Stack Smashing Protector

Tobias Klein
tk@trapkit.de

Version 1.0, 2003.

Abstract

Es gibt eine Reihe von Compiler-Erweiterungen, welche das Ziel verfolgen, die gezielte Ausnutzung Stack-basierter Buffer-Overflow-Schwachstellen zu erschweren. Der *Stack Smashing Protector* (SSP) a.k.a. *ProPolice* stellt dabei bei näherer Betrachtung eine der vielversprechendsten Implementationen dieser Art dar. In diesem Paper werden die einzelnen verwirklichten Schutzkonzepte von SSP beschrieben und anschließend hinsichtlich eventueller Schwächen untersucht.

1 Überblick

Bei dem *Stack Smashing Protector* (SSP) handelt es sich um eine Erweiterung des *GNU C Compilers*, welche wie *StackGuard*[1], *StackShield*[2] oder die */GS*-Option[3] von Microsoft das Ziel verfolgt, die Ausnutzung Stack-basierter Buffer-Overflow-Schwachstellen zu erschweren.
Um dies zu verwirklichen, wird von SSP, ähnlich wie von StackGuard oder der /GS-Option, u.a. ein Canary-Mechanismus eingesetzt. Wie in [1] bereits beschrieben, besitzt ein solcher Canary-Mechanismus jedoch wiederum einige Schwachstellen, welche unter besonderen Umständen trotz vorhandenem Schutzmechanismus eine gezielte Ausnutzung Stack-basierter Buffer-Overflow-Schwachstellen erlauben. SSP stellt nun zusätzliche Features mit dem Ziel bereit, diese Implementationsschwachstellen konventioneller Canary-Schutzmechanismen weitgehend zu beseitigen.

Bezugsquelle von SSP: http://www.trl.ibm.com/projects/security/ssp/

Im Folgenden sollen diese Besonderheiten von SSP zusammengestellt und kurz beschrieben werden.

Schutz des Saved Frame Pointers

Zum einen wird durch SSP, ähnlich wie bei der */GS*-Option von Visual Studio .NET 2002, neben der Rücksprungadresse ebenfalls der *Saved Frame Pointer* (SFP) in das Schutzkonzept integriert. So werden neben der gezielten Manipulation der Rücksprungadresse ebenfalls gezielte Manipulationen des auf dem Stack gesicherten Frame Pointers erfolgreich unterbunden.

Schutz lokaler (Objekt-)Zeiger und Variablen

Zum anderen ist SSP dazu in der Lage, eine gezielte Umstrukturierung des vorherrschenden Stack Layouts vorzunehmen, so dass Zeiger stets unterhalb (in Richtung niedrigerer Adressen; IA-32[4]) von Puffern auf dem Stack abgelegt werden. Dank dieser Vorgehensweise kann neben der gezielten Manipulation der Rücksprungadresse bzw. des Saved Frame Pointers ebenfalls die Manipulation von lokalen (Objekt-)Zeigern (z.B. Funktionszeigern, Zeigern auf Dateien, etc.) auf dem Stack weitgehend erfolgreich unterbunden werden.

Schutz von Funktionsargumenten

Ein weiterer Mechanismus von SSP verwirklicht ein gezieltes Schutzkonzept hinsichtlich eventueller Funktionsargumente. So werden durch SSP von allen Funktionsargumenten „lokale“ Kopien angelegt, welche sich stets unterhalb (in Richtung niedrigerer Adresswerte; IA-32) des Canaries befinden. Anschließend werden ausschließlich diese „Kopien“ entsprechend referenziert. Die „originalen“ Funktionsargumente werden hingegen nicht weiter berücksichtigt. Beim Anlegen dieser Argumentkopien wird ebenfalls die zuvor erwähnte Umstrukturierung des Stack Layouts berücksichtigt. So werden die Referenzkopien eventueller Puffer beispielsweise unmittelbar angrenzend an das Canary auf dem Stack abgelegt.

Performance-Optimierung

SSP verfügt zudem über die Möglichkeit, die entsprechenden Prüfungen aus Performance-Gründen lediglich in solchen Funktionen bereitzustellen, in welchen es zu entsprechenden sicherheitskritischen Problemen kommen kann.

[1] http://immunix.org/stackguard.html
[2] http://www.angelfire.com/sk/stackshield/
[3] http://msdn.microsoft.com/library/en-us/vccore/html/vclrfGSBufferSecurity.asp
[4] http://www.sandpile.org/docs/intel/ia-32.htm

Random Canary

Neben den genannten Funktionalitäten, ist SSP darüber hinaus dazu in der Lage, als Wert für den Canary durch `/dev/urandom` erzeugte Zufallswerte zu nutzen, was die Wahrscheinlichkeit der erfolgreichen Vorhersage eines Canaries weitgehend ausschließt.

Kompatibilität

Zudem sind die durch SSP verwirklichten Mechanismen weitgehend plattformunabhängig, da im Gegensatz zu StackGuard und StackShield keinerlei architekturspezifische Assembler-Anweisungen in den Programmcode eingefügt werden. Vielmehr werden die einzelnen beschriebenen Schutzmechanismen durch eine gezielte Modifikation des *Intermediate Language Codes* des GGCs verwirklicht: „*We have implemented our system as a intermediate language translator for gcc, which means the implementation is independent of the operating systems and the processors used*" (aus [7]). Trotz dieses plattformübergreifenden Ansatzes ist zumindest die Verwirklichung eines Random Canaries unmittelbar von dem Vorhandensein des `/dev/urandom`-Devices abhängig.

Im Anschluss sollen die einzelnen durch SSP verwirklichten Mechanismen näher beschrieben und hinsichtlich eventueller Grenzen und Schwächen überprüft werden.

