# VULNERABILITĂȚI CAUZATE DE MANIPULAREA ȘIRURILOR DE CARACTERE ÎN C SAU C++

Șirurile de caractere din surse cum ar fi argumentele din linia de comandă, consolă, fișiere text sau variabile de mediu reprezintă una din preocupările programării sigure deoarece reprezintă mijlocul prin care o intrare externă poate influența comportamentul și ieșirea unui program. Prin urmare, erorile frecvente în manipularea string-urilor au rezultat într-o gamă largă de vulnerabilități și exploatări software.

Deși șirurile de caractere sunt un concept fundamental în ingineria software, nu sunt un tip de date primitiv în C sau C ++, fiind implementate sub forma unor vectori de caractere terminate cu un caracter nul. Biblioteca C standard acceptă șiruri de caractere de tip char, wchar\_t, uchar8\_t, uchar16\_t sau uchar32\_t. În programele C++ mai poate fi folosită și clasa std::string.

## VULNERABILITĂȚI DE DIMENSIONARE ALE ȘIRURILOR DE CARACTERE

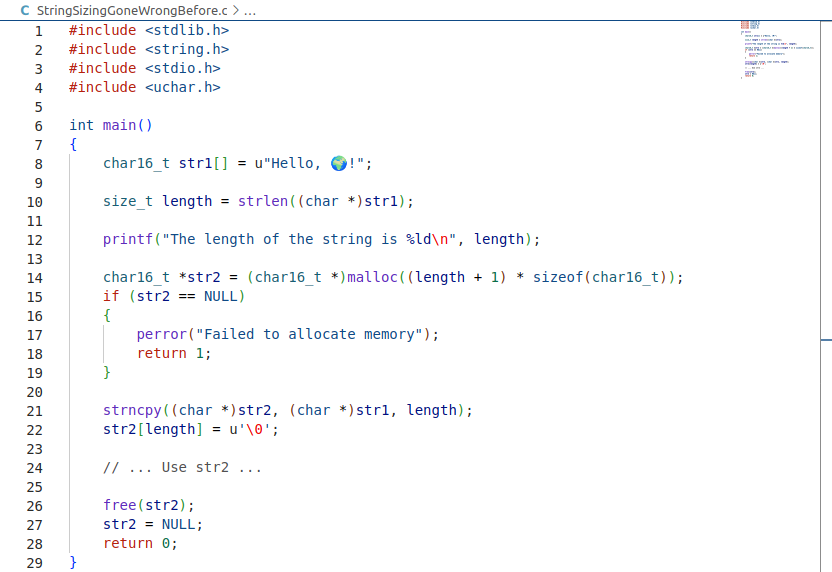
Dimensionarea corectă a șirurilor este esențială pentru prevenirea depăsirii buffer-ului și a altor erori de execuție. Atunci când datele sunt copiate într-un buffer de dimensiuni necorespunzătoare, zona de memorie adiacentă poate fi suprascrisă. Prin urmare, trebuie garantat ca string-ul să aibă destul spațiu pentru caractere și terminatorul nul definit de limbaje ca 0x00.

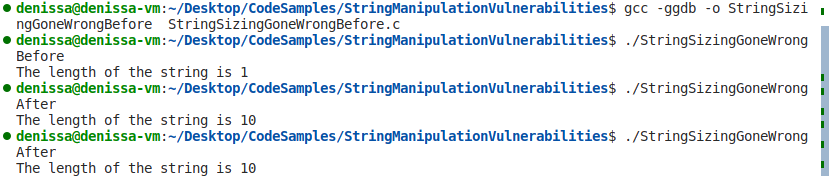
Câteva proprietăți ale șirurilor de caractere și distincțiile dintre acestea sunt esențiale pentru alocarea corectă a spațiului și prevenirea depășirilor buffer-ului: size (numărul de octeți alocați), count (numărul de elemente), length (numărul de caractere înainte de terminatorul nul).

Standardul C garantează că obiectele de tip char constau dintr-un singur octet. În consecință, proprietatea size a unui șir de tip char este egală cu proprietatea count, iar proprietatea length este mai mică sau egală cu size - 1. Pe majoritatea platformelor, dimensiunea wchar\_t este de cel puțin 2 octeți și, în consecință, proprietatea size a unui șir de tip wchar\_t nu mai este egală cu proprietatea count. Programele care presupun altfel sunt susceptibile de a conține erori.

Caracterele de tip wchar\_t, char16\_t, char32\_t din C și C++ sunt utilizate pentru a reprezenta caractere din setul extins de caractere, care include caractere dincolo de setul standard de caractere ASCII. Acestea sunt de obicei utilizate pentru internaționalizare sau când se lucrează cu seturi de caractere care necesită mai mult de un octet pentru reprezentare. În C și C++, string-urile literale asociate sunt L"string", u"string", U"string".

În următorul fragment de program, funcția strlen() este utilizată incorect pentru a determina dimensiunea unui șir de caractere de tip char16\_t.

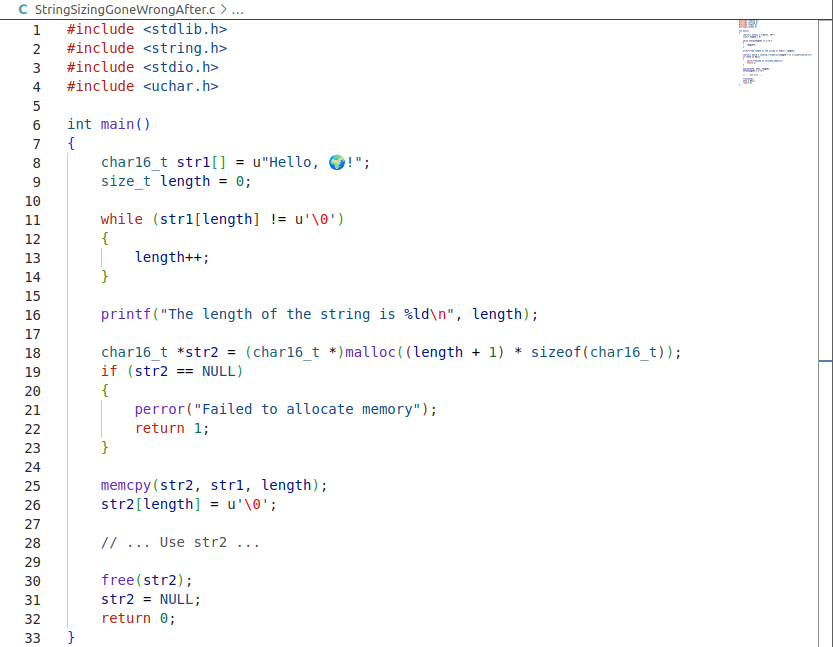


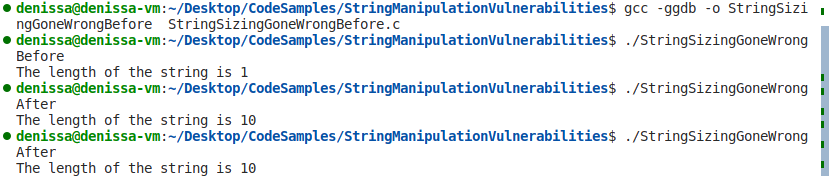


Tipul de date char16\_t este utilizat pentru codificarea UTF-16, unde fiecare caracter este reprezentat prin doi octeți. Funcțiile standard de manipulare a șirurilor din C, cum ar fi strlen() presupun că fiecare caracter este un octet, iar execuția corectă a lor depinde de prezența terminatorului nul. Dacă este utilizată cu un șir de caractere de tip char16\_t, strlen() ar interpreta greșit prima apariție a unui octet cu valoarea 0 ca sfârșit al șirului. Acest lucru se întâmplă într-un caracter UTF-16 valid și duce la copierea incorectă a șirului.

