Introdução à Segurança da Informação e sua Relação com o Gerenciamento de Memória em Python

A segurança da informação é um pilar fundamental no desenvolvimento de software, garantindo a confidencialidade, integridade e disponibilidade dos dados. Um dos maiores desafios nessa área é a prevenção de vulnerabilidades relacionadas ao gerenciamento de memória, como buffer overflow, use-after-free, integer overflow e race conditions, que frequentemente exploram falhas em linguagens de baixo nível, como C e C++. Essas vulnerabilidades podem permitir a execução de código malicioso, vazamento de informações ou até mesmo a tomada de controle de sistemas.

Neste estudo, temos o objetivo de mostrar, como a implementação das estruturas de dados podem privilegiar segurança ou desempenho ou um equilíbrio entre os dois. Através de uma exploração em linguagens como o Python, uma linguagem de alto nível, e como ela mitiga essas ameaças através de técnicas como o Overallocation e o gerenciamento automático de memória, além de otimizar o desempenho de estruturas dados como listas. Ao analisar a implementação de estruturas dados como listas, destacamos mecanismos internos — como a função list_resize, a qual, foi realizada um processo de dessecagem e engenharia reversa detalhado, para mostrar como a mesma garante uma alocação segura e alinhada, evitando estouros e fragmentação, preservando a eficiência — que eliminam riscos como corrupção de memória e acesso indevido. Além disso, comparamos o comportamento seguro de Python com cenários vulneráveis em outras linguagens de baixo nível como C — que tem como prioridade o desempenho — demonstrando como escolhas de design (ex: tipagem dinâmica, coleta de lixo) transformam otimizações de desempenho em barreiras de segurança. Tudo isso foi demonstrado através de gráficos, tabelas, dados de debug e analise de código fonte do interpretador CPython.

Por fim, discutimos Trade-offs entre Desempenho e Segurança no Gerenciamento de Memória, vinculando esses conceitos aos princípios da segurança da informação, e a importância do conhecimento sobre as estruturas de dados no desenvolvimento de código seguro, eficiente e adequados as especificações.

- Todos os códigos, imagens, dados e gráficos estão disponíveis no repositório: https://github.com/michel-sudo/estudy-py-memory-manager
- Foi utilizado software de debug C GDB para debug do código fonte do interpretador CPython.
- Bibliotecas Python: Ctypes para ter acesso a estrutura interna de lista em Python, Matplotlib Para plotagem de gráficos, Timeit para analise de tempo.
- Código do interpretador CPython analisado: https://github.com/python/cpython

(Este estudo será focado na técnica de Overallocation em listas baseadas em arrays, enquantos outras estruturas como HashSet e HashMap, utilzam tabelas hash para armazenar elementos, mas também implementam o conceito de redimensionamento dinâmico, porém com suas próprias implementações.)

Overallocation

Gerenciamento de memória

A rotina de **Overallocation** em Python está relacionada ao gerenciamento de memória dinâmica de estruturas de dados como listas. Em Python, as listas são implementadas como arrays dinâmicos, o que significa que elas podem crescer ou diminuir conforme necessário. Para otimizar o desempenho, o Python usa uma técnica chamada **Overallocation** (ou "sobre-alocação") para evitar realocações frequentes de memória à medida que a lista cresce, garantindo desempenho **O(n)** amortizado para uma sequencia longa de adições, além da prevenção contra o transbordamento de buffer.

1. Análise de alocação de memória

Em outras linguagens como C, a adição de elementos fora dos limites de um array não é verificado pelo compilador ou runtime. Isso pode levar a comportamentos indefinidos, como sobrescrever regiões de memória adjacentes, o que é a base para explorações de **buffer overflow**. Na Tabela 1.0 e no gráfico da figura 1.1, podemos analisar uma relação entre adição de elementos, redimensionamento do array e tamanho em bytes incluindo o **Overhead** da estrutura interna usada pelo CPython (Interpretador do Python) para representar listas em Python.

Exemplo de código em python que gera Overallocation:

```
lista = []

for i in range(20):
    lista.append(i)
```

Figura 1.0

Tabela de redimensionamento do array de interno da lista:

Elementos (qtde)	Array (Slots)	Lista (Bytes)	Elementos (qtde)	Array (Slots)	Lista (Bytes)	Elementos (qtde)	Array (Slots)	Lista (Bytes)
0	0	40	7	8	104	14	16	168
1	4	72	8	8	104	15	16	168
2	4	72	9	16	168	16	16	168
3	4	72	10	16	168	17	24	232
4	4	72	11	16	168	18	24	232
5	8	104	12	16	168	19	24	232
6	8	104	13	16	168	20	24	232

