GCC 3.x Stack Layout

Auswirkungen auf Stack-basierte Exploit-Techniken

Tobias Klein, 2003
tk@trapkit.de

Version 1.0

Abstract

Eine spezielle Eigenschaft des GNU C Compilers (GCC) der Version 3.x wirkt sich mitunter äußerst gravierend auf die gezielte Ausnutzung Stack-basierter Buffer-Overflow-Schwachstellen aus. Innerhalb dieses Papers werden die Ursache sowie die sich aufgrund dieses GCC-Verhaltens ergebenden Auswirkungen einer näheren Betrachtung unterzogen.

1 Spezielles Verhalten des GCC 3.x

Die GCC Versionen 3.x allokieren unter manchen Umständen mehr Speicherplatz auf dem Stack, als eigentlich erforderlich wäre. Dies kann wiederum dazu führen, dass einige der in [1] beschriebenen Beispiele sowie Exploit-Techniken nur noch bedingt nachvollziehbar sind. Im Anschluss soll diese Verhaltensweise der GCC 3.x Compiler-Versionen anhand eines konkreten Beispiels beschrieben werden.

Listing 1
gcc-test.c

```
01 static void
02 funk (void)
03 {
04        char    buff[BUFFERGR];
05 }
06
07 int
08 main (void)
09 {
10        funk();
11
12        return 0;
13 }
```

Assemblercode

Wie sich dem Listing 1: *gcc-test.c* entnehmen lässt, wird innerhalb der Unterfunktion `funk()` ein Puffer mit einer Größe von `BUFFERGR` definiert. Im Anschluss soll der Assemblercode dieses Beispielprogramms erstellt werden. Die Größenangabe des Puffers erfolgt dabei mittels der `-DBUFFERGR`-Option.

```
[user]$ gcc -v[1]
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --
infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-
checking --with-system-zlib --enable-__cxa_atexit --host=i386-redhat-linux
Thread model: posix
gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)

[user]$ gcc -Wall -DBUFFERGR=256 -S -o gcc-test.s gcc-test.c
```

1. Das im folgenden beschriebene Verhalten konnte neben dem Standard-Compiler von Red Hat 9 ebenfalls mittels den Compiler-Versionen GCC 3.2 (Standard-Compiler von SuSE Enterprise 8) und GCC 3.3.1 (Full-Distribution-Tarball-Installation unter Debian 3.0r1) erfolgreich nachvollzogen werden.

Nachdem der Assemblercode des Programms aus Listing 1: *gcc-test.c* mit einer Puffergröße von 256 Byte erstellt wurde, soll dieser im Anschluss näher analysiert werden.

```
[user]$ cat gcc-test.s
        .file   "gcc-test.c"
        .text
        .type   funk,@function
funk:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $264, %esp        ; für den Puffer reservierte Bytes
        leave
        ret
.Lfe1:
        .size   funk,.Lfe1-funk
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        call    funk
        movl    $0, %eax
        leave
        ret
.Lfe2:
        .size   main,.Lfe2-main
        .ident  "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)"
```

Wie sich dem Assemblercode der Unterfunktion `funk()` entnehmen lässt, werden während des Funktionsprologs insgesamt 264 Byte auf dem Stack reserviert. Vergleicht man diesen Wert mit den zuvor als Puffergröße spezifizierten 256 Byte, so lässt sich nachvollziehen, dass insgesamt 8 Byte zuviel auf dem Stack allokiert werden. Im Anschluss soll dieses Verhalten nun unter Zuhilfenahme verschiedener Puffergrößen getestet werden.

> Zur Ermittlung der folgenden Werte wurde bei der Erstellung des Assemblercodes des Programms aus Listing 1: *gcc-test.c* die Puffergröße stets mittels der `-DBUFFERGR`-Option entsprechend angegeben:
>
> ```
> [user]$ gcc -Wall -DBUFFERGR=1 -S -o gcc-test.s gcc-test.c
> [user]$ gcc -Wall -DBUFFERGR=2 -S -o gcc-test.s gcc-test.c
> ```
>
> [...]