Șirul "Hello, 🌍 !" începe cu caracterul ASCII "H", care în codificarea UTF-16 (char16\_t) este reprezentat ca doi octeți: 0x48 0x00 (deoarece "H" este 0x48 în ASCII și UTF-16 adaugă un octet nul suplimentar pentru caracterele ASCII). Când pentru șirul de caractere se face cast la char \*, strlen() parcurge 0x48 (H) urmat de 0x00 (terminator nul pentru un șir de un octet) și concluzionează incorect că șirul are o lungime de un octet.

Din păcate, biblioteca C standard nu oferă un echivalent direct al funcției strlen() pentru șiruri de tip char16\_t. În lucrul cu șiruri UTF-16 se va utiliza adesea fie o funcție furnizată de o bibliotecă de manipulare Unicode fie se va scrie propria funcție pentru a calcula lungimea.





Lungimea calculată pentru șirul u"Hello, 🌍 !" este 10 din cauza modului în care codificarea UTF-16 reprezintă anumite caractere precum emoji-ul "🌍". UTF-16 utilizează o pereche de 16 biți, cunoscută sub numele de pereche surogat, pentru a reprezenta caracterele care nu se află în planul multilingv de bază (BMP). BMP include caractere care pot fi reprezentate pe 16 biți. Deci, totalul este 7 ("Hello, ") + 2 (emoji) + 1 ("!") = 10.

În C++, este recomandată folosirea std::u16string pentru a gestiona șirurile UTF-16.

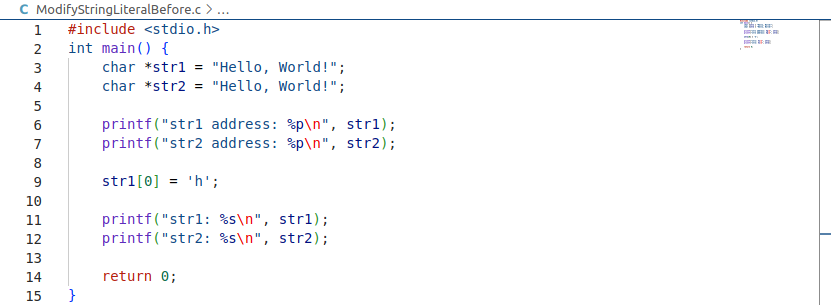
## VULNERABILITĂȚI ALE ȘIRURILOR DE CARACTERE LITERALE

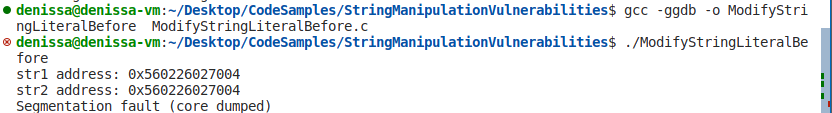
### MODIFICAREA UNUI ȘIR LITERAL DIN MEMORIA READ-ONLY

Un șir de caractere literal este un șir de caractere de tip char în C, respectiv const char în C ++. În consecință, un șir literal este modificabil în C. Încercarea de a modifica direct un șir literal poate duce la comportamente nedefinite sau erori de tip segmentation fault din cauză că regiunea de memorie în care sunt stocate nu poate fi modificată în momentul rulării.

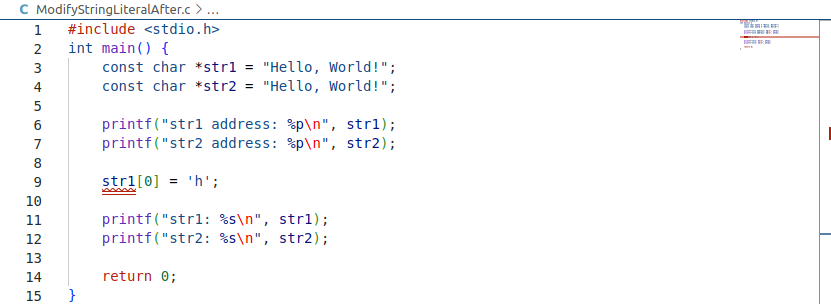
Compilatoarele C și C++ stochează adesea șirurile literale identice la aceeași adresă de memorie. Aceasta este o optimizare cunoscută sub numele de "string pooling" sau "string interning". Compilatorul poate economisi memorie făcându-le pe toate să indice aceeași locație de memorie, reutilizând efectiv aceleași date. Șirurile literale sunt frecvent stocate într-o zonă de memorie doar pentru citire. Dacă se încearcă modificarea unui șir literal, modificarea acestuia le poate schimba și pe celelalte, ceea ce poate duce la un comportament nedefinit.

În acest exemplu, str1 și str2 indică aceeași adresă de memorie, deoarece conțin același șir literal. Încercarea de a modifica str1 duce la un comportament nedefinit și, în practică, duce la o eroare de tip segmentation fault.





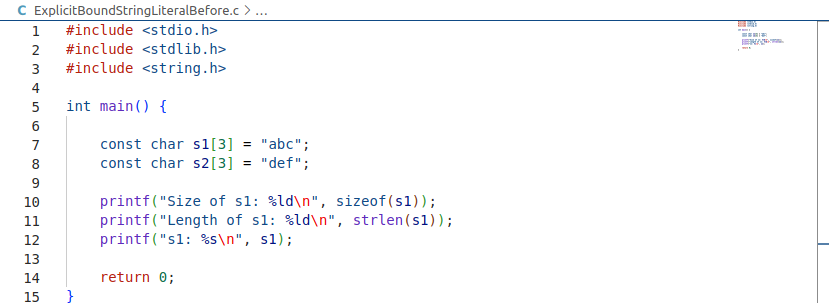
Pentru a evita acest tip de probleme, este recomandată utilizarea șirurilor de caractere pentru șirurile create cu intenția de a le modifica, respectiv a pointer-ilor pentru șiruri care vor fi constante. De exemplu, șirul char str[] = "Hello, world!" prezintă intenția de a fi modificat, în timp ce șirul const char \*const\_str = "Hello, world!" prezintă intenția de a nu îl modifica. Pentru șirul char str[] compilatorul determină proprietatea size pe baza lungimii șirului literal plus terminatorul nul și alocă destulă memorie pe stivă pentru toate cele 12+1 caractere, în timp ce pointer-ul const char \*const\_str va indica o zonă de memorie read-only unde se află șirul literal.