> Sämtliche Beschreibungen beziehen sich auf den SSP-Patch *protector-3.3-3* sowie den GNU C Compiler der Version 3.3. Als Testplattform wurde Debian 3.0r1[5] auf IA-32 eingesetzt.

2 SSP – Erstellung

Folgende Schritte wurden zur Erstellung eines SSP-fähigen GNU C Compilers durchgeführt:

```
[user]$ ls
gcc-3.3.tar.gz  protector-3.3-3.tar.gz
```

Entpacken der Sourcen:

```
[user]$ tar zxvf protector-3.3-3.tar.gz
[user]$ tar zxvf gcc-3.3.tar.gz
```

Patchen des GCCs:

```
[user]$ cd gcc-3.3/gcc
[user]$ patch -p1 < ../../protector.dif
[user]$ cp ../../protector.c .
[user]$ cp ../../protector.h .
[user]$ cd ..
```

Erstellung und Installation der gepatchten GCC-Version:

```
[user]$ mkdir obj
[user]$ cd obj
[user]$ ../configure --prefix=/usr/ssp_test1
[user]$ make bootstrap-lean

[user]$ su
Password:
[root]# make install prefix=/usr/ssp_test1
[root]# exit
```

[5] http://www.debian.org

Nachdem man die SSP-fähige GCC-Version erstellt und installiert hat, sollte man darüber hinaus den Pfad entsprechend anpassen:

```
[user]$ cd
[user]$ vi .bash_profile
[...]
PATH=/usr/ssp_test1/bin/:"${PATH}"
```

3 SSP – Canary-Mechanismus und Schutz von Funktionsargumenten

Zunächst soll nun der durch SSP verwirklichte Canary-Mechanismus sowie das Schutzkonzept hinsichtlich eventueller Funktionsargumente einer näheren Betrachtung unterzogen werden.

```
01 #include <stdio.h>
02 #include <string.h>
03
04 void
05 funktion (char *args)
06 {
07       char   buff[512];
08       strcpy (buff, args);
09 }
10
11 int
12 main (int argc, char *argv[])
13 {
14       if (argc > 1)
15       {
16             funktion (argv[1]);
17       } else
18             printf ("Kein Argument!\n");
19
20       return 0;
21 }
```

Listing 1: *stack_bof.c*

Wie sich dem Quellcode aus Listing 1: *stack_bof.c* entnehmen lässt, wird in Zeile 8 eine klassische Stack-basierte Buffer-Overflow-Schwachstelle verursacht, indem von `strcpy(3)` das erste übergebene Kommandozeilenargument ohne Längenüberprüfung in den Puffer `buff` kopiert wird, welcher eine statische Größe von 512 Elementen besitzt. Im Anschluss soll der Assemblercode dieses fehlerhaften Programms aus Listing 1: *stack_bof.c* zunächst ohne jeden SSP-Mechanismus erstellt werden.

```
[user]$ gcc -v
Reading specs from /usr/ssp_test1/lib/gcc-lib/i686-pc-linux-gnu/3.3/specs
Configured with: ../configure --prefix=/usr/ssp_test1
Thread model: posix
gcc version 3.3

[user]$ gcc -S -o stack_bof.s stack_bof.c
[user]$ cat stack_bof.s
        .file   "stack_bof.c"
        .text
.globl funktion
        .type   funktion, @function
funktion:
        pushl   %ebp
        movl    %esp, %ebp
```

```
        subl    $536, %esp
        movl    8(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    -520(%ebp), %eax
        movl    %eax, (%esp)
        call    strcpy
        leave
        ret
        .size   funktion, .-funktion
        .section        .rodata
.LC0:
        .string "Kein Argument!\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        cmpl    $1, 8(%ebp)
        jle     .L3
        movl    12(%ebp), %eax
        addl    $4, %eax
        movl    (%eax), %eax
        movl    %eax, (%esp)
        call    funktion
        jmp     .L4
.L3:
        movl    $.LC0, (%esp)
        call    printf
.L4:
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 3.3"
```

Bei näherer Betrachtung des Assemblercodes der Unterfunktion `funktion()`, lassen sich die folgenden Instruktionen nachvollziehen:

```
funktion:
        pushl   %ebp                ; Der momentane Frame Pointer wird auf dem Stack gesichert.
        movl    %esp, %ebp          ; Die derzeitige Stackspitze wird zur neuen Basis erklärt.
        subl    $536, %esp          ; Reservierung des notwendigen Speicherplatzes.
        movl    8(%ebp), %eax       ; Das erste übergebene Argument wird in EAX gesichert.
        movl    %eax, 4(%esp)       ; Der Wert von EAX wird auf den Stack geschrieben (2.
                                    ; Argument für strcpy(3)).
        leal    -520(%ebp), %eax    ; Die Adresse des Puffers buff wird in EAX gesichert.
        movl    %eax, (%esp)        ; Der Wert von EAX wird auf den Stack geschrieben (1.
                                    ; Argument von strcpy(3)).
        call    strcpy              ; Funktionsaufruf von strcpy(3)
        leave                       ; Funktionsepilog
        ret                         ; s.o.
```

Um eine Abbildung des Stack Layouts der Unterfunktion `funktion()` anfertigen zu können, soll das Programm im Folgenden erstellt und anschließend mittels des Debuggers analysiert werden.

```
[user]$ gcc -g -o stack_bof stack_bof.c

[user]$ gdb -q stack_bof
(gdb) list
5       funktion (char *args)
```

```
6       {
7               char    buff[512];
8               strcpy (buff, args);
9       }
10
11      int
12      main (int argc, char *argv[])
13      {
14              if (argc > 1)

(gdb) break 9
Breakpoint 1 at 0x80483ce: file stack_bof.c, line 9.
```