Puffergröße	SUB-Instruktion
1	subl $4, %esp
2	subl $4, %esp
3	subl $24, %esp
4	subl $4, %esp
5	subl $24, %esp
6	subl $24, %esp
7	subl $24, %esp
8	subl $8, %esp
9	subl $24, %esp
10	subl $24, %esp
11	subl $24, %esp
12	subl $24, %esp
13	subl $24, %esp
14	subl $24, %esp
15	subl $24, %esp
16	subl $24, %esp
17	subl $40, %esp
18	subl $40, %esp
19	subl $40, %esp
20	subl $40, %esp
24	subl $40, %esp
48	subl $56, %esp
64	subl $72, %esp
128	subl $136, %esp
256	subl $264, %esp

Tab. 1
Ergebnisse

klassische Stack-basierte Buffer Overflows

Wie sich diesen Ergebnissen entnehmen lässt, werden in den meisten Fällen in der Tat mehr Bytes auf dem Stack reserviert, als für den eigentlichen Puffer, bzw. für das 4 Byte Alignment von IA-32-Prozessoren, benötigt werden. Im Anschluss soll nun betrachtet werden, wie sich dieses Verhalten auf die gezielte Ausnutzung Stack-basierter Buffer-Overflow-Schwachstellen auswirkt.

Listing 2
stack_vuln.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04
```

```
05 void
06 funk (char *str)
07 {
08        char buff[256];
09
10        strcpy (buff, str);
11        printf ("[ %s ]\n\n", buff);
12 }
13
14 int
15 main (int argc, char *argv[])
16 {
17        if (argc < 2) {
18                printf ("Kein Argument!\n\n");
19                exit (EXIT_FAILURE);
20        }
21
22        funk (argv[1]);
23
24        return (EXIT_SUCCESS);
25 }
```

Wie sich dem Listing 2: *stack_vuln.c* entnehmen lässt, wird darin in Zeile 10 das erste übergebene Kommandozeilenargument ohne Längenüberprüfung mittels `strcpy(3)` in den Puffer `buff` kopiert. Da der Zielpuffer mit einer statischen Elementgröße von 256 Byte deklariert wurde, kommt es in diesem Fall zu einer klassischen Stack-basierten Buffer-Overflow-Schwachstelle.

Im weiteren soll dieses Beispielprogramm nun erstellt und anschließend während der Programmausführung mittels des Debuggers analysiert werden.

```
[user]$ gcc -Wall -g -o stack_vuln stack_vuln.c
[user]$ gdb -q stack_vuln

(gdb) list 10
5       void
6       funk (char *str)
7       {
8               char buff[256];
9
10              strcpy (buff, str);
11              printf ("[ %s ]\n\n", buff);
12      }
13
14      int
```

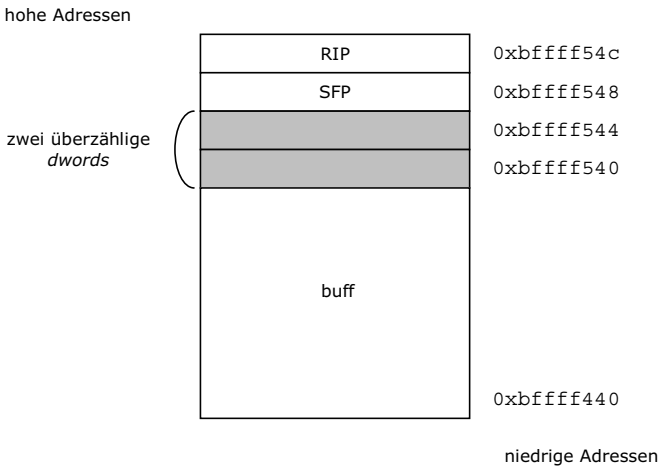
```
(gdb) break 11
Breakpoint 1 at 0x80483ae: file stack_vuln.c, line 11.
```

Nachdem ein Breakpoint in Zeile 11 gesetzt wurde, sollen dem Programm nun bei der Ausführung insgesamt 255 As übergeben werden.

```
(gdb) run `perl -e 'print "A"x255'`
Starting program: /home/user/stack_vuln `perl -e 'print "A"x255'`

Breakpoint 1, funk (str=0xbffffb7b 'A' <repeats 200 times>...) at
stack_vuln.c:11
11              printf ("[ %s ]\n\n", buff);

(gdb) x/68x &buff
0xbffff440:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff450:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff460:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff470:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff480:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff490:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff500:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff510:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff520:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff530:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffff540:     0x42131f18      0x42131a14      0xbffff568      0x08048407