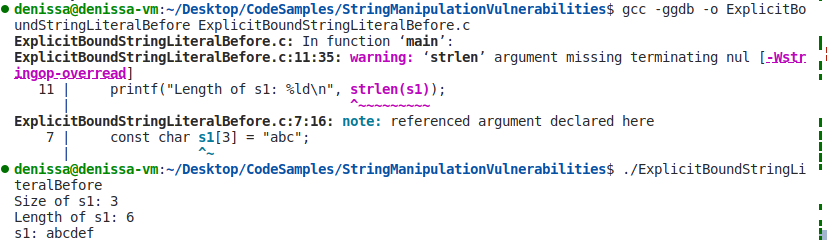


### DIMENSIONAREA EXPLICITĂ A UNUI ȘIR LITERAL

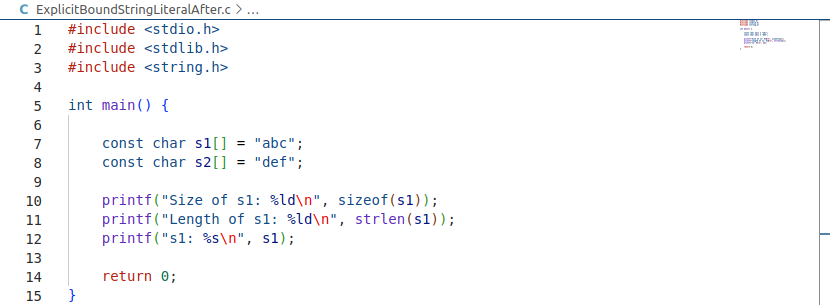
Șirurile de caractere sunt adesea inițializate printr-un șir literal și declarate cu o dimensiune explicită care se potrivește cu numărul de caractere din șirul literal.

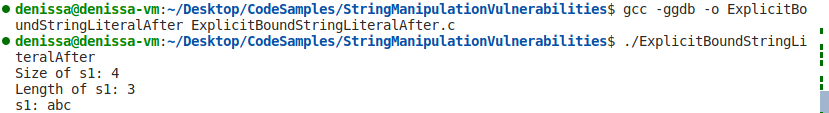
O greșeală frecventă este cea din exemplu, unde dimensiunea explicită nu permite terminarea nulă corectă a șirului. Dimensiunea șirului s este 3, deși dimensiunea șirului literal este 4. În consecință, octetul nul final este omis. Orice utilizare ulterioară a șirului ca șir de octeți cu terminație nulă poate duce la o vulnerabilitate, deoarece s nu este terminat corect.





După cum se poate observa, cum s nu este terminat corect, sunt afișate niște caractere suplimentare care se află după șirul intenționat. Proprietatea size are valoarea 3, iar proprietatea length este 6, ceea ce nu este comportamentul așteptat pentru un șir cu terminație nulă. Acesta este motivul pentru care omiterea dimensionării explicite atunci când se utilizează șiruri literale este recomandată. Compilatorul va aloca automat suficient spațiu pentru întregul șir literal, inclusiv pentru terminatorul nul. Această abordare simplifică mentenanța deoarece dimensiunea șirului de caractere poate fi întotdeauna dedusă, chiar dacă dimensiunea șirului literal se modifică.



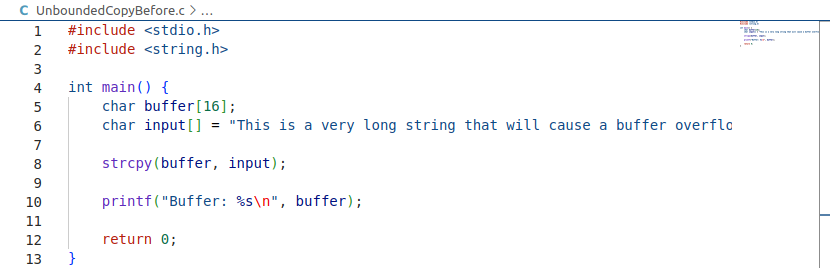


## VULNERABILITĂȚI DE COPIERE NEMĂRGINITĂ A ȘIRURILOR

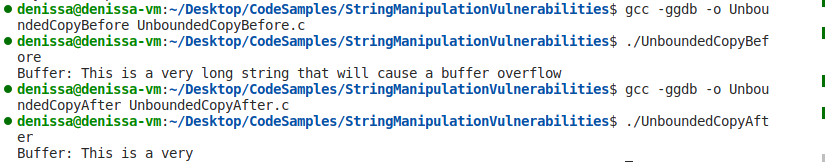
### STRCPY()

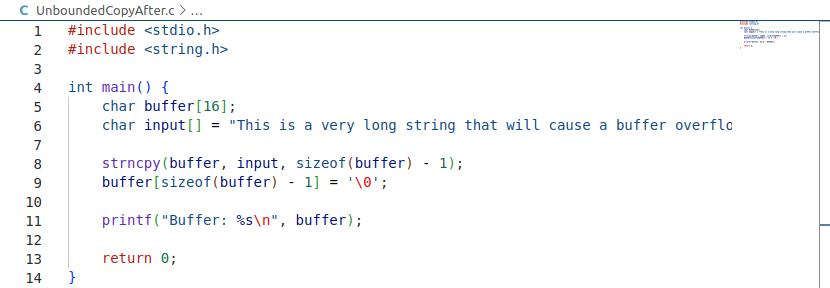
Este ușor a face erori atunci când se copiază sau concatenează șiruri de caractere, deoarece multe dintre apelurile din biblioteca standard cum ar fi strcpy(), strcat() și sprintf(), efectuează operații de copiere nemărginită.

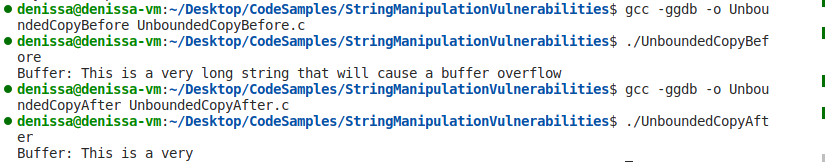
Programul definește un buffer de dimensiune fixă cu o capacitate de 16 caractere. Utilizează funcția strcpy() pentru a copia conținutul șirului de intrare în buffer. Din cauză că intrarea are o lungime mai mare decât dimensiunea buffer-ului, zona de memorie alocată acestuia va fi depășită și datele din zona adiacentă vor fi suprascrise. Acest program este vulnerabil din cauză că utilizează funcția strcpy() fără a verifica dimensiunea buffer-ului destinație, ceea ce duce la un comportament nedefinit prin coruperea memoriei.



Pentru a face acest program mai sigur, ar trebui utilizate funcții de copiere a șirurilor mai sigure, cum ar fi strncpy() sau garantat faptul că șirul sursă nu este mai lung decât buffer-ul destinație înainte de a utiliza strcpy.