Tabela 1.0

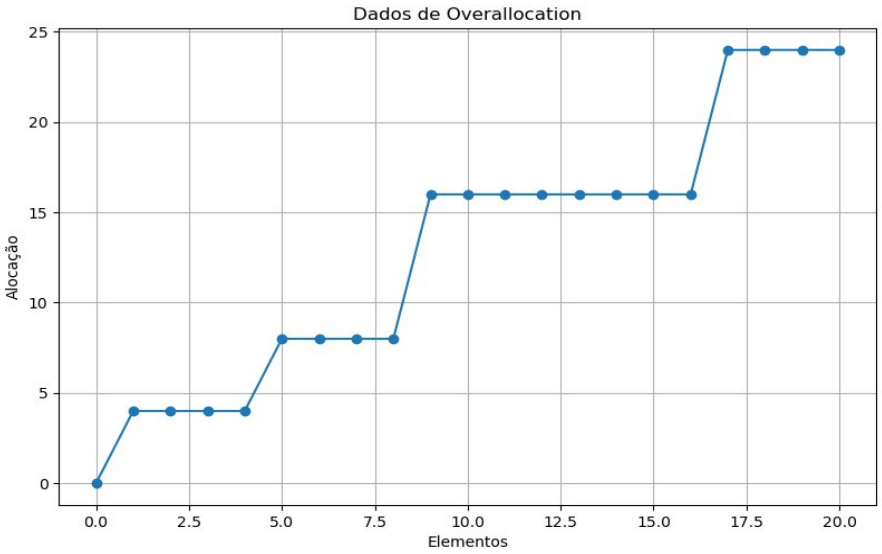


Figura 1.1

Vale ressaltar que em Python, não há um forma oficial ou direta para manipular arrays ou acessar a capacidade do array alocado para os elementos de uma lista, porque esse é um detalhe interno da implementação do CPython, que mais a frente será detalhado, na seção 1.1. No entanto, você pode usar uma combinação de técnicas para acessar indiretamente esse valor, que está no campo **allocated** pertencente ao objeto do tipo **PyListObject** (estrutura interna usada pelo CPython para representar listas em Python). Como mostra na figura 1.2, o código utilizado para obter esses valores, utilizando a biblioteca **ctypes** para acessar a estrutura interna de uma lista.

```python
import ctypes
import sys

# Estrutura de uma lista no CPython (simplificada)
class PyListObject(ctypes.Structure):
    _fields_ = [
        ("ob_refcnt", ctypes.c_ssize_t),
        ("ob_type", ctypes.py_object),
        ("ob_size", ctypes.c_ssize_t),
        ("ob_item", ctypes.POINTER(ctypes.py_object)),
        ("allocated", ctypes.c_ssize_t),
    ]

# Obtém o endereço da lista
lista = []
list_address = id(lista)

# Converte o endereço para um ponteiro da estrutura PyListObject
list_struct = ctypes.cast(list_address, ctypes.POINTER(PyListObject)).contents

# Acessa o campo 'allocated'
print(len(lista), list_struct.allocated)

for i in range(20):
    lista.append(i)
    list_address = id(lista)
    list_struct = ctypes.cast(list_address, ctypes.POINTER(PyListObject)).contents
    print(len(lista), list_struct.allocated)
```

Figura 1.2

2. Análise tempo de alocação

Foi realizado um teste para medir o tempo de execução de dois algorítimos de inserção de elementos em Python, onde um insere valores inteiros em uma lista vazia, e outro que insere esses mesmos elementos em uma lista já pre-alocada. Onde foi utilizado a biblioteca **timeit** para medir o tempo de cada algorítimo. Como pode ser visto no gráfico da figura 1.3, a linha verde trata-se da algorítimo de inserção com a lista vazia, ou seja, realizando Overallocation. Ambos os algorítimos e o código para medição do tempo, estão na figura 1.4. A linha azul trata-se do algorítimo com a lista já pre-alocada com valores do tipo '**none**'.

Figura 1.3

No primeiro algorítimo com o nome de overallocation (linha verde), a cada **append** pode adicionar o redimensionamento dinâmico, através da função **list_resize**, que será explicado em detalhes mais adiante, esse redimensionamento envolve, alocar um novo bloco de memória, copiar os elementos existentes e liberar o bloco antigo, isso consome tempo extra, especialmente quando a lista cresce muito, como é visto no gráfico quando o algorítimo é executado para a adição de 2^{22} ou 4.194.304 de elementos, o algorítimo leva cerca 289,4586 segundos para completar a execução, ou aproximadamente 69 microssegundos para cada adição. As duas curvas tem um crescimento exponencial, mas a linha de pré-alocação (linha azul), é visível na figura 1.3, que mantém um tempo sistematicamente menor do que a função Overallocation, pois a memória já está reservada, então a operação de atribuição (**lista[i] = valor**) é direta e **O(1)** constante, sem overhead de realocação. É possível observar também que para pequenas entradas que o a diferença de tempo é mínima, porém para grandes entradas entradas como 2^{22} a razão de tempo entre a função de Overallocation e pre-alocation chega ser ~1,87x, ou seja a função de Overallocation chega a ser 87% lento, mas a razão parece se estabilizar em ~2x para N grandes,

confirmando que ambos são O(n), mas com uma diferença em relação a uma constante multiplicativa, pois a inserção com Overallocation, tem O(n) amortizado porque, embora alguns appends sejam O(n) (durante redimensionamento), a média é O(1) por operação, oque torna a inserção com Overallocation mais lenta porém ainda O(n).

As tabelas 1.1 e 1.2, mostram detalhadamente cada execução dos algorítimos e a relação entre o numero de elementos inseridos, tempo médio de uma única adição e tempo total da execução do algorítimo. Logo em seguida na figura 1.4 possui o código das funções e de análise de tempo.