Nachdem in Zeile 9 ein Breakpoint gesetzt wurde, soll das fehlerhafte Programm im Anschluss mittels eines Kommandozeilenarguments bestehend aus 511 As gestartet werden:

```
(gdb) run `perl -e 'print "A"x511'`
Starting program: /home/user/stack_bof `perl -e 'print "A"x511'`

Breakpoint 1, funktion (args=0xbffffce4 'A' <repeats 200 times>...) at stack_bof.c:9
9       }

(gdb) x/137x $esp
0xbffff970:     0xbffff980      0xbffffce4      0x00000000      0x00000000
0xbffff980:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff990:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa00:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa70:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffaa0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffab0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffac0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffad0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffae0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffaf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb00:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb70:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffffb80:     0x08049560      0x000004a2      0xbffffb9c      0x080483f6
0xbffffb90:     0xbffffce4

(gdb) bt
#0  funktion (args=0xbffffce4 'A' <repeats 200 times>...) at stack_bof.c:9
#1  0x080483f6 in main (argc=2, argv=0xbffffc04) at stack_bof.c:16

(gdb) info frame 0
Stack frame at 0xbffffb88:
 eip = 0x80483ce in funktion (stack_bof.c:9); saved eip 0x80483f6
```

```
called by frame at 0xbffffb9c
source language c.
Arglist at 0xbffffb88, args: args=0xbffffce4 'A' <repeats 200 times>...
Locals at 0xbffffb88, Previous frame's sp is 0x0
Saved registers:
 ebp at 0xbffffb88, eip at 0xbffffb8c
```

Anhand der gesammelten Informationen lässt sich aus dem bestehenden Stack Layout folgende Abbildung erstellen.

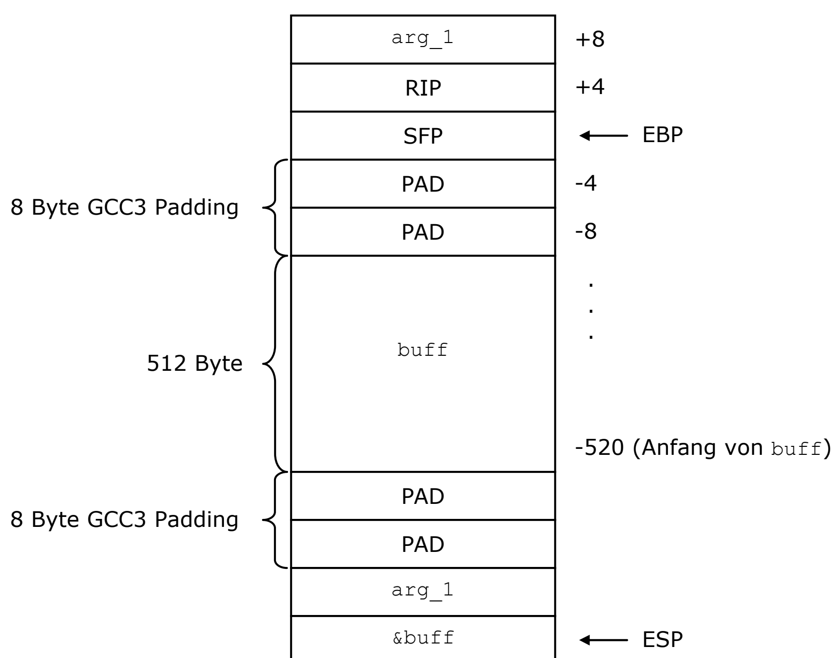

Abb. 1: Stack Layout von `funktion()` ohne SSP

Erstellt man den Assemblercode des Programms aus Listing 1: *stack_bof.c* nun mit Hilfe der durch SSP verwirklichten Mechanismen, so ergibt sich folgende Ausgabe:

> Damit die durch SSP verwirklichten Schutzmechanismen in der Tat bei der Kompilierung berücksichtigt werden, ist es notwendig, die *-fstack-protector*-Option anzugeben.

```
[user]$ gcc -fstack-protector -S -o stack_bof.s stack_bof.c
[user]$ cat stack_bof.s
        .file   "stack_bof.c"
.globl __stack_smash_handler
        .section        .rodata
.LC0:
        .string "funktion"
        .text
.globl funktion
        .type   funktion, @function
funktion:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $568, %esp
        movl    __guard, %eax
        movl    %eax, -24(%ebp)
        movl    8(%ebp), %eax
        movl    %eax, -540(%ebp)
        movl    -540(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    -536(%ebp), %eax
```

```
        movl    %eax, (%esp)
        call    strcpy
        movl    -24(%ebp), %eax
        cmpl    __guard, %eax
        je      .L2
        movl    -24(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    __stack_smash_handler
.L2:
        leave
        ret
        .size   funktion, .-funktion
.globl __stack_smash_handler
        .section        .rodata
.LC1:
        .string "Kein Argument!\n"
.LC2:
        .string "main"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    __guard, %eax
        movl    %eax, -24(%ebp)
        cmpl    $1, 8(%ebp)
        jle     .L4
        movl    12(%ebp), %eax
        addl    $4, %eax
        movl    (%eax), %eax
        movl    %eax, (%esp)
        call    funktion
        jmp     .L5
.L4:
        movl    $.LC1, (%esp)
        call    printf
.L5:
        movl    $0, %eax
        movl    -24(%ebp), %edx
        cmpl    __guard, %edx
        je      .L6
        movl    -24(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC2, (%esp)
        call    __stack_smash_handler
.L6:
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 3.3"
```