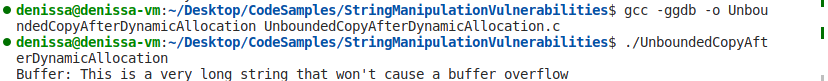


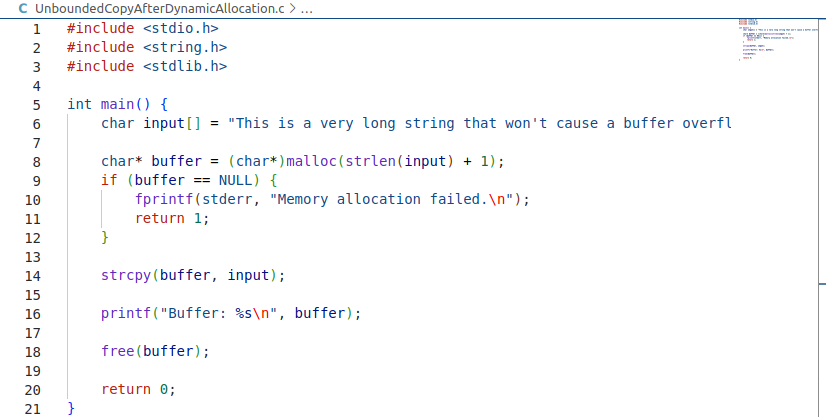




Funcția strncpy() se asigură că operațiunea de copiere nu depășește dimensiunea buffer-ului destinație. De asemenea, este adăugat manual terminatorul nul la sfârșitul buffer-ului pentru a fi asigurat faptul că șirul este terminat corect. Acesta nu este adăugat de funcția strncpy() dacă șirul sursă este mai lung decât buffer-ul destinație.

Poate fi utilizată alocarea dinamică a memoriei pentru a gestiona șiruri de lungimi arbitrare pentru a nu avea griji cu privire la buffer-ele de dimensiuni fixe și potențialele depășiri ale memoriei.

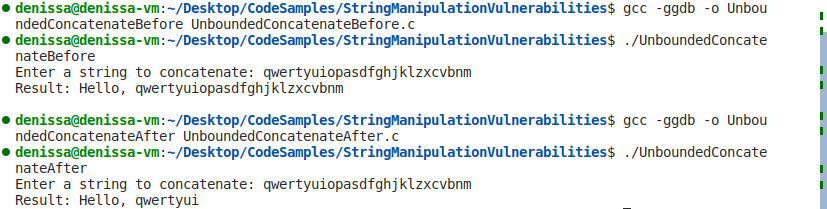




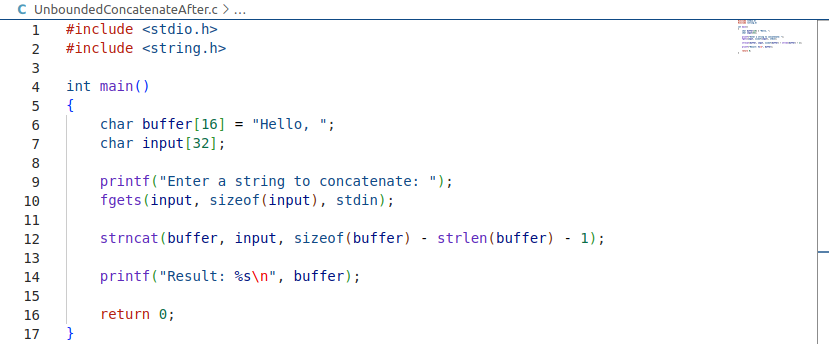
### STRCAT()

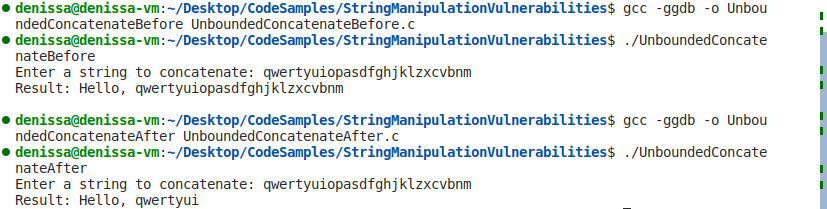
Acesta este un exemplu de program C vulnerabil care utilizează funcția strcat() fără o verificare a limitelor buffer-ului destinație. Acest program concatenează un șir furnizat de utilizator la un buffer de dimensiune fixă, ceea ce poate duce la o depășire a memoriei buffer-ului dacă șirul de intrare este prea lung.





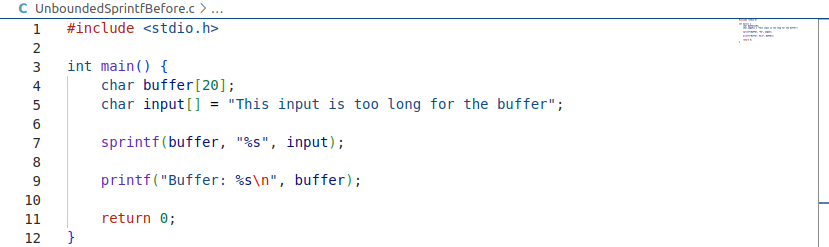
Pentru a face acest program mai sigur, ar trebui utilizată o funcție mai sigură de concatenare a șirurilor, cum ar fi strncat() și asigurat faptul că buffer-ul destinație are suficient spațiu pentru a stoca atât șirul original cât și șirul de intrare:

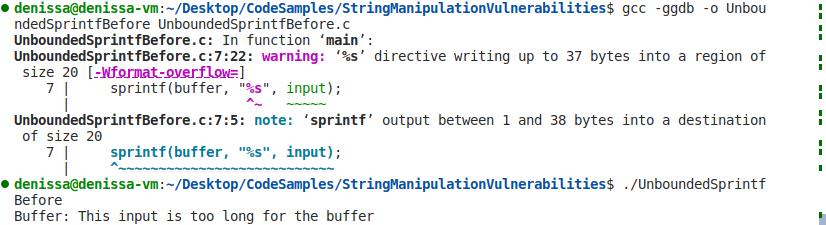




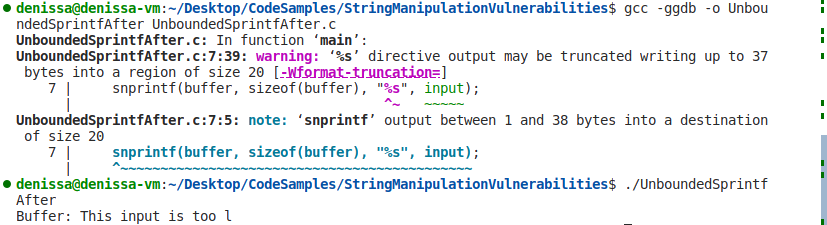
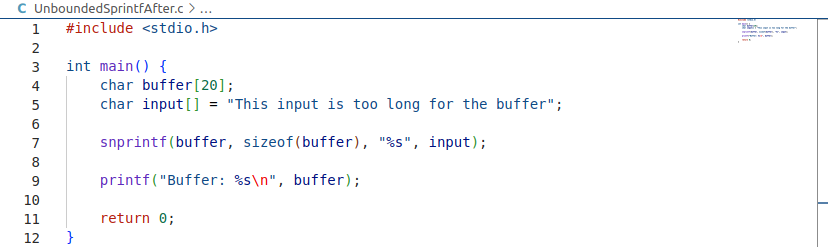
### SPRINTF()

Acesta este un exemplu de program C care este vulnerabil la o depășire a buffer-ului atunci când se utilizează funcția sprintf() fără verificarea corectă a limitelor buffer-ului destinație.