Algorítimo Overallocation

Elementos (qtde)	Único (Micro S)	Lista (S)	Elementos (qtde)	Único (Micro S)	Lista (S)	Elementos (qtde)	Único (Micro S)	Lista (S)
2^0	200,0	0,0002	2^8	24,6	0,0063	2^{16}	37,3	2,4454
2^1	150,0	0,0003	2^9	31,2	0,0160	2^{17}	50,9	6,6740
2^2	75,0	0,0003	2^{10}	34,4	0,0353	2^{18}	56,0	14,6976
2^3	50,0	0,0004	2^{11}	35,2	0,0721	2^{19}	59,5	31,2394
2^4	37,5	0,0006	2^{12}	36,0	0,1478	2^{20}	61,2	64,2059
2^5	31,2	0,0010	2^{13}	37,7	0,3094	2^{21}	62,0	130,0952
2^6	28,1	0,0018	2^{14}	39,4	0,6469	2^{22}	69,0	289,4586
2^7	28,1	0,0036	2^{15}	41,0	1,3459			

Tabela 1.1

Algorítimo Pre-alocation

Elementos (qtde)	Único (Micro S)	Lista (S)	Elementos (qtde)	Único (Micro S)	Lista (S)	Elementos (qtde)	Único (Micro S)	Lista (S)
2^0	200,0	0,0002	2^8	18,3	0,0047	2^{16}	36,3	2,3823
2^1	100,0	0,0002	2^9	27,1	0,0139	2^{17}	36,4	4,7789
2^2	75,0	0,0003	2^{10}	31,0	0,0318	2^{18}	36,3	9,5373
2^3	37,5	0,0003	2^{11}	33,2	0,0680	2^{19}	37,0	19,4414
2^4	31,2	0,0005	2^{12}	34,5	0,1414	2^{20}	36,9	38,7875
2^5	25,0	0,0008	2^{13}	35,3	0,2892	2^{21}	36,8	77,2264
2^6	21,8	0,0014	2^{14}	35,0	0,5748	2^{22}	36,7	154,1526
2^7	19,5	0,0025	2^{15}	35,8	1,1754			

Tabela 1.2

src > codigos-python > analiseTempoOverallocation.py > ...

```python
import timeit

# Função de inserção de elementos sem pre-alocação;
def overallocation(r):
    lista = []
    for i in range(r):
        lista.append(i)

# Função de inserção de elementos com a a lista pre-alocada
def pre_allocation(lista, r):
    for i in range(r):
        lista[i] = i

# Mede o tempo aproximado em segundos da inserção n de elementos em uma lista
for i in range(23):
    r = 2 ** i
    lista_pre = [None] * r
    time_overallocation = timeit.timeit(lambda: overallocation(r), number=1000)
    time_pre_allocation = timeit.timeit(lambda: pre_allocation(lista_pre, r), number=1000)
    print(f"{time_overallocation:.4f} {time_pre_allocation:.4f}")

```

Figura 1.4

3. Análise da segurança

Vulnerabilidades

/Falha de Buffer Overflow em C

Buffer Overflow é uma vulnerabilidade clássica em linguagens de baixo nível (como C/C++), onde dados escritos em um buffer ultrapassam seus limites alocados, corrompendo memória adjacente. Isso pode sobrescrever endereços de retorno (causando crashes ou execução de código malicioso). Vazar informações sensíveis (como senhas armazenadas em memória próxima). Ser explorado para ataques como Stack Smashing e Heartbleed. Na figura 1.5 está um exemplo de código vulnerável.

src > codigos-c > C buffer_overflow.c > bufferOverflow()

```c
#include <string.h>
#include <stdio.h>

// Função vulneravel a buffer overflow
void bufferOverflow() {
    char str[] = "Mundo";
    char buffer[3];

    printf("vetor antes: %s\n", str); // Saída: "Mundo"
    printf("buffer antes: %s\n", buffer); // Saída: ""

    strcpy(buffer, str); // É feito a cópia do conteúsdo do vetor str para o vetor buffer

    printf("vetor depois: %s\n", str); // Saída: "do"
    printf("buffer depois: %s\n", buffer); // Saída: "Mundo"

```

Figura 1.5

Através do Debuger de C (GDB), foi feito uma analise da do código da Figura 1.5 que copia- através da função **strcpy** -um vetor de caracteres **str (**“Mundo”**)** para um outro vetor **buffer** (“”) de tamanho 3, que estão localizados de forma adjacente entre si na memória. Logo é possível ver nas figuras 1.6 e 1.7, os dados nas posições 0, 1, 2 nas respectivas posições 0x7fffffffdd02, 0x7fffffffdd03 e 0x7fffffffdd04 do vetor **str** (de tamanho 6) sendo sobrescritos, com os dados que não “couberam” no vetor **buffer** (de tamanho 3).

Antes do strcpy

```
11      strcpy(buffer, str);
(gdb) print &buffer
$1 = (char (*)[3]) 0x7fffffffdcff
(gdb) x/3sb buffer
0x7fffffffdcff: ""
0x7fffffffdd00: ""
0x7fffffffdd01: ""

(gdb) print &str
$2 = (char (*)[6]) 0x7fffffffdd02
(gdb) x/6sb str
0x7fffffffdd02: "Mundo"

(gdb) print &str[0]
$3 = 0x7fffffffdd02 "Mundo"
(gdb) print &str[1]
$4 = 0x7fffffffdd03 "undo"
(gdb) print &str[2]
$5 = 0x7fffffffdd04 "ndo"
(gdb) print &str[3]
$6 = 0x7fffffffdd05 "do"
(gdb) print &str[4]
$7 = 0x7fffffffdd06 "o"
(gdb) print &str[5]
$8 = 0x7fffffffdd07 ""
```