(gdb) i f 0
Stack frame at 0xbffff548:
 eip = 0x80483ae in funk (stack_vuln.c:11); saved eip 0x8048407
 called by frame at 0xbffff568
 source language c.
 Arglist at 0xbffff548, args: str=0xbffffb7b 'A' <repeats 200 times>...
 Locals at 0xbffff548, Previous frame's sp in esp
 Saved registers:
  ebp at 0xbffff548, eip at 0xbffff54c
```

Wie sich den Ausgaben des Debuggers entnehmen lässt, befinden sich zwischen dem Überlaufpuffer, welcher vollständig mittels den übergebenen As gefüllt ist, und dem *Saved Frame Pointer* (SFP) zwei weitere *dwords*.

Abbildung 1
Stack Layout



Diese zwei zusätzlichen *dwords* ergeben sich nun aus dem bereits beschriebenen Verhalten des GCC 3.x, unter Umständen mehr Platz für einen Puffer auf dem Stack zu allokieren, als eigentlich notwendig wäre. Dies wird wiederum durch das im folgenden dargestellte Disassembly der Unterfunktion `funk` verdeutlicht.

```
(gdb) disass funk
Dump of assembler code for function funk:
0x08048390 <funk+0>:    push   %ebp
0x08048391 <funk+1>:    mov    %esp,%ebp
0x08048393 <funk+3>:    sub    $0x108,%esp
0x08048399 <funk+9>:    sub    $0x8,%esp
0x0804839c <funk+12>:   pushl  0x8(%ebp)
0x0804839f <funk+15>:   lea    0xfffffef8(%ebp),%eax
0x080483a5 <funk+21>:   push   %eax
0x080483a6 <funk+22>:   call   0x80482d0 <strcpy>
0x080483ab <funk+27>:   add    $0x10,%esp
0x080483ae <funk+30>:   sub    $0x8,%esp
0x080483b1 <funk+33>:   lea    0xfffffef8(%ebp),%eax
0x080483b7 <funk+39>:   push   %eax
0x080483b8 <funk+40>:   push   $0x80484c0
0x080483bd <funk+45>:   call   0x80482b0 <printf>
0x080483c2 <funk+50>:   add    $0x10,%esp
0x080483c5 <funk+53>:   leave
0x080483c6 <funk+54>:   ret
End of assembler dump.
```

Dem Disassembly lässt sich entnehmen, dass für den Puffer insgesamt 264 (Hex: `0x108`) und somit 8 Byte zuviel auf dem Stack allokiert wur-

den (`sub $0x108,%esp`). Diese beiden überzähligen *dwords* befinden sich nun, wie bereits geschildert, zwischen dem eigentlichen Puffer und dem SFP der Unterfunktion `funk`.

2 Exploiting

Dieses Verhalten hat natürlich Auswirkungen auf die in [1] beschriebenen Beispiele und Exploit-Techniken. Um nun beispielsweise die Rücksprungadresse des Programms aus Listing 2: *stack_vuln.c* gezielt zu überschreiben, reichen 264 Byte (Puffergröße + SFP + RET -> 256 + 4 + 4) nicht mehr aus. Vielmehr ist es notwendig, ebenfalls die 8 überzähligen Bytes zwischen dem SFP und dem Überlaufpuffer zu berücksichtigen. So ist es für eine gezielte Modifikation der Rücksprungadresse in diesem Fall notwendig, 272 Bytes zu übergeben.

```
(gdb) run `perl -e 'print "A"x264 . "B"x4 . "C"x4'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/stack_vuln `perl -e 'print "A"x264 . "B"x4 .
"C"x4'`

Breakpoint 1, funk (str=0xbffffb00 "") at stack_vuln.c:11
11              printf ("[ %s ]\n\n", buff);

(gdb) x/68x &buff
0xbffff620:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff630:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff640:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff650:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff660:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff670:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff680:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff690:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff6f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff700:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff710:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff720:     0x41414141      0x41414141      0x42424242      0x43434343

(gdb) info frame 0
Stack frame at 0xbffff728:
 eip = 0x80483ae in funk (stack_vuln.c:11); saved eip 0x43434343
```

```
source language c.
Arglist at 0xbffff728, args: str=0xbffffb00 ""
Locals at 0xbffff728, Previous frame's sp in esp
Saved registers:
 ebp at 0xbffff728, eip at 0xbffff72c

(gdb) continue
Continuing.
[
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC ]

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

Wie sich den Ausgaben des Debuggers entnehmen lässt, wurde die Rücksprungadresse in der Tat erfolgreich manipuliert.