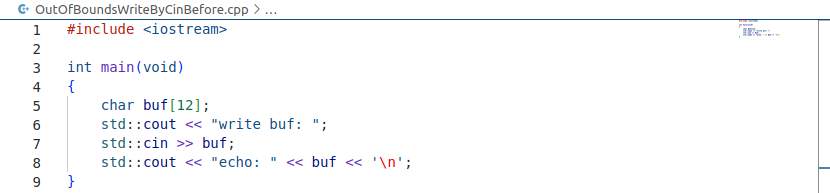


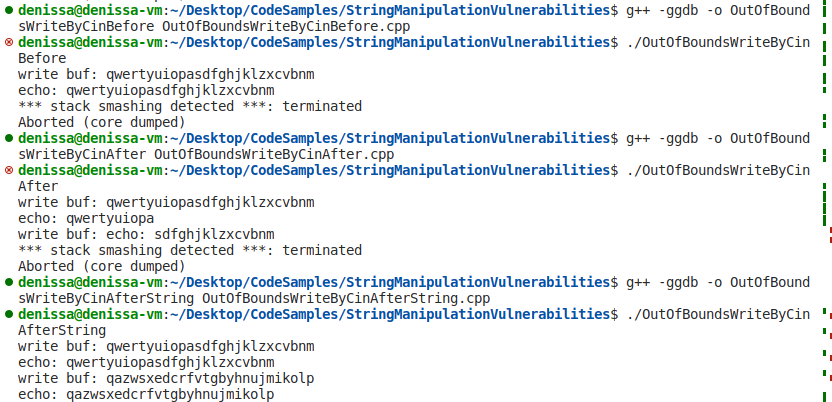
Pentru a face acest program mai sigur, ar trebui utilizată o alternativă mai sigură, cum ar fi snprintf(), care permite specificarea numărului maxim de caractere ce pot fi scrise în buffer-ul destinație, prevenind depășirea acestuia.



### STD::CIN

Copierea nemărginită a șirurilor nu se limitează la limbajul de programare C. De exemplu, dacă un utilizator introduce mai mult de 11 caractere în următorul program C++, va rezulta o scriere în afara limitelor.

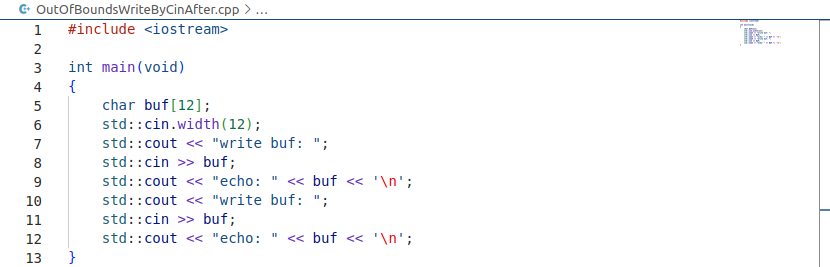


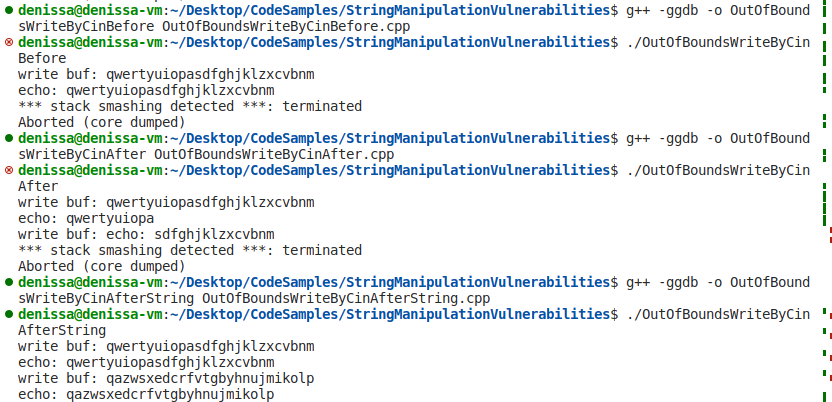


Tipul obiectului standard std::cin este clasa std::stream ce oferă funcții membre pentru citirea și interpretarea intrărilor dintr-un stream buffer, cum ar fi diferite supraîncărcări ale operatorului de extracție>>, inclusiv istream& operator>> (istream& is, char\* str).

Acest operator extrage caractere și le stochează succesiv în șirul str. Extragerea se termină atunci când următorul element este fie un spațiu alb valid fie un caracter nul sau EOF. Operațiunea de extracție poate fi limitată la un anumit număr de caractere (evitând posibilitatea depășirii buffer-ului) dacă membrul width (care poate fi setat prin funcția width() sau setw()) este setat cu o valoare mai mare de 0. În acest caz, extragerea se termină cu un caracter înainte ca numărul de caractere extrase să atingă valoarea membrului width, lăsând spațiu pentru caracterul nul final. După un apel al operației de extracție, valoarea câmpului width este resetată automat la 0.

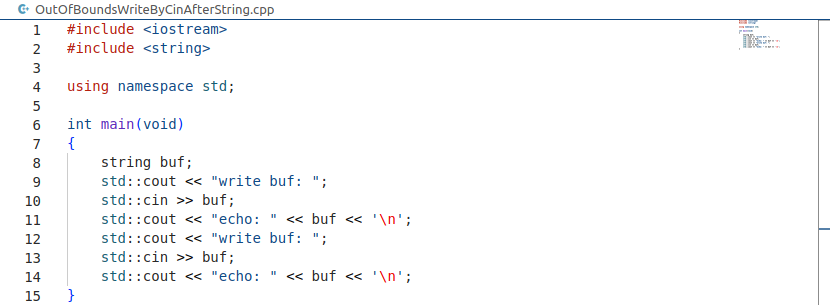
Următorul program elimină depășirea din exemplul anterior prin setarea membrului width cu dimensiunea buffer-ului.