Figura 1.6

Depois do strcpy

```
(gdb) x/3sb buffer
0x7fffffffdcff: "Mundo"
0x7fffffffdd05: "do"
0x7fffffffdd08: ""

(gdb) print &buffer[0]
$9 = 0x7fffffffdcff "Mundo"
(gdb) print &buffer[1]
$10 = 0x7fffffffdd00 "undo"
(gdb) print &buffer[2]

(gdb) print &buffer
$15 = (char (*)[3]) 0x7fffffffdcff

(gdb) print &str
$16 = (char (*)[6]) 0x7fffffffdd02

(gdb) print &str[0]
$17 = 0x7fffffffdd02 "do"
(gdb) print &str[1]
$18 = 0x7fffffffdd03 "o"
(gdb) print &str[2]
$19 = 0x7fffffffdd04 ""
(gdb) print &str[3]
$20 = 0x7fffffffdd05 "do"
(gdb) print &str[4]
$21 = 0x7fffffffdd06 "o"
(gdb) print &str[5]
$22 = 0x7fffffffdd07 ""
```

Figura 1.7

	buffer			str					
Antes	""	""	""	M	u	n	d	o	""
Depois	M	u	n	d	o	""	d	o	""

Figura 1.8

/Falha de Buffer Overflow em Python

Em Python, não existe uma função exatamente como **strcpy** do C que copia cegamente o conteúdo de uma string (ou lista) para um buffer sem verificar limites, o que pode causar um buffer overflow. No entanto, existem várias maneiras de copiar listas ou strings em Python, cada uma com comportamentos diferentes. Aqui estão as principais abordagens:

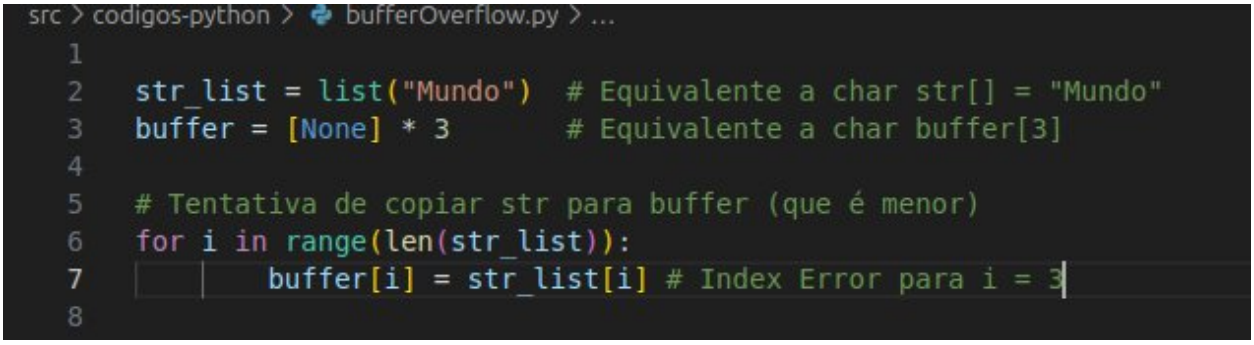


Figura 1.9

Na figura 1.9, na tentativa de copiar cada elemento de uma lista para outra, Python se protege e lança uma exceção de índice inválido, pois a lista buffer não possui memória alocada para o indice 3.

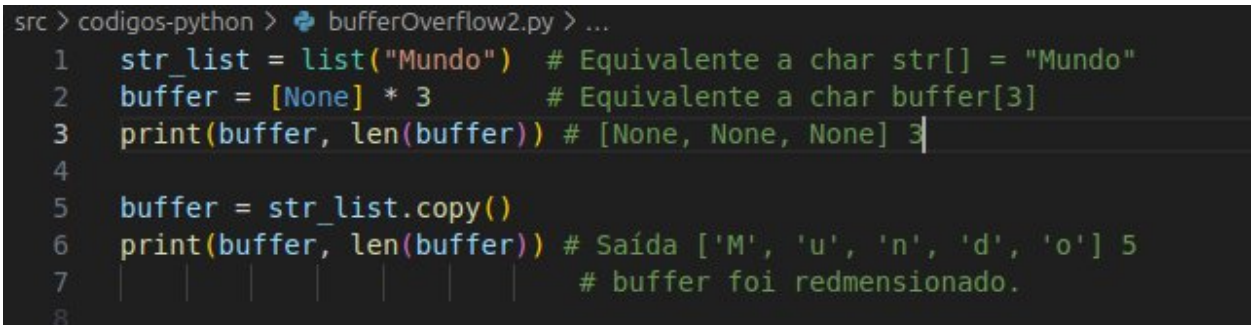
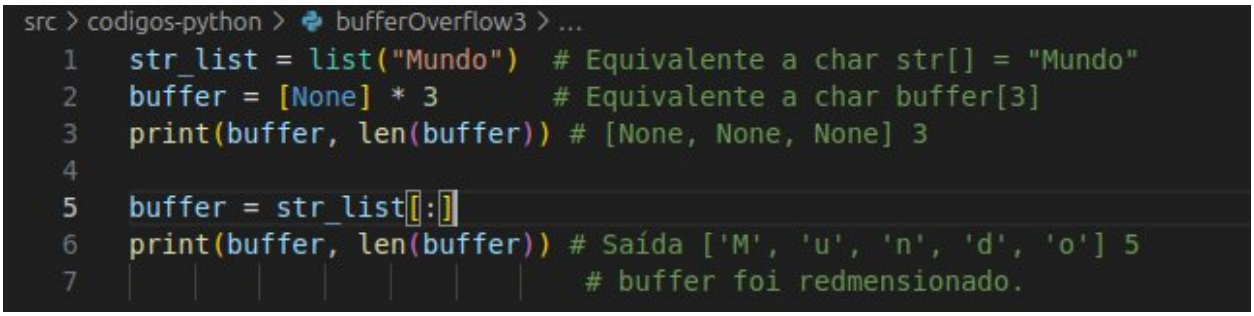


Figura 2.0

Na figura 2.0, utilizando o método **copy()**, Python redimensiona a lista a qual recebe a cópia.