Wie zuvor, soll nun ebenfalls der Assemblercode der Unterfunktion `funktion()` näher betrachtet werden:

```
funktion:

[ Normaler Funktionsprolog: ]

        pushl   %ebp                    ; Der momentane Frame Pointer wird auf dem Stack gesichert.
        movl    %esp, %ebp              ; Die derzeitige Stackspitze wird zur neuen Basis erklärt.
        subl    $568, %esp              ; Reservierung des notwendigen Speicherplatzes.
```

```
[ Canary Prolog: ]

        movl    __guard, %eax           ; Der Canary-Wert wird in EAX gesichert.
        movl    %eax, -24(%ebp)         ; Der Wert von EAX (das Canary) wird auf dem Stack
                                        ; gespeichert.

[ Schutzkonzept für Funktionsargumente: ]

        movl    8(%ebp), %eax           ; Das erste übergebene Funktionsargument von funktion() wird
                                        ; in EAX gesichert.
        movl    %eax, -540(%ebp)        ; Der Wert von EAX wird auf den Stack geschrieben
                                        ; ("Argumentkopie").

[ Normaler Aufruf von strcpy(3): ]

        movl    -540(%ebp), %eax        ; Der Wert des zuvor "gesicherten" Funktionsargumentes wird
                                        ; erneut in EAX kopiert.
        movl    %eax, 4(%esp)           ; Der Wert von EAX wird auf den Stack geschrieben (2.
                                        ; Argument für strcpy(3)).
        leal    -536(%ebp), %eax        ; Die Adresse des Puffers buff wird in EAX gesichert.
        movl    %eax, (%esp)            ; Der Wert von EAX wird auf den Stack geschrieben (1.
                                        ; Argument von strcpy(3)).
        call    strcpy                  ; Funktionsaufruf von strcpy(3).

[ Canary Epilog: ]

        movl    -24(%ebp), %eax         ; Der zuvor gesicherte Wert des Canaries wird vom Stack
                                        ; geholt und in EAX kopiert.
        cmpl    __guard, %eax           ; Der vom Stack geholte Wert des Canaries wird mit dem in
                                        ; __guard verglichen.
        je      .L2                     ; Sind die beiden Werte identisch (keine Manipulation) so
                                        ; springe nach .L2.
        movl    -24(%ebp), %eax         ; Sind die beiden Werte nicht identisch, so wird der
                                        ; manipulierte Wert des Canaries in EAX kopiert.
        movl    %eax, 4(%esp)           ; Der Wert von EAX wird auf dem Stack gesichert (der Wert des
                                        ; manipulierten Canaries).
        movl    $.LC0, (%esp)           ; Der Name der Unterfunktion "funktion" wird auf den Stack
                                        ; kopiert.
        call    __stack_smash_handler   ; Der __stack_smash_handler wird aufgerufen.

[ Normaler Funktionsepilog: ]

.L2:
        leave                           ; Funktionsepilog
        ret                             ; s.o.
```

Um auch in diesem Fall eine Abbildung des vorherrschenden Stack Layouts der Unterfunktion `funktion()` erstellen zu können, soll das Programm nun mittels SSP-Unterstützung kompiliert und anschließend mittels des Debuggers analysiert werden.

```
[user]$ gcc -g -fstack-protector -o stack_bof stack_bof.c
[user]$ gdb -q stack_bof
(gdb) l
5       funktion (char *args)
6       {
7               char    buff[512];
8               strcpy (buff, args);
9       }
10
11      int
12      main (int argc, char *argv[])
13      {
14              if (argc > 1)
```

Im Anschluss soll ein Breakpoint am Ende der Unterfunktion `funktion()` gesetzt und das Programm mittels eines 511 Zeichen umfassenden Kommandozeilenarguments gestartet werden:

```
(gdb) break 9
Breakpoint 1 at 0x8048802: file stack_bof.c, line 9.

(gdb) run `perl -e 'print "A"x511'`
Starting program: /home/user/stack_bof `perl -e 'print "A"x511'`

Breakpoint 1, funktion (args=0xbffffce4 'A' <repeats 200 times>...) at stack_bof.c:9
9       }

(gdb) x/145x $esp
0xbffff930:     0xbffff950      0xbffffce4      0x00000000      0x00000000
0xbffff940:     0x00000000      0x00000000      0x00000000      0xbffffce4
0xbffff950:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff960:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff970:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff980:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff990:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa00:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa70:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffaa0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffab0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffac0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffad0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffae0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffaf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb00:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb40:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffffb50:     0x7c685d21      0x400097c0      0x00000020      0x08049e40
0xbffffb60:     0x400da370      0x40013678      0xbffffb9c      0x08048850
0xbffffb70:     0xbffffce4

(gdb) bt
#0  funktion (args=0xbffffce4 'A' <repeats 200 times>...) at stack_bof.c:9
#1  0x08048850 in main (argc=2, argv=0xbffffc04) at stack_bof.c:16

(gdb) info frame 0
Stack frame at 0xbffffb68:
 eip = 0x8048802 in funktion (stack_bof.c:9); saved eip 0x8048850
 called by frame at 0xbffffb9c
 source language c.
 Arglist at 0xbffffb68, args: args=0xbffffce4 'A' <repeats 200 times>...
 Locals at 0xbffffb68, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffb68, eip at 0xbffffb6c

(gdb) x/1x __guard
```

```
0x8049e40 <__guard>:    0x7c685d21
```

Im Folgenden soll aus den gewonnenen Debugger-Informationen eine entsprechende Abbildung des vorherrschenden Stack Layouts der Unterfunktion `funktion()` erstellt werden.