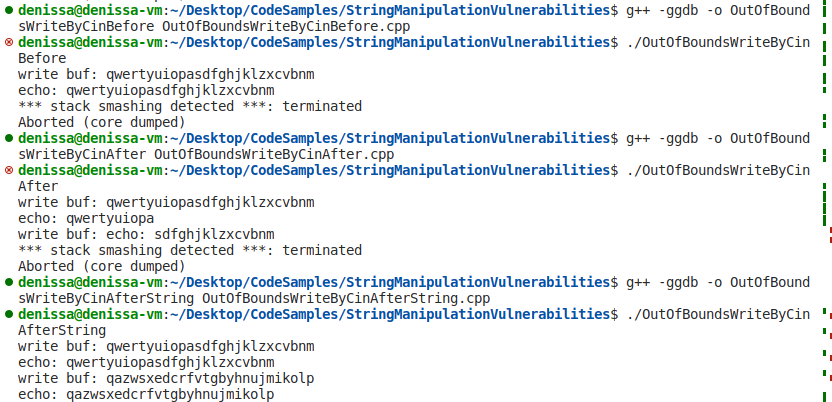




Deși setarea membrului width elimină vulnerabilitatea de depășire a memoriei buffer-ului, aceasta nu abordează problema trunchierii. De asemenea, comportamentul neașteptat al programului rezultă din cauza faptului că valoarea maximă a membrului este atinsă și caracterele rămase în fluxul de intrare sunt consumate de următorul apel către operatorul de extracție. Programatorii C++ au opțiunea de a utiliza clasa std::string.

Următorul program prezintă o soluție pentru extragerea caracterelor din std::cin într-un std::string, folosind un obiect std::string în loc de un șir de caractere:

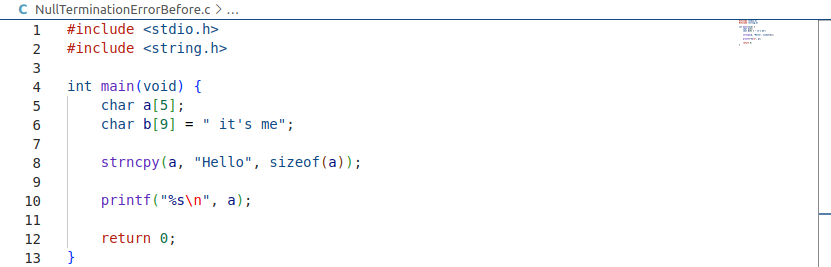


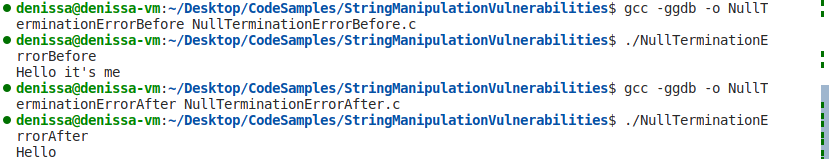


Acest program este simplu și elegant, gestionează depășirile buffer-ului și trunchierea șirurilor și se comportă într-un mod previzibil. Clasa std::string este mai puțin predispusă la vulnerabilități de securitate decât șirurile de octeți cu terminație nulă.

## VULNERABILITĂȚI DE (NE)TERMINARE NULĂ

Majoritatea funcțiilor definite în biblioteca standard de gestionare a șirurilor <string.h>, inclusiv strcpy(), strcat(), strncpy(), strncat() și strtok(), sunt susceptibile la erori. Un șir este terminat corect dacă un terminator nul este prezent ca ultim element din șir. Dacă unui șir îi lipsește caracterul nul terminator, programul poate fi păcălit să citească sau să scrie date în afara limitelor buffer-ului. Șirurile trebuie să conțină terminatorul nul înainte de a putea fi transmise ca argumente pentru funcțiile standard de manipulare a șirurilor, care țin cont de acest lucru în implementarea lor pentru a marca sfârșitul unui șir.



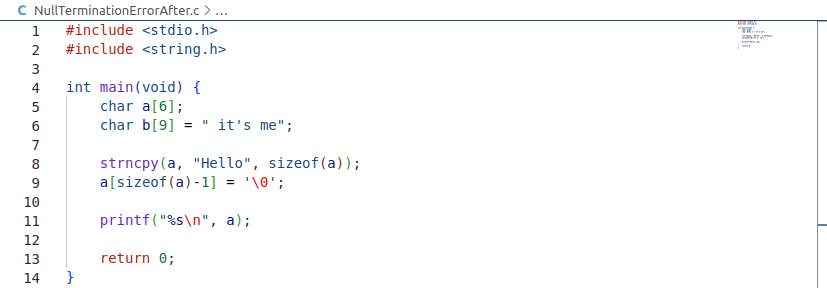


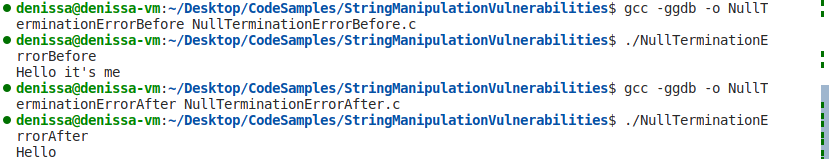
În codul furnizat, există o problemă potențială cu dimensiunea șirului a. Este definit ca char a[5], ceea ce înseamnă că poate conține 4 caractere plus un terminator nul. Cu toate acestea, atunci când este utilizată funcția strncpy() pentru a copia șirul "Hello", acesta conține 5 caractere, ceea ce înseamnă că nu va termina nul șirul.

Astfel, atunci când se încearcă afișarea folosind specificatorul de format %s în printf(), aceasta ar putea duce la un comportament nedefinit, deoarece va continua să afișeze conținutul din memorie până când găsește un terminator nul, așteptându-se la acesta și rezultând într-o citire în afara limitelor.

Conform standardului C, funcția strncpy() copiază nu mai mult de n caractere (caracterele care urmează unui caracter nul nu sunt copiate) din șirul sursă în șirul destinație. În consecință, dacă nu există un caracter nul în primele n caractere ale șirului sursă, ca în acest exemplu, rezultatul nu va fi terminat nul.

Pentru a evita această problemă și pentru a garanta că a este terminat nul, codul este modificat print dimensiunea lui a mărită cu 1 și adăugarea manuală a terminatorului nul.

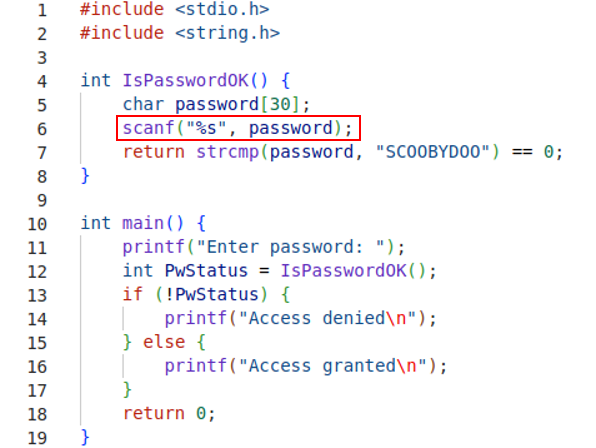




## ATAC (STACK) BUFFER OVERFLOW

Un atac de tip buffer overflow asupra stivei are loc atunci când un program scrie mai multe date pe stivă, depășind dimensiunea limită alocată buffer-ului. Acest lucru duce la suprascrierea unor date importante de pe stivă și provoacă o terminare sau o execuție anormală a programului, cauzată de cele mai multe ori de suprascrierea valorii registrului EIP, care stochează adresa următoarei instrucțiuni ce se va executa după execuția funcției curente, redirecționând astfel execuția fluxului de program.