Figurar 2.1

Na figura 2.1, utilizando o método de **slice,** Python também redimensiona a lista de destino.

4. Análise da Função list_resize:

Agora vamos analisar o código da função **list_resize** no CPython, que está definida no código-fonte do Cpython, em (cpython/Objects/listobjects.c). Ela é responsável por redimensionar uma lista (list) quando necessário. Essa função é crucial para o funcionamento eficiente e seguro das listas em Python, especialmente durante operações como append, insert, ou extend.

Na figura 1.3 podemos ver na primeira linha, a assinatura da função e os seus parâmetros. O parâmetro ***self** é um ponteiro que aponta para um objeto do tipo **PyListObject,** neste caso, a lista que está sendo redimensionada, enquanto o **newsize** é o novo tamanho desejado para a lista, do tipo **Py_ssize_t**. A variável local **new_allocated** do tipo **size_t,** é o novo tamanho que será alocado para a lista após o redimensionamento, e a **allocated** do tipo **Py_ssize_t**, refere-se ao tamanho já alocado no campo **allocated** do objeto de lista **self.**

```
104    static int
105    list_resize(PyListObject *self, Py_ssize_t newsize)
106    {
107        size_t new_allocated, target_bytes;
108        Py_ssize_t allocated = self->allocated;
```

Figura 2.2

/ Estrutura PyListObject:

A estrutura **PyListObject** é definida no código-fonte do CPython em (Include/cpython /listobject.h). Ela contém os seguintes campos principais:

1. PyObject_VAR_HEAD: É um cabeçalho padrão para objetos de tamanho variável em Python. Contém campos como **ob_size**, que armazena o número de elementos atualmente na lista (equivalente a **len(lista)** em Python).

2. ob_item: É um ponteiro para um array de ponteiros (**PyObject** **), onde cada elemento do array é um ponteiro para um objeto Python (os elementos da lista). Esse array é alocado dinamicamente e redimensionado conforme necessário.

3. allocated: É um campo do tipo **Py_ssize_t** que armazena o número de slots de memória alocados para a lista. Esse valor representa a capacidade atual da lista, ou seja, quantos elementos a lista pode armazenar sem precisar realocar memória.

/ Tipo Py_ssize_t:

Py_ssize_t, é um tipo de dado inteiro com sinal (**signed integer**) definido no código-fonte do CPython. Ele é tipicamente um **typedef** para um tipo de dado nativo da plataforma, como **ssize_t**. Ele é amplamente utilizado em várias partes da implementação do CPython para representar tamanhos, índices e contagens de elementos em estruturas de dados como listas, tuplas, strings, etc. Ele ajuda a

evitar problemas de estouro de buffer **(buffer overflow)** e outros problemas relacionados ao uso incorreto de tipos de dados inteiros, principalmente em sistemas de 64 bits. A definição exata pode variar dependendo da plataforma e do compilador, mas geralmente é um inteiro de 32 ou 64 bits, dependendo da arquitetura do sistema.

a) Tamanho Adequado para Índices e Contagens:

Em sistemas de 64 bits, o espaço de endereçamento de memória é muito maior do que em sistemas de 32 bits. Isso significa que objetos Python (como listas, strings, etc.) podem ter um número muito maior de elementos. Se um tipo de dado inadequado (como int, que geralmente tem 32 bits) fosse usado para representar índices ou tamanhos, ele poderia **transbordar** (**overflow**) ao tentar representar valores muito grandes. Por exemplo, em uma lista com mais de 2 bilhões de elementos (o limite de um int de 32 bits com sinal), o valor retornado seria incorreto. O **Py_ssize_t** é definido como um tipo de dado com sinal e com tamanho suficiente para acomodar o maior índice ou tamanho possível em uma plataforma específica. Em sistemas de 64 bits, ele geralmente é um inteiro de 64 bits, o que permite representar valores muito maiores sem risco de **overflow**.

b) Consistência com o Tamanho de Ponteiros:

Além disso, os ponteiros (endereços de memória) têm 64 bits de tamanho. Se um tipo de dado menor (como int de 32 bits) fosse usado para indexar ou contar elementos em uma estrutura de dados, ele poderia não ser capaz de cobrir todo o espaço de endereçamento disponível. Dessa forma, o **Py_ssize_t** é definido para ter o mesmo tamanho que os ponteiros na plataforma em que o CPython está sendo executado. Isso garante que ele possa ser usado de forma segura para indexar qualquer posição de memória válida.

c) Prevenção de Comportamentos Inesperados:

O uso de tipos de dados inadequados pode levar a comportamentos inesperados, como o **Estouro de buffer,** se um valor de índice ou tamanho for maior do que o tipo de dado pode suportar, ele pode "dar a volta" (**wrap around**) e resultar em um valor negativo ou incorreto. Isso pode causar acesso a regiões de memória inválidas, levando a falhas de segmentação (**segmentation faults**) ou corrupção de memória. **Falsos negativos** também podem ocorrer**,** em operações de comparação, um valor que transbordou pode ser interpretado como negativo, causando erros lógicos no código, como mostrado na figura 1.4, se esse valor fosse usado para acessar uma lista ou alocar memória, o resultado seria catastrófico.

```
int len = 2147483647; // Valor máximo para um int de 32 bits com sinal
len += 1; // Estouro de buffer: len agora é -2147483648 (comportamento indefinido)
```

Figura 2.3

```
Py_ssize_t len = 2147483647;
len += 1; // len agora é 2147483648 (comportamento correto)
```

Figura 2.4

/ Verificação de Redimensionamento necessário:

No próximo trecho do código da função **list_resize** (figura 1.6), o código verifica se o tamanho atualmente alocado (**allocated**) é suficiente para acomodar o novo tamanho (**newsize**), e se **allocated >= newsize** e **newsize** for pelo menos metade de **allocated**, o redimensionamento é evitado. Isso ocorre porque a lista já tem espaço suficiente para o novo tamanho, e não há necessidade de realocar memória, para aumentar ou diminuir a capacidade. Em seguida é verificado através de um **assert**, se o ponteiro **ob_item** não é nulo (**NULL**) **ou** se o novo tamanho (**newsize**) é zero. Se **newsize** for zero, é permitido que **ob_item** seja **NULL**, pois isso significa que o objeto está sendo redimensionado para um tamanho zero (ou seja, está sendo esvaziado). Na proxima linha, **Py_SET_SIZE(self, newsize)** define o tamanho do objeto **self** para o novo tamanho **newsize**, ou seja, atualiza o campo **ob_size** presente no cabeçalho do objeto de lista **self,** para refletir o novo tamanho. O **return 0;** Retorna o valor 0, que geralmente indica sucesso em funções que retornam um código de erro. Neste caso, o 0 significa que o redimensionamento foi bem-sucedido e não houve necessidade de realocar memória.

```
114         if (allocated >= newsize && newsize >= (allocated >> 1)) {
115             assert(self->ob_item != NULL || newsize == 0);
116             Py_SET_SIZE(self, newsize);
117             return 0;
118         }
```

Figura 2.5

Resumo dos cenários

Cenário	allocated	newsize	If	comportamento
allocated > newsize newsize >= metade	10	7	true	Atualiza tamanho para 7, retorna 0.
newsize < metade	10	4	false	Necessário realocar memória
newsize = allocated	10	10	true	Atualiza tamanho para 10, retorna 0
newsize > allocated	10	15	false	Necessário realocar memória
newsize == 0	10	0	false	Necessário realocar memória
self->ob_item == null	10	8	true (assert false)	AssertError (newsize != 0)

Tabela 1.3

/ Cálculo do Novo Tamanho de Alocação:

Se o redimensionamento for necessário, o código calcula o novo tamanho alocado (**new_allocated**), (figura 1.7).

```
130         new_allocated = ((size_t)newsize + (newsize >> 3) + 6) & ~(size_t)3;
131         /* Do not overallocate if the new size is closer to overallocated size
132          * than to the old size.
133          */
```

Figura 2.6

A formula **(newsize + (newsize >> 3) + 6)** adiciona uma margem de super-alocação ao novo tamanho, onde **(newsize >> 3)** é um deslocamento a direita de 3 bits, ou seja, (**newsize >> 3**) = (**newsize / 2^3**) = (**newsize / 8**) = **12,5%** do tamanho atual. O valor **6** é um **padding** (preenchimento) adicional, para garantir que a lista tenha espaço suficiente para crescer. O operador **& ~(size_t)3** arredonda o valor para o múltiplo de 4 mais próximo e menor que o valor. Isso é feito para alinhar a memória e melhorar a eficiência. Assim forma-se um padrão de crescimento como: 0, 4, 8, 16, 24, 32, 40, 52, 64, 76,(...).

Exemplo:

```
new_allocated = ((size_t)5 + (5 >> 3) + 6) & ~(size_t)3;
              = (5 + 0 + 6) & ~3;
              = 11 & ~3;
              = 8;
```

Figura 2.7

Importante: O **new_allocated** também não transbordará, pois o tipo **size_t**, assegura o tamanho máximo possível de endereçamento. A super-alocação é moderada, mas é suficiente para dar comportamento amortizado de tempo linear **O(n)**, sobre uma sequência longa de **appends()**. Logo em seguida é feita uma verificação se o crescimento necessário é maior do que a alocação extra calculada, como mostra na figura 1.9.

```
131         /* Do not overallocate if the new size is closer to overallocated size
132          * than to the old size.
133          */
134         if (newsize - Py_SIZE(self) > (Py_ssize_t)(new_allocated - newsize))
135             new_allocated = ((size_t)newsize + 3) & ~(size_t)3;
```

Figura 2.8

O código acompanha o comentário “ ***Não superaloque se o novo tamanho (newsize) estiver mais próximo do tamanho superalocado (new_allocated) do que do tamanho antigo (Py_SIZE(self))***”. Isso significa que o código irá verificar se a super-alocação é realmente justificável, com intuito de evitar alocar memória extra desnecessariamente em certos casos. Dessa forma **(newsize - Py_SIZE(self))** representa o crescimento necessário, ou a quantidade de elementos a mais que se pretende adicionar, onde **newsize** é o novo tamanho desejado, e **Py_SIZE(self)** corresponde ao número atual de elementos presentes na lista. **(Py_ssize_t)(new_allocated – newsize)** representa a sobre-alocação extra calculada,

ou a quantidade de slots de memoria que serão alocados a mais do que a quantidade necessária para armazenar o **newsize**, onde **new_allocated** é a memória total calculada. Em seguida é feito um cast para o tipo **Py_ssize_t**.