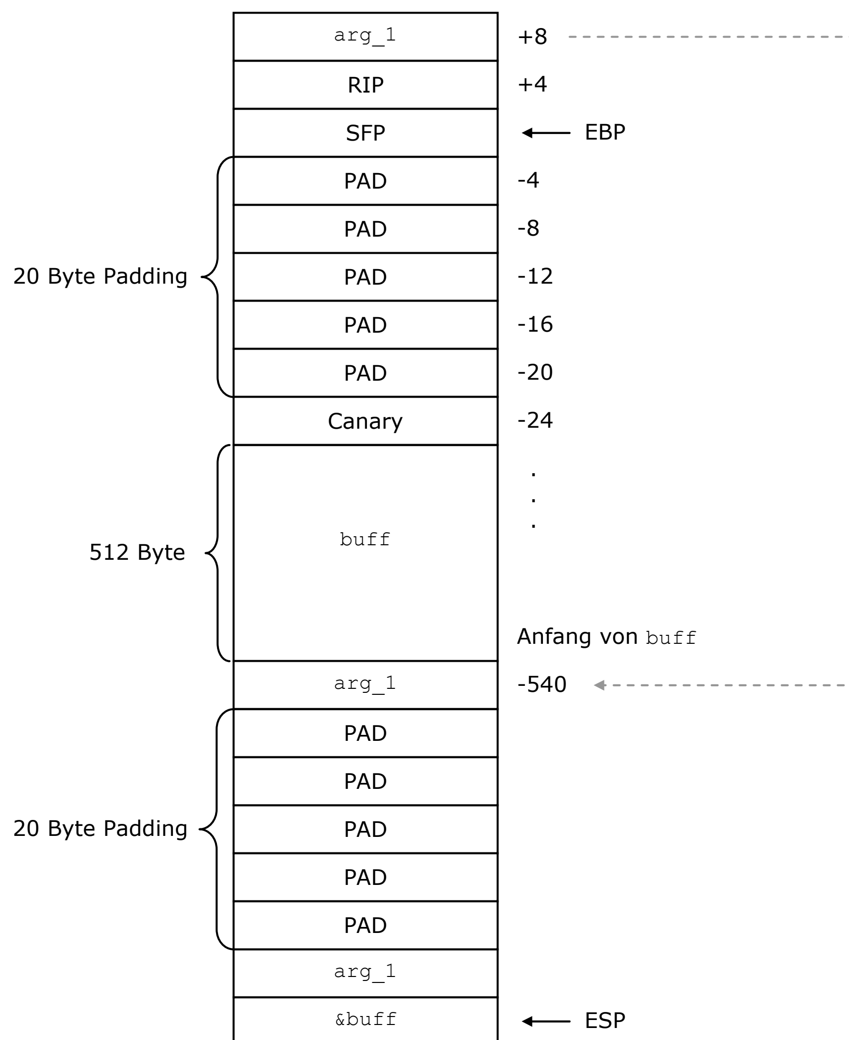

Abb. 2: Stack Layout von `funktion()` mit SSP

Anhand des Assemblercodes, den Debugger-Ausgaben sowie der erstellten Abbildung lässt sich entnehmen, dass durch SSP einige Instruktionen innerhalb der Unterfunktion `funktion()` hinzugefügt wurden, welche bei näherer Betrachtung neben einem klassischen Canary-Mechanismus ebenfalls das bereits erwähnte Schutzkonzept hinsichtlich eventueller Funktionsargumente verwirklichen.

Zum besseren Verständnis sollen zunächst die Assembler-Anweisungen näher betrachtet werden, die das angesprochene Schutzkonzept hinsichtlich vorhandener Funktionsargumente verwirklichen.

```
[ Schutzkonzept für Funktionsargumente: ]

        movl    8(%ebp), %eax          ; Das erste übergebene Funktionsargument von funktion() wird
                                       ; in EAX gesichert.
        movl    %eax, -540(%ebp)       ; Der Wert von EAX wird auf den Stack geschrieben
                                       ; ("Argumentkopie").

[ Normaler Aufruf von strcpy(3): ]
```

```
        movl    -540(%ebp), %eax        ; Der Wert des zuvor "gesicherten" Funktionsargumentes wird
                                        ; erneut in EAX kopiert.
        [...]
```

Wie sich diesen Instruktionen entnehmen lässt, wird in der Tat eine Kopie des übergebenen Funktionsarguments – in diesem Fall des `char`-Zeigers `*args`, welcher wiederum auf `argv[1]` aus `main()` verweist (`movl 8(%ebp), %eax`) – auf dem Stack angelegt. Diese Kopie wird dabei unterhalb (in Richtung niedrigerer Adresswerte; IA-32) des Puffers `buff` platziert (`movl %eax, -540(%ebp)`). Ab sofort wird nun stets diese Argumentkopie entsprechend referenziert. Dies lässt sich beispielsweise bei der anschließenden Referenzierung der `strcpy(3)`-Argumente nachvollziehen, bei der das entsprechende Argument über die zuvor angelegte Kopie referenziert wird (`movl -540(%ebp), %eax`).