Off-by-Ones und Frame Pointer Overwrites

Bei Off-by-Ones und Frame Pointer Overwrites wirkt sich das beschriebene Verhalten der GCC 3.x Compiler-Versionen weitaus gravierender aus, als bei der Ausnutzung klassischer Stack-basierter Buffer-Overflow-Schwachstellen mit dem Ziel der gezielten Manipulation der Rücksprungadresse. So ist es bei diesen Buffer-Overflow-Schwachstellen lediglich möglich, das dem Überlaufpuffer unmittelbar nachfolgende Byte erfolgreich zu manipulieren. Werden nun durch den GCC 3.x mehr Bytes auf dem Stack allokiert als eigentlich benötigt, so ist ein Angreifer lediglich dazu in der Lage, ein Byte der überzähligen *dwords* erfolgreich zu manipulieren. Die gezielte Manipulation eines dem Überlaufpuffer nachfolgenden Zeigers oder Saved Frame Pointers ist in diesem Fall jedoch nicht mehr möglich.

Listing 3 *obo.c*

```
01 #include <stdio.h>
02 #include <string.h>
03 #include <stdlib.h>
04
05 #define BUFFERGR        256
06
07 static void
08 funk (char *str)
09 {
10        char    buff[BUFFERGR];
11        int     i;
12
13        for (i = 0; i <= BUFFERGR ; i++)
14                buff[i] = str[i];
15
```

```
16        printf ("[ %s ]", buff);
17 }
18
19 int
20 main (int argc, char *argv[])
21 {
22        if (argc < 2) {
23                printf ("Kein Argument!\n");
24                exit (EXIT_FAILURE);
25        }
26
27        funk (argv[1]);
28
29        return (EXIT_SUCCESS);
30 }
```

Dem Listing 3: *obo.c* lässt sich entnehmen, dass durch die fehlerhafte Verwendung der for-Schleife in Zeile 13 ein klassischer Off-by-One-Overflow verursacht wird. Im Anschluss soll das Beispielprogramm erstellt und mittels des Debuggers analysiert werden.

```
[user]$ gcc -Wall -g -o obo obo.c
[user]$ gdb -q obo

(gdb) list
13              for (i = 0; i <= BUFFERGR ; i++)
14                      buff[i] = str[i];
15
16              printf ("[ %s ]", buff);
17      }
18
19      int
20      main (int argc, char *argv[])
21      {
22              if (argc < 2) {

(gdb) break 15
Breakpoint 1 at 0x80483a2: file obo.c, line 15.
```

Nachdem in Zeile 15 ein Breakpoint gesetzt wurde, soll das Programm jetzt mittels eines Kommandozeilenarguments bestehend aus 257 As gestartet werden.

```
(gdb) run `perl -e 'print "A"x257'`
Starting program: /home/user/obo `perl -e 'print "A"x257'`

Breakpoint 1, funk (str=0xbffffb6b 'A' <repeats 200 times>...) at obo.c:16
16              printf ("[ %s ]", buff);
```

```
(gdb) x/68x &buff
0xbfffeea0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeeb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeec0:     0x41414141      0x41414141      0x41414141      0x41414141

[...]

0xbfffef80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffefa0:     0x42131f41      0x42131a14      0xbfffefc8      0x080483fb

(gdb) info frame 0
Stack frame at 0xbfffefa8:
 eip = 0x80483a2 in funk (obo.c:16); saved eip 0x80483fb
 called by frame at 0xbfffefc8
 source language c.
 Arglist at 0xbfffefa8, args: str=0xbffffb6b 'A' <repeats 200 times>...
 Locals at 0xbfffefa8, Previous frame's sp in esp
 Saved registers:
  ebp at 0xbfffefa8, eip at 0xbfffefac
```

Der Ausgabe zufolge, wird das dem Überlaufpuffer direkt nachfolgende Byte erfolgreich mittels eines As (Hex: `0x41`) überschrieben (siehe `0xbfffefa0`). Jedoch handelt es sich dabei um das Least Significant Byte eines der beiden zusätzlich durch den GCC 3.x Compiler reservierten *dwords*. Wie dieses Beispiel belegt, ist es unter den geschilderten Umständen nicht möglich, die Off-By-One-Schwachstelle innerhalb des Listings 3: *obo.c* erfolgreich auszunutzen, insofern dieses mit einem GCC Compiler der Version 3.x erstellt wurde.

wu-ftpd und rpc.mountd

Aufgrund dieses geschilderten Verhaltens können einige Off-By-One-Schwachstellen innerhalb von Produktionssoftware, darunter z.B. die entsprechende Schwachstelle des *wu-ftpd*-Servers (siehe [5]) oder des *rpc.mountd* Daemons (siehe [6]) nicht auf solchen Plattformen erfolgreich ausgenutzt werden, welche den GCC der Version 3.x als Standard-Compiler einsetzen[1]. Plattformen, die GCC-Versionen kleiner als 3.x als Standard-Compiler verwenden, sind dagegen entsprechend verwundbar.