Acest program verifică dacă parola introdusă de utilizator corespunde unei parole predefinite "SCOOBYDOO". Accesul este refuzat pentru orice parolă care nu coincide cu cea predefinită. În cadrul variabilei declarate password va putea fi stocată o parolă de maximum 29 de caractere plus terminatorul nul \0 care indică sfârșitul șirului de caractere. Funcția scanf() citește șirul de intrare al utilizatorului și îl stochează în variabila password. Utilizarea specificatorului de format %s cu scanf() poate fi nesigură, deoarece nu verifică lungimea intrării și astfel dacă intrarea este mai lungă decât dimensiunea declarată a șirului, poate fi suprascrisă zona de memorie adiacentă.

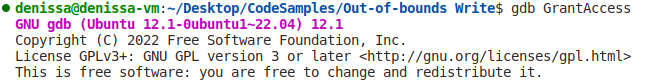


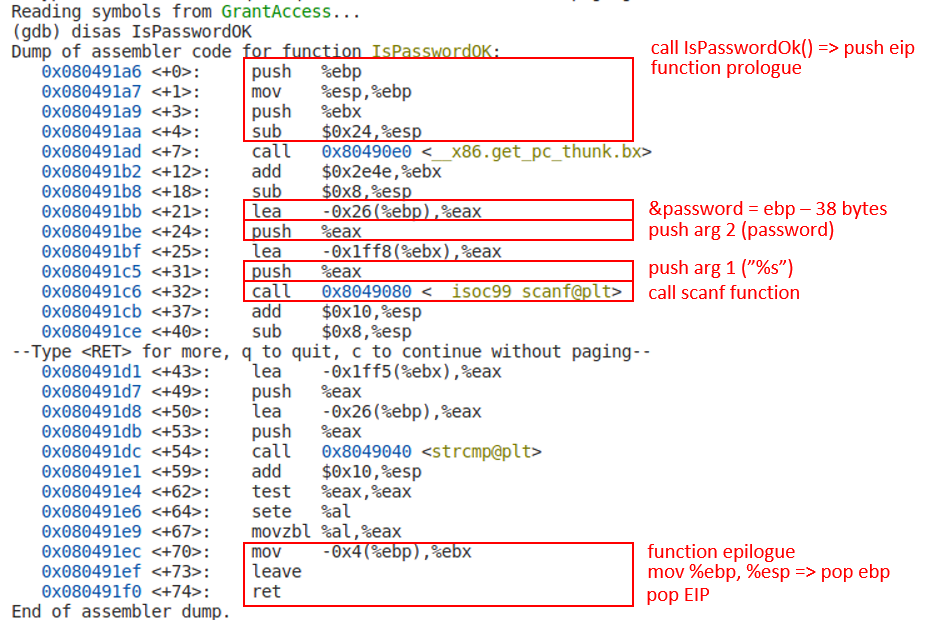


Pentru a începe analiza acestei vulnerabilități programul va fi compilat folosind GNU Compiler Collection (GCC) cu următoarele flag-uri:

* m32: generează cod pentru o arhitectură pe 32 de biți;
* O0: dezactivează aproape toate optimizările (nivelul de optimizare 0), de obicei folosit în timpul dezvoltării pentru a produce cod mai ușor de citit și depanat;
* ggdb: include informații suplimentare în executabil pentru a ajuta la depanarea cu GNU Debugger (GDB);
* fno-stack-protector: dezactivează protecția stivei, precum canarii de stivă care ajută la detectarea atacurilor asupra acesteia (stack smashing attacks);
* zexecstack: marchează stiva ca executabilă, util în dezvoltarea unui shellcode;
* no-pie: dezactivează generarea de executabile independente de poziție, care sunt executabile concepute pentru a fi încărcate la orice adresă de memorie;

După generarea executabilului, acesta va fi depanat, respectiv dezasamblat pentru a putea observa în detaliu formatul memoriei, zona de stivă și zona de cod dând destul de multe indicii despre cum ar putea fi exploatată vulnerabilitatea funcției scanf().



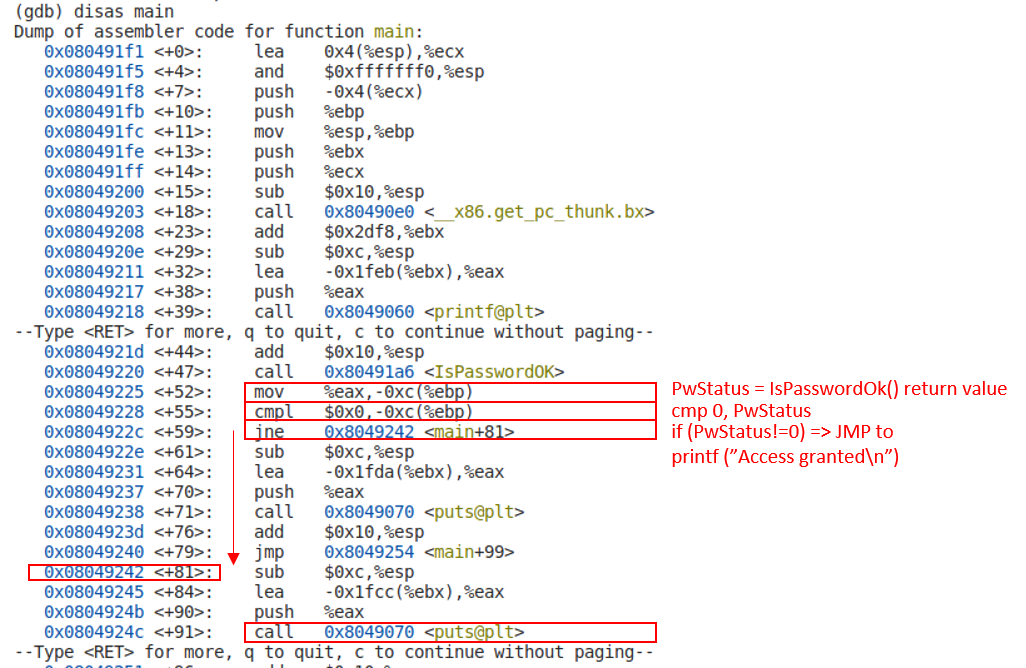


Prologul funcției este reprezentat de 4 instrucțiuni care pregătesc stiva și regiștrii pentru utilizare în cadrul acesteia.