Neben diesen Assembler-Anweisungen hinsichtlich der gezielten Erstellung von Argumentkopien, sollen im Anschluss ebenfalls die Instruktionen einer näheren Betrachtung unterzogen werden, welche den Canary-Mechanismus von SSP verwirklichen.

```
funktion:

[ Normaler Funktionsprolog: ]

        pushl   %ebp                    ; Der momentane Frame Pointer wird auf dem Stack gesichert.
        movl    %esp, %ebp              ; Die derzeitige Stackspitze wird zur neuen Basis erklärt.
        subl    $568, %esp              ; Reservierung des notwendigen Speicherplatzes.

[ Canary Prolog: ]

        movl    __guard, %eax           ; Der Canary-Wert wird in EAX gesichert.
        movl    %eax, -24(%ebp)         ; Der Wert von EAX (das Canary) wird auf dem Stack
                                        ; gespeichert.

        [...]

        call    strcpy                  ; Funktionsaufruf von strcpy(3).

[ Canary Epilog: ]

        movl    -24(%ebp), %eax         ; Der zuvor gesicherte Wert des Canaries wird vom Stack
                                        ; geholt und in EAX kopiert.
        cmpl    __guard, %eax           ; Der vom Stack geholte Wert des Canaries wird mit dem in
                                        ; __guard verglichen.
        je      .L2                     ; Sind die beiden Werte identisch (keine Manipulation) so
                                        ; springe nach .L2.
        movl    -24(%ebp), %eax         ; Sind die beiden Werte nicht identisch, so wird der
                                        ; manipulierte Wert des Canaries in EAX kopiert.
        movl    %eax, 4(%esp)           ; Der Wert von EAX wird auf dem Stack gesichert (der Wert des
                                        ; manipulierten Canaries).
        movl    $.LCO, (%esp)           ; Der Name der Unterfunktion "funktion" wird auf den Stack
                                        ; kopiert.
        call    __stack_smash_handler   ; Der __stack_smash_handler wird aufgerufen.

[ Normaler Funktionsepilog: ]

.L2:
        leave                           ; Funktionsepilog
        ret                             ; s.o.
```

Wie sich dem Assemblercode entnehmen lässt, wird das Canary direkt nach dem normalen Funktionsprolog in das EAX-Register kopiert:

```
        movl    __guard, %eax           ; Der Canary-Wert wird in EAX gesichert.
```

Das Symbol `__guard`, welches den Wert des Canaries bereitstellt, wird in der Patch-Datei *protector.dif* entsprechend initialisiert.

```
726 + long __guard[8] = {0,0,0,0,0,0,0,0};
727 + static void __guard_setup (void) __attribute__ ((constructor)) ;
728 + static void __guard_setup (void)
729 + {
730 +   int fd;
731 +   if (__guard[0]!=0) return;
732 +   fd = open ("/dev/urandom", 0);
733 +   if (fd != -1) {
734 +     ssize_t size = read (fd, (char*)&__guard, sizeof(__guard));
735 +     close (fd) ;
736 +     if (size == sizeof(__guard)) return;
737 +   }
738 +   /* If a random generator can't be used, the protector switches the guard
739 +      to the "terminator canary" */
740 +   ((char*)__guard)[0] = 0; ((char*)__guard)[1] = 0;
741 +   ((char*)__guard)[2] = '\n'; ((char*)__guard)[3] = 255;
742 + }
```

Auszug aus *protector.dif*

Wie sich dem Auszug aus der Datei *protector.dif* des SSP Tarballs entnehmen lässt, wird in Zeile 732/733 überprüft, ob das `/dev/urandom`-Device auf dem entsprechenden System verfügbar ist. Ist dies der Fall, so wird `__guard` mit Hilfe von `/dev/urandom` mit einem zufällig bestimmten Wert gefüllt. Ist das `/dev/urandom`-Device jedoch nicht verfügbar, so wird durch SSP, wie bei StackGuard 2.0, standardmäßig ein sogenanntes Terminator-Canary eingesetzt.

> Ein klassisches Terminator-Canary besteht in der Regel aus einer Kombination aus NUL (`0x00`), LF (`0x0a`), EOF (`0xff`) und CR (`0x0d`), um damit die meisten String-Operationen gezielt terminieren zu können. Für eine detailliertere Erläuterung des Terminator-Canaries siehe [1].

Durch die folgende Assembler-Instruktion wird das Canary innerhalb des EAX-Registers nun auf dem Stack hinterlegt:

```
movl    %eax, -24(%ebp)        ; Der Wert von EAX (das Canary) wird auf dem Stack
                               ; gespeichert.
```

Wie sich der *MOV*-Instruktion entnehmen lässt, wird das Canary unterhalb des SFP (und den durch den GCC 3.x überzähligen Padding Bytes (siehe [6])) auf dem Stack abgelegt, so dass dieser erfolgreich in das Schutzkonzept von SSP mit einbezogen wird.

Nachdem alle Instruktionen der Unterfunktion abgearbeitet wurden, wird nun kurz vor dem normalen Funktionsepilog die Integrität des zuvor auf dem Stack abgelegten Canaries überprüft.

```
movl    -24(%ebp), %eax        ; Der zuvor gesicherte Wert des Canaries wird vom Stack
                               ; geholt und in EAX kopiert.
cmpl    __guard, %eax          ; Der vom Stack geholte Wert des Canaries wird mit dem in
                               ; __guard verglichen.
```

Wird keinerlei Manipulation des Canaries festgestellt, so wird der normale Funktionsepilog durchlaufen:

```
je      .L2                    ; Sind die beiden Werte identisch (keine Manipulation) so
                               ; springe nach .L2.

[...]

.L2:
        leave                  ; Funktionsepilog
        ret                    ; s.o.
```

Wird jedoch eine Manipulation des Canaries festgestellt, so wird die `__stack_smash_handler`-Funktion zur Ausführung gebracht.

```
movl    -24(%ebp), %eax        ; Sind die beiden Werte nicht identisch, so wird der
                               ; manipulierte Wert des Canaries in EAX kopiert.
movl    %eax, 4(%esp)          ; Der Wert von EAX wird auf dem Stack gesichert (der Wert des
                               ; manipulierten Canaries).
movl    $.LC0, (%esp)          ; Der Name der Unterfunktion "funktion" wird auf den Stack
                               ; kopiert.
call    __stack_smash_handler  ; Der __stack_smash_handler wird aufgerufen.
```

