Niektórzy będą importować jedną, a niektórzy tysiąc. W językach bez zarządzania pamięcią musisz sprawdzić liczbę importowanych transakcji, a następnie przydzielić im odpowiednią ilość pamięci. W przypadku gdy programista nie bierze pod uwagę tego, ile pamięci potrzebuje aplikacja, pojawiają się takie błędy, jak przepełnienie bufora.

Szukanie i exploitowanie podatności pamięci jest na tyle skomplikowane, że na sam ten temat zostały napisane całe książki. Z tego powodu ten rozdział jest jedynie wstępem do tematu omawiającym tylko dwa z wielu innych rodzajów podatności pamięci: przepełnienie bufora oraz odczyt poza granicami bufora. Jeśli jesteś zainteresowany dalszym zgłębianiem wiedzy w tym zakresie, polecam przeczytać *Hacking: The Art of Exploitation* autorstwa Jona Ericksona bądź *A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security* napisaną przez Tobiasa Kleina.

Przepełnienie bufora

Przepełnienie bufora następuje tam, gdzie aplikacja zapisuje dane, których jest zbyt wiele dla przypisanej im pamięci. Przepełnienie bufora prowadzi w najlepszym przypadku do nieprzewidywalnego zachowania aplikacji, a w najgorszym do poważnej podatności bezpieczeństwa. Kiedy atakujący może kontrolować przepełnienie w celu wykonania własnego kodu, jest on w stanie narazić aplikację bądź nawet, zależnie od uprawnień użytkownika, cały serwer. Ten typ podatności jest podobny do przykładów RCE z rozdziału 12.

Przepełnienie bufora najczęściej pojawia się wtedy, gdy programista zapomni sprawdzić wielkość zapisywanych danych do zmiennej. Zdarzają się również pomyłki w obliczeniach co do ilości pamięci, jakiej wymagają dane. Ponieważ tego rodzaju błędy zdarzają się na wiele sposobów, zbadamy tylko jeden typ – *pominięcie kontroli długości*. W języku C pominięcie kontroli długości zazwyczaj wiąże się z użyciem funkcji, które modyfikują pamięć, takich jak `strcpy()` i `memcpy()`. Ten typ błędu może pojawić się również tam, gdzie programiści używają funkcji alokacji pamięci, takich jak `malloc()` lub `calloc()`. Funkcja `strcpy()` (oraz `memcpy()`) przyjmuje dwa parametry: bufor, do którego mają zostać skopiowane dane, oraz dane do skopiowania. Oto przykład w języku C:

```
#include <string.h>
int main()
{
❶ char src[16]="hello world";
❷ char dest[16];
❸ strcpy(dest, src);
❹ printf("src is %s\n", src);
  printf("dest is %s\n", dest);
  return 0;
}
```

W tym przykładzie `scr` ❶ przyjmuje ciąg `"hello world"`, który ma 11 znaków długości, wliczając w to spację. Ten kod przydziela 16 bajtów dla `src` oraz `dest` ❷ (każdy znak zajmuje 1 bajt). Ponieważ każdy znak wymaga 1 bajta pamięci, a ciąg znaków musi się kończyć znakiem null (`\0`), ciąg `"hello world"` wymaga w sumie 12 bajtów, co oczywiście mieści się w granicach przydzielonych 16 bajtów. Funkcja `strcpy()` kopiuje ciąg z `src` do `dest` ❸. Sformułowania `printf` w ❹ wyświetlają poniższy tekst:

```
src is hello world
dest is hello world
```

Powyższy kod działa tak, jakbyśmy tego oczekiwali. Co jeśli jednak ktoś chciałby wyraźnie zaakcentować to powitanie? Rozważmy następujący przykład:

```
#include <string.h>
#include <stdio.h>
int main()
{
❶ char src[17]="hello world!!!!!";
❷ char dest[16];
❸ strcpy(dest, src);
  printf("src is %s\n", src);
  printf("dest is %s\n", dest);
  return 0;
}
```

Mamy tutaj pięć dodatkowych wykrzykników, co zwiększa sumę znaków w ciągu do 16. Programista pamiętał, że wszystkie ciągi w języku C muszą być zakończone znakiem null (\0). Przydzielił 17 bajtów do `src` ❶, jednakże zapomniał zrobić to samo dla `dest` ❷. Po skompilowaniu i uruchomieniu programu oczom programisty ukazuje się następujące wyjście:

```
src is
dest is hello world!!!!!
```

Zmienna `src` jest pusta mimo przypisania do niej `'hello world!!!!!'`. Dzieje się to z powodu tego, w jaki sposób C alokuje *pamięć na stosie*. Adresy pamięci na stosie są przydzielane rosnąco, tak więc zmienna zdefiniowana w programie wcześniej będzie mieć niższy adres pamięci od tej zdefiniowanej później. W tym przypadku `src` jest dodany do stosu pierwszy, a dopiero za nim `dest`. Kiedy następuje przepełnienie, 17 znaków dla `'hello world!!!!!!'` zostaje zapisane w zmiennej `dest`, jednak znak null (`\0`) zostaje wypchnięty do pierwszego znaku zmiennej `src`. Ponieważ znaki null oznaczają koniec ciągu, `src` ukazuje się jako pusty.