, conteúdo da memória e registos. Que valor (em hexadecimal) é armazenado em `%rax` em cada uma das seguintes instruções? **Justifique as suas respostas**.

Endereço	Conteúdo
0x8000	0x5
0x8004	0xA
0x8008	0xF

Registo	Conteúdo
%rdx	0x8000
%rbx	2

```
leaq (%rdx), %rax
movl (%rdx), %eax
leaq 4(%rdx), %rax
movl 4(%rdx), %eax
leaq (%rdx, %rbx, 4), %rax
movl (%rdx, %rbx, 4), %eax
```

[1,5v] **b)** Admita o seguinte excerto de código em C. Implemente a função `func1` em Assembly.

```
void xpto(int *p1, int p2);
```

```
void func1(int a, int b, int c) {
   xpto(&b, a+c);
}
```

[3v] **Grupo III – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

Considere as seguintes declarações:

```
typedef struct {
    short code;
    int   start;
    char  raw[3];
    long  data;
} OldSensor;
```

```
typedef struct {
    short code;
    short start;
    char  raw[5];
    short sense;
    int   ext;
    long  data;
} NewSensor;
```

[1,5v] **a)** Indique o alinhamento dos campos de uma estrutura do tipo `OldSensor`. Indique claramente, para cada campo, o seu endereço, bem como as partes alocadas, mas não usadas, para satisfazer as restrições de alinhamento. Indique o tamanho total da estrutura. **Admita que a estrutura está colocada a partir do endereço `0x100`.**

[1,5v] **b)** Considere o seguinte fragmento de código em C, respeitando as declarações das estruturas apresentadas acima.

```
void xpto(OldSensor *oldData){
   NewSensor *newData;

   /* zeros out all the space of oldData */
   bzero((void *)oldData,sizeof(OldSensor));

   oldData->code   = 0x104f;
   oldData->start  = 0x80501ab8;
   oldData->raw[0] = 0xe1;
   oldData->raw[1] = 0xe2;
   oldData->raw[2] = 0x8f;
   oldData->data   = 15;

   newData = (NewSensor *) oldData;
   ...
 }
```

Admita que após estas linhas de código começamos a aceder aos campos da estrutura `NewSensor` através da variável `newData`. Indique, em hexadecimal, o valor de cada um dos campos de `newData` indicados a seguir. Tenha em atenção a ordenação dos bytes em memória em Linux/x86-64!

a) `newData->code` = 0x_____
b) `newData->raw[0]` = 0x_____
c) `newData->raw[2]` = 0x_____
d) `newData->raw[4]` = 0x_____
e) `newData->sense` = 0x_____

[3v] **Grupo IV – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

Considere a seguinte declaração:

```
struct test{
  short *p;
  typedef struct {
    short x;
    short y;
  }s;
  struct test *next;
};
```

Considerando o seguinte código em Assembly gerado pelo compilador para para a função `st_init`, preencha os espaços em branco da mesma função em C (**escreva a função completa na folha A4**).

```
st_init:
  movw 8(%rdi), %ax
  movw %ax, 10(%rdi)
  leaq 10(%rdi), %rax
  movq %rax, (%rdi)
  movq %rdi, 16(%rdi)
  ret

void st_init(struct test *st){
  st->s.y  = ___________;
  st->p    = ___________;
  st->next = ___________;
}
```

[3v] **Grupo V – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

Admita o seguinte excerto de código em C. A função `calc_hash` recebe como primeiro parâmetro o endereço de uma estrutura onde está armazenado o endereço de um vetor de *strings* (`strs`), assim como o número de *strings* armazenadas nesse vetor (`num`). A função recebe como segundo parâmetro o endereço de um inteiro `hash` onde é armazenado o resultado computado.

```
typedef struct{
  int num;
  char **strs;
}data_t;

void calc_hash(data_t *src, int *hash){
 int i, j;
 *hash = 0;

 for(i = 0; i< get_num(src); i++)
  for(j= 0; j < strlen(src->strs[i]); j++)
   *hash += secret(src->strs[i],j) + strlen(src->strs[i])/2;
}
```

```
int get_num(data_t *src){
  return src->num;
}

int secret(char *str, int pos){
  return str[pos] % 26;
}
```

Apresente uma segunda versão da função `calc_hash` em C com a mesma funcionalidade, mas melhor desempenho. Admita que o compilador que é usado não efetua nenhuma otimização. Deve, por isso, indicar todas as otimizações possíveis. **Indique claramente cada uma das otimizações usadas sob a forma de comentário no código.**