Wie sich den Assembler-Instruktionen entnehmen lässt, werden der `__stack_smash_handler`-Funktion zwei Argumente übergeben, der Wert des manipulierten Canaries, sowie der Name der Funktion, in welcher der Fehler aufgetreten ist.
Im Anschluss soll die Funktionsweise der `__stack_smash_handler`-Funktion, deren Quellcode sich ebenfalls innerhalb der Datei *protector.dif* befindet, näher betrachtet werden.

```
743 + void __stack_smash_handler (char func[], int damaged ATTRIBUTE_UNUSED)
744 + {
745 + #if defined (__GNU_LIBRARY__)
746 +   extern char * __progname;
747 + #endif
748 +   const char message[] = ": stack smashing attack in function ";
749 +   int bufsz = 256, len;
750 +   char buf[bufsz];
751 + #if defined(HAVE_SYSLOG)
752 +   int LogFile;
753 +   struct sockaddr_un SyslogAddr;  /* AF_UNIX address of local logger */
754 + #endif
755 + #ifdef _POSIX_SOURCE
756 +   {
757 +     sigset_t mask;
758 +     sigfillset(&mask);
759 +     sigdelset(&mask, SIGABRT);        /* Block all signal handlers */
760 +     sigprocmask(SIG_BLOCK, &mask, NULL); /* except SIGABRT */
761 +   }
762 + #endif
763 +
764 +   strcpy(buf, "<2>"); len=3;  /* send LOG_CRIT */
765 + #if defined (__GNU_LIBRARY__)
766 +   strncat(buf, __progname, bufsz-len-1); len = strlen(buf);
767 + #endif
768 +   if (bufsz>len) {strncat(buf, message, bufsz-len-1); len = strlen(buf);}
769 +   if (bufsz>len) {strncat(buf, func, bufsz-len-1); len = strlen(buf);}
770 +
771 +   /* print error message */
772 +   write (STDERR_FILENO, buf+3, len-3);
773 + #if defined(HAVE_SYSLOG)
774 +   if ((LogFile = socket(AF_UNIX, SOCK_DGRAM, 0)) != -1) {
775 +
776 +     /*
777 +      * Send "found" message to the "/dev/log" path
778 +      */
779 +     SyslogAddr.sun_family = AF_UNIX;
780 +     (void)strncpy(SyslogAddr.sun_path, _PATH_LOG,
781 +                   sizeof(SyslogAddr.sun_path) - 1);
782 +     SyslogAddr.sun_path[sizeof(SyslogAddr.sun_path) - 1] = '\0';
783 +     sendto(LogFile, buf, len, 0, (struct sockaddr *)&SyslogAddr,
784 +          sizeof(SyslogAddr));
785 +   }
786 + #endif
787 +
788 + #ifdef _POSIX_SOURCE
789 +   { /* Make sure the default handler is associated with SIGABRT */
790 +     struct sigaction sa;
791 +
792 +     memset(&sa, 0, sizeof(struct sigaction));
793 +     sigfillset(&sa.sa_mask);  /* Block all signals */
794 +     sa.sa_flags = 0;
```

```
795 +     sa.sa_handler = SIG_DFL;
796 +     sigaction(SIGABRT, &sa, NULL);
797 +     (void)kill(getpid(), SIGABRT);
798 +   }
799 + #endif
800 +   _exit(127);
801 + }
802 + #endif
```

Auszug aus *protector.dif*

Wie sich dem Auszug aus der Datei protector.dif entnehmen lässt, wird durch die `__stack_smash_handler`-Funktion neben einer entsprechenden Fehlermeldung auf dem Terminal ebenfalls ein Eintrag in den Systemlogfiles vorgenommen und der Prozess anschließend gezielt beendet.

Im Anschluss soll das beschriebene Verhalten des durch SSP verwirklichten Canary-Mechanismus nun anhand eines konkreten Beispiels überprüft werden. Dazu soll dem fehlerhaften Programm aus Listing 1: *stack_bof.c* bei der Ausführung ein Kommandozeilenargument von 600 Zeichen übergeben werden. Da der Puffer `buff` mit einer statischen Anzahl von lediglich 512 Elementen deklariert wurde, sollte damit der Canary, der SFP sowie die Rücksprungadresse erfolgreich überschrieben werden.

```
[user]$ ./stack_bof `perl -e 'print "A"x600'`
stack_bof: stack smashing attack in function funktionAborted
```

Wie sich der Terminal-Ausgabe entnehmen lässt, wurde das Programm in der Tat erfolgreich durch den SSP-Canary-Mechanismus terminiert. Zudem wurde ebenfalls der folgende Eintrag in den System-Logfiles festgehalten:

```
[user]$ su
Password:

[root]# tail -1 /var/log/syslog
Sep  6 02:45:26 morpheus stack_bof: stack smashing attack in function funktion
```

Fazit – Canary-Mechanismus und Schutz von Funktionsargumenten

Anhand der Beschreibung des Canary-Mechanismus (Random Canary) sowie des Schutzkonzeptes hinsichtlich eventueller Funktionsargumente wird deutlich, dass solche Angriffe gezielt unterbunden werden können, welche das Ziel verfolgen, die Rücksprungadresse, den gesicherten Frame Pointer oder die vorhandenen Funktionsargumente erfolgreich zu manipulieren. Somit bleibt einem Angreifer lediglich noch die Option, lokale Variablen entsprechend zu manipulieren, welche sich zwischen einem Überlaufpuffer und dem Canary auf dem Stack befinden. SSP begegnet diesem Schlupfloch jedoch wiederum mit einer dedizierten Gegenmaßnahme, welche im Folgenden näher untersucht wird.

4 SSP – Schutz lokaler (Objekt-)Zeiger und Variablen

SSP ist dazu in der Lage eine gezielte Umstrukturierung des vorherrschenden Stack Layouts vorzunehmen, so dass Zeiger stets unterhalb (in Richtung niedrigerer Adressen; IA-32) von Puffern auf dem Stack abgelegt werden.
Dank dieser Vorgehensweise kann neben der gezielten Manipulation der Rücksprungadresse, des Saved Frame Pointers sowie eventueller Funktionsargumente ebenfalls die Manipulation von lokalen (Objekt-)Zeigern (z.B. Funktionszeigern, Zeigern auf Dateien, etc.) auf dem Stack weitgehend erfolgreich unterbunden werden.