Se condição for verdadeira, ou seja, se a diferença entre o novo tamanho e o tamanho atual for maior que a diferença entre a capacidade alocada e o novo tamanho, significa que o newsize está mais distante do tamanho atual (**Py_SIZE(self)**) do que a capacidade alocada (**new_allocated**). Nesse caso, o Python decide não usar a capacidade alocada atual como base e, em vez disso, ajusta **new_allocated** para um valor mais próximo de **newsize**, evitando alocação excessiva.

Em seguida é feita uma verificação se o **newsize** é igual a 0, nesse caso o código define **new_allocated** como 0. Isso ocorre quando a lista está sendo esvaziada. Como mostra na figura 2.0.

```
137         if (newsize == 0)
138             new_allocated = 0;
139
```

Figura 2.9

A função **ensure_shared_on_resize** é chamada para garantir que a lista compartilhe memória corretamente com outras estruturas, se necessário. Isso é importante para otimizar o uso de memória em cenários onde múltiplas listas compartilham os mesmos dados.

```
140         ensure_shared_on_resize(self);
141
142     #ifdef Py_GIL_DISABLED
```

Figura 3.0

Na linha 142, o código se divide em duas partes por uma condição, dependendo de se o **GIL (Global Interpreter Lock)** está desabilitado ou não. O **GIL (Global Interpreter Lock)** é um mecanismo no CPython (a implementação padrão do Python) que garante que apenas uma thread execute código Python por vez. Ele é essencialmente um mutex (ou lock) que protege o interpretador Python, evitando que múltiplas threads executem código Python simultaneamente. Isso é necessário porque o gerenciamento de memória do CPython não é thread-safe (ou seja, não é seguro para execução concorrente). Neste caso, vamos considerar o GIL habilitado, que é o padrão atual de implementação do CPython. Logo o código segue para a linha 167.

```
167     #else
168         PyObject **items;
169         if (new_allocated <= (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
170             target_bytes = new_allocated * sizeof(PyObject *);
171             items = (PyObject **)PyMem_Realloc(self->ob_item, target_bytes);
172         }
```

Figura 3.1

Na linha 167 **items** é uma variável temporária que será usada para armazenar o ponteiro para o novo bloco de memória realocado para um array de ponteiros de objetos Python (**PyObject ***). Em seguida é feita uma verificação de de estouro de inteiro, onde **PY_SSIZE_T_MAX** é o valor máximo de um **Py_ssize_t**, em sistemas de 64 bits, por exemplo, é geralmente 2^{63} - 1. O código verifica se **new_allocated** é pequeno o suficiente para que o total de bytes necessários não cause um estouro de inteiro ao exceder **PY_SSIZE_T_MAX.** Dessa forma **(size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)** calcula o número máximo de elementos que podem ser alocados sem exceder o limite de **Py_ssize_t** em bytes, e verifica se é maior ou igual a nova alocação (**new_allocated**).

Se essa condição for verdadeira, segue para a próxima linha, onde o código calcula o tamanho em bytes necessários para armazenar a nova alocação (**new_allocated**) em ponteiros **PyObject** *. Em seguida é chamada a função **PyMem_Realloc**, responsável pelo gerenciamento de memória, semelhante ao **realloc** da biblioteca C padrão, mas otimizada para o interpretador. **PyMem_Realloc(self->ob_item, target_bytes)** realoca o bloco de memória apontado para **self->ob_item** parar o novo tamanho **target_bytes**. O resultado então é atribuído a **items**.

O código segue para linha 181, onde é feito atualização do objeto (figura 2.3). Onde **self->ob_item = items** atualiza o ponteiro do array interno da lista para o novo bloco de memória realocado. Na linha seguinte **Py_SET_SIZE(self, newsize)** define o novo tamanho da lista (**newsize**), ou seja, atualiza o campo **ob_size**, que indica quantos elementos a lista contém atualmente. Logo após **self->allocated = new_allocated**, atualiza a capacidade alocada da lista para refletir o novo valor.

```
181         self->ob_item = items;
182         Py_SET_SIZE(self, newsize);
183         self->allocated = new_allocated;
184     #endif
185         return 0;
186     }
187
```

Figura 3.2

Caso a condição da linha 169 seja falsa, ou seja, se o total de bytes excede o **Py_SSIZE_T_MAX** o código não aloca memória, e atribui **items = null**. Em seguida define uma exceção Python de “sem memória” no interpretador. Por fim retorna -1 para indicar falha na função, e não segue para a atualização do objeto.

```
173         else {
174             // integer overflow
175             items = NULL;
176         }
177         if (items == NULL) {
178             PyErr_NoMemory();
179             return -1;
180         }
```