În cazul de față este pusă pe stivă valoarea curentă a registrului EBP, respectiv baza stivei funcției main(), astfel încât să poată fi restaurată ulterior, la ieșirea din funcție. Valoarea registrului ESP este atribuită registrului EBP, astfel încât aceasta să devină baza stivei funcției IsPasswordOk(). Spre deosebire de un prolog generic, în acest caz este pusă pe stivă și valoarea registrului EBX.

Datorită arhitecturii x86 ce implică faptul că stiva este o zonă de memorie ce crește invers, de la o adresă mai mare la o adresă mai mică, compilatorul alocă spațiu pentru variabilele locale scăzând din valoarea registrului ESP 36 de octeți.

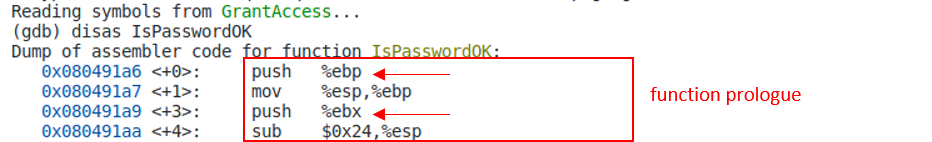
Dacă nu ar fi fost compilat cu flag-ul fno-stack-protector, programul ar fi conținut și canarii de stivă, instrucțiuni suplimentare pentru a preveni atacurile ce constau în scrierea dincolo de dimensiunea unui buffer precum cel ce va fi executat ulterior.



În codul dezasamblat al funcției main() se poate observa că prima instrucțiune din setul de instrucțiuni ce permit accesul se află la adresa 0x08049242.

De asemenea, se poate observa că înainte de apelul funcției scanf() sunt puse pe stivă, în ordine de la dreapta la stânga, argumentele acesteia, adresa șirului de caractere password aflându-se la EBP-38 octeți.

Epilogul funcției este reprezentat de 3 instrucțiuni care au rolul de a restaura contextul anterior execuției acesteia. În cazul de față este restaurată valoarea registrului EBX înainte de intrarea în funcție prin asignarea valorii de la adresa EBP-4, acesta fiind pus pe stivă în prolog.



Instrucțiunea leave constă în reversul primelor două instrucțiuni din prolog, respectiv asignarea valorii registrului EBP, registrului ESP și scoaterea de pe stivă a valorii registrului EBP. După această operațiune, valoarea registrului ESP este constituită de instrucțiunea de unde ar trebui să continue programul, respectiv valoarea EIP care a fost pusă pe stivă în momentul efectuării apelului CALL IsPasswordOk().

Instrucțiunea ret scoate de pe stivă valoarea registrului EIP.



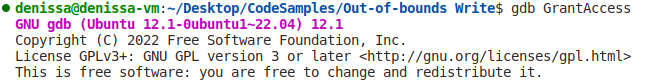
După analiza codului sursă și a celui dezasamblat, propunerea de atac constă în furnizarea unui șir de intrare de 46 de octeți (38 de octeți alocați pe stivă pentru variabila password, 4 octeți pentru EBP și 4 octeți pentru EIP), care să exploateze vulnerabilitatea funcției scanf() și să suprascrie valoarea registrului EIP pusă pe stivă cu valoarea instrucțiunii care oferă accesul, respectiv 0x08049242. Astfel, chiar dacă se efectuează verificarea parolei, indiferent de rezultat, la ieșirea din funcție accesul va fi permis.

Deoarece conform arhitecturii x86, datele sunt stochate în memorie în format little-endian (byte-ul cel mai nesemnificativ la adresa cea mai mică), acești octeți vor fi furnizați în ordine inversă.

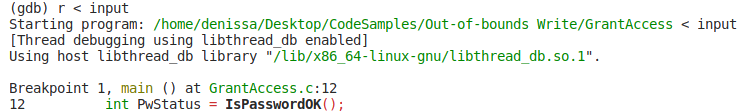


Rezultatul atacului nu este cel așteptat. Deoarece valoarea registrului EBP a fost suprascrisă cu valoarea 0x61616161, la ieșirea din funcția IsPasswordOk() contextul restaurat este corupt (registrului ESP îi este asignată valoarea suprascrisă a registrului EBP), iar instrucțiunea următoare de la adresa 0x08049242 încearcă să aloce pe stivă 12 octeți, scăzându-i de la adresa 0x61616161, zonă de memorie inaccesibilă. Acest fapt este confirmat și de eroarea specifică Segmentation fault.

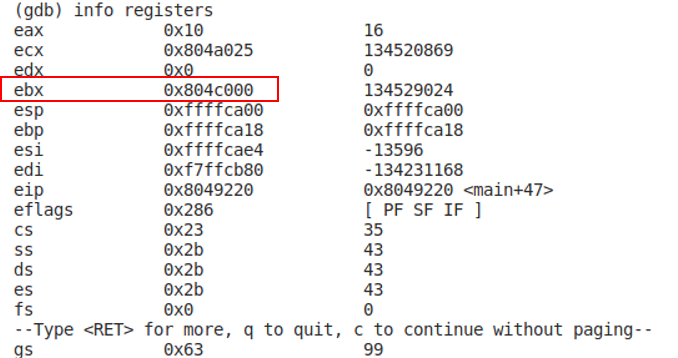
Pentru a determina contextul care trebuie restaurat, programul se va depana cu un breakpoint setat la linia 12 unde are loc apelul funcției IsPasswordOk().



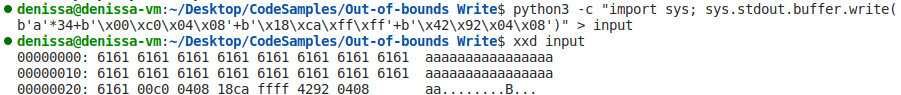






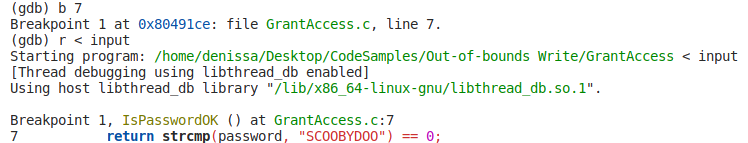


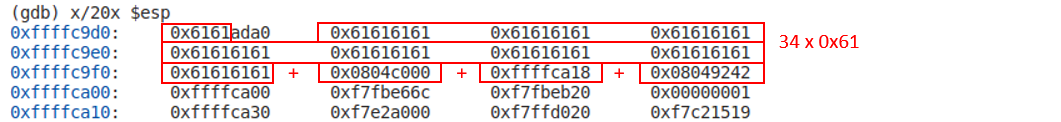
Înainte de apel registrul EBP are valoarea 0xffffca18, iar registrul EBX are valoarea 0x0804c000.



În urma execuției programului cu șirul de caractere generat anterior, accesul a fost permis.







Pentru a remedia vulnerabilitatea, a fost utilizată funcția fgets, precizând dimensiunea șirului de intrare. După cum se poate observa și pe stivă, au fost citiți doar 29 de octeți din șirul de intrare furnizat, iar aceștia sunt urmați de terminatorul nul.