Um die Funktionsweise dieser gezielten Umstrukturierung des Stack Layouts nachvollziehen zu können, soll das Programm zu folgendem Listing dienen.

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04
05 void
06 funktion (char *s)
07 {
08         printf ("\nParameter: %s\n", s);
09 }
10
11 int
12 main (int argc, char *argv[])
13 {
14         void    (*zgr)(char *s);
15         char    buff[256];
16
17         if (argc <= 2)
18         {
19                 printf ("Gebrauch: %s <Eingabe> <Parameter>\n", argv[0]);
20                 exit (EXIT_FAILURE);
21         }
22
23         zgr = (void (*)(char *s)) funktion;
24         printf ("vorher : zgr -> %p\n", zgr);
25
26         strcpy (buff, argv[1]);
27         printf ("nachher: zgr -> %p\n", zgr);
28
29         (void)(*zgr)(argv[2]);
30         return (EXIT_SUCCESS);
31 }
```

Listing 2: *funktions_zeiger.c*

Wie sich dem Listing 2: *funktions_zeiger.c* entnehmen lässt, wird in Zeile 26 eine klassische Buffer-Overflow-Schwachstelle verursacht, indem der Inhalt des ersten übergebenen Kommandozeilenarguments mittels `strcpy(3)` ohne jede Längenprüfung in den Puffer `buff` kopiert wird, welcher über eine statische Größe von lediglich 256 Byte verfügt. Des Weiteren lässt sich dem Quellcode entnehmen, dass unmittelbar vor dem Überlaufpuffer ein Zeiger deklariert wird (Zeile 14), welcher später als Funktionszeiger genutzt wird (Zeile 23 bzw. 29). Erstellt man das Programm aus Listing 2: *funktions_zeiger.c* ohne SSP-Unterstützung, so lässt sich anschließend nachvollziehen, dass man den Wert des Zeigers beliebig manipulieren kann.

```
[user]$ gcc -o funktions_zeiger funktions_zeiger.c

[user]$ ./funktions_zeiger `perl -e 'print "A"x280'` BB
vorher : zgr -> 0x80483f0
nachher: zgr -> 0x41414141
Segmentation fault
```

Wie sich der Ausgabe des Programms entnehmen lässt, wird der Zeiger erfolgreich mittels den übergebenen As (Hex: `0x41`) überschrieben.

Die Manipulation eines solchen Funktionszeigers wird durch herkömmliche Canary-Mechanismen nicht erfolgreich unterbunden, da sich der Zeiger unterhalb (in Richtung niedrigerer Adresswerte) des Canaries auf dem Stack befindet. SSP begegnet dieser Implementationsschwachstelle durch eine gezielte Umstrukturierung des Stack Layouts, bei der darauf geachtet wird, dass lokale Variablen nie zwischen dem Canary und einem Puffer auf dem Stack abgelegt werden. Um diese Vorgehensweise nachvollziehen zu können, soll im Anschluss der Assembler-Code des Programms aus Listing 2 mit Hilfe von SSP erstellt und anschließend näher untersucht werden.

```
[user]$ gcc -fstack-protector -S -o funktions_zeiger.s funktions_zeiger.c
[user]$ cat funktions_zeiger.s

[...]

main:

[...]

[ Initialisierung des Funktionszeigers:]

        movl    $funktion, -284(%ebp)   ; Initialisierung des Funktionszeigers.

[ Erste Referenzierung des Zeigers: ]

        movl    -284(%ebp), %eax        ; Entsprechende Referenzierung des Funktionszeigers
                                        ; (als Parameter für printf(3)).
        movl    %eax, 4(%esp)           ; Der Wert von EAX wird auf den Stack gelegt (2. Argument für
                                        ; printf(3)).
        movl    $.LC2, (%esp)           ; Der Format String wird auf den Stack geschrieben (1.
                                        ; Argument für printf(3)).
        call    printf                  ; 1. Aufruf von printf(3)

[...]

[ Zweite Referenzierung des Zeigers: ]

        movl    -284(%ebp), %eax        ; Entsprechende Referenzierung des Funktionszeigers (als
                                        ; Parameter für printf(3)).
        movl    %eax, 4(%esp)           ; Der Wert von EAX wird auf den Stack gelegt (2. Argument für
                                        ; printf(3)).
        movl    $.LC3, (%esp)           ; Der Format String wird auf den Stack geschrieben (1.
                                        ; Argument für printf(3)).
        call    printf                  ; 2. Aufruf von printf(3)

[ Aufruf des Funktionszeigers: ]

        movl    12(%ebp), %eax          ; Argument für Funktionszeiger
        addl    $8, %eax                ; → argv[2]
        movl    (%eax), %eax            ;
        movl    %eax, (%esp)            ;
        movl    -284(%ebp), %eax        ; Entsprechende Referenzierung des Funktionszeigers
        call    *%eax                   ; Aufruf
```

Wie sich den ausgewählten Stellen innerhalb des Assembler-Codes entnehmen lässt, wird der Funktionszeiger in der Tat unterhalb des Überlaufpuffers auf dem Stack abgelegt (`movl $funktion, -284(%ebp)`) und anschließend stets über diese Stelle referenziert (`movl -284(%ebp), %eax`). Dieses Verhalten soll ebenfalls innerhalb des Debuggers nachvollzogen werden.

```
[user]$ gcc -fstack-protector -g -o funktions_zeiger funktions_zeiger.c