Figura 3.3

/Depuração do interpretador do CPython

Também foi utilizado o **GDB** (software de debug em C) para coleta de dados e análise dos comportamentos das funções internas do CPython. Foi introduzido um **breakpoint** na função **list_resize()**, e sondas nas referências **new_allocated**, **allocated**, **target_bytes** e **newsize**. Abaixo está um trecho da execução. O documento de texto com o terminal completo da depuração está presente no repositório do estudo.
github.com/michel-sudo/estudy-py-memory-manager/data/terminalRotinaDeAlocacaoListaGDB.txt

```
Breakpoint 1, list_resize (self=self@entry=0x7ffff7587f20, newsize=newsize@entry=1) at
Objects/listobject.c:106
106     {
1: new_allocated = <optimized out>
2: allocated = <optimized out>
3: target_bytes = <optimized out>
4: newsize = 1
(gdb) n
108         Py_ssize_t allocated = self->allocated;
1: new_allocated = <optimized out>
2: allocated = <optimized out>
3: target_bytes = <optimized out>
4: newsize = 1
(gdb) n
114         if (allocated >= newsize && newsize >= (allocated >> 1)) {
1: new_allocated = <optimized out>
2: allocated = 0
3: target_bytes = <optimized out>
4: newsize = 1
(gdb) n
130         new_allocated = ((size_t)newsize + (newsize >> 3) + 6) & ~(size_t)3;
1: new_allocated = <optimized out>
2: allocated = 0
3: target_bytes = <optimized out>
4: newsize = 1
(gdb) n
134         if (newsize - Py_SIZE(self) > (Py_ssize_t)(new_allocated - newsize))
1: new_allocated = 4
2: allocated = 0
3: target_bytes = <optimized out>
4: newsize = 1
(...)
```

5. Discussão: Trade-offs entre Desempenho e Segurança no Gerenciamento de Memória

O gerenciamento de memória em linguagens de programação envolve um equilíbrio delicado entre desempenho e segurança. Nosso estudo sobre Overallocation em Python e a comparação com vulnerabilidades como buffer overflow em C ilustra claramente esse trade-off. Abaixo, discutimos os principais pontos dessa relação e suas implicações práticas.

/ Desempenho: O Custo da Flexibilidade

Overallocation em Python:

- **Vantagem:** Permite que listas cresçam dinamicamente com operações append() em tempo O(1) amortizado, garantindo eficiência em cenários onde o tamanho final é desconhecido.
- **Custo:** A alocação de memória extra (ex: ~12.5% a mais) e os redimensionamentos periódicos introduzem um overhead de tempo. Seus dados mostraram que listas pré-alocadas são até 1.87x mais rápidas para grandes volumes de dados (289.45s vs 154.15s para 2^{22} elementos).

Pré-alocação:

- **Vantagem:** Elimina o custo de redimensionamento, oferecendo tempo O(n) exato. Ideal quando o tamanho final é conhecido.
- **Limitação:** Requer conhecimento prévio do tamanho e pode levar a subutilização de memória se a alocação for superestimada.

Exemplo Prático:

Em aplicações de processamento de grandes datasets (ex: análise científica), a pré-alocação pode reduzir tempos de execução pela metade. Porém, em cenários interativos (ex: construção incremental de listas), o Overallocation é mais flexível.

/ Segurança: A Proteção Contra Vulnerabilidades

Python (Alto Nível):

- **Verificação de limites:** Operações como lista[100] em uma lista de tamanho 10 geram IndexError, impedindo acesso a memória inválida.
- **Gerenciamento automático:** Overallocation e funções como list_resize garantem que a memória seja alocada e liberada de forma controlada, evitando:
    - **Buffer overflow:** Não há como escrever além dos limites da lista.

C (Baixo Nível):

- **Performance bruta:** Operações são mais rápidas (sem verificações de limites ou alocação dinâmica oculta).

- **Riscos:** Um erro no cálculo de índices ou alocação manual pode levar a:
    - Corrupção de memória.
    - Exploits como stack smashing ou arbitrary code execution.

Exemplo de Vulnerabilidade:

No nosso estudo, um claro exemplo é o código em C da figura 1.5, vulnerável a buffer overflow

/ Trade-off Fundamental: Flexibilidade vs. Controle

Critério	Python (Overallocation)	C (Alocação Manual)
Desempenho	Overhead de ~2x (nossos dados)	Máxima eficiência
Segurança	Proteção integrada (ex: IndexError, Tipagem dinâmica, GIL)	Nenhuma verificação automática
Complexidade	Gerenciamento automático	Controle total (e risco total)
Casos de Uso	Aplicações genéricas, rápidas para prototipagem	Sistemas embarcados, kernels

Tabela 1.4

/ Quando Priorizar Desempenho ou Segurança?

Opte por Python (e Overallocation) quando:

- A segurança é crítica (ex: aplicações web, manipulação de dados sensíveis).
- O tamanho dos dados é imprevisível (ex: leitura de arquivos dinâmicos).
- A produtividade do desenvolvedor é prioritária.

Opte por pré-alocação ou linguagens de baixo nível quando:

- O tamanho dos dados é conhecido e fixo (ex: processamento de imagens).
- A performance é absolutamente crítica (ex: sistemas em tempo real).
- O ambiente controlado reduz riscos (ex: firmware de dispositivos).

Em resumo, o trade-off entre desempenho e segurança não é uma escolha binária, mas uma escala móvel que deve ser ajustada conforme os requisitos do projeto. Nosso trabalho ilustra como Python e C navegam esse espectro, e como as estrutura de dados, podem ser implementadas privilegiando segurança sem ignorar totalmente a eficiência.