Ausnahmen

Nichtsdestotrotz lassen sich jedoch einige Arten von Off-By-Ones auch dann erfolgreich ausnutzen, wenn das entsprechend verwundbare Programm mittels GCC 3.x erstellt wurde. So wurde bereits aus den Werten aus Tabelle 1 ersichtlich, dass einige Puffergrößen exakt als solche mittels GCC 3.x auf dem Stack allokiert werden. Dies be-

1. Dazu zählen auch Red Hat 7.x Systeme, deren Standard-Compiler (GCC 2.96) ebenfalls ein entsprechendes Stack Layout produziert.

trifft dabei z.B. sehr kleine Puffer, welche mit 4 oder 8 Byte Größe deklariert werden. Ändert man die Puffergröße in Listing 3: *obo.c* auf 8 Byte, so lässt sich das Programm wiederum erfolgreich mittels eines entsprechenden Exploits ausnutzen.

Listing 3: *obo.c*, vorher:

```
05 #define BUFFERGR        256
```

Listing 3: *obo.c*, nachher:

```
05 #define BUFFERGR          8
```

```
[user]$ gcc -Wall -g -o obo obo.c
[user]$ gdb -q obo

(gdb) list
13              for (i = 0; i <= BUFFERGR ; i++)
14                      buff[i] = str[i];
15
16              printf ("[ %s ]", buff);
17      }
18
19      int
20      main (int argc, char *argv[])
21      {
22              if (argc < 2) {

(gdb) break 15
Breakpoint 1 at 0x804838a: file obo.c, line 15.
```

Nachdem in Zeile 15 ein Breakpoint gesetzt wurde, soll das Programm nun mittels eines Kommandozeilenarguments von 9 Zeichen gestartet werden.

```
(gdb) run AAAAAAAAB
Starting program: /home/user/obo AAAAAAAAB

Breakpoint 1, funk (str=0xbffffc63 "AAAAAAAAB") at obo.c:16
16              printf ("[ %s ]", buff);

(gdb) x/4x &buff
0xbfffdb10:     0x41414141      0x41414141      0xbfffdb42      0x080483e0

(gdb) info frame 0
Stack frame at 0xbfffdb18:
 eip = 0x804838a in funk (obo.c:16); saved eip 0x80483e0
```

```
called by frame at 0xbfffdb42
source language c.
Arglist at 0xbfffdb18, args: str=0xbffffc63 "AAAAAAAAB"
Locals at 0xbfffdb18, Previous frame's sp in esp
Saved registers:
 ebp at 0xbfffdb18, eip at 0xbfffdb1c
```

Wie sich den Ausgaben des Debuggers entnehmen lässt, wurde das *Least Significant Byte* des *Saved Frame Pointers* (SFP) in diesem Fall erfolgreich mittels dem als neuntes Zeichen übergebenen B überschrieben. Das fehlerhafte Programm lässt sich somit ebenfalls dann erfolgreich ausnutzen, wenn es mittels eines GCC der Version 3.x erstellt wurde.

3 Bug oder beabsichtigtes Verhalten

Das beschriebene Verhalten der GCC 3.x Compiler-Versionen wurde bereits in die Fehlerdatenbank des GNU C Compilers mit aufgenommen und dort ansatzweise diskutiert (siehe [2], [3] und [4]). Dazu zwei Zitate seitens der GCC-Enwickler:

"*Note that allocating too much stack space is not wrong, it's just a waste.*"

"*Well, I still think that this is a quality of implementation bug, gcc should be able to do better. I mean, it is not _wrong_ if the compiler allocated too much stack space, it is just wasteful.*"

4 Literatur

[1] T. Klein: *Buffer Overflows und Format-String-Schwachstellen - Funktionsweisen, Exploits und Gegenmaßnahmen*, dpunkt.verlag, 2003.

[2] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=9624

[3] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=10415

[4] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=11232

[5] http://isec.pl/vulnerabilities/isec-0011-wu-ftpd.txt

[6] http://isec.pl/vulnerabilities/isec-0010-linux-nfs-utils.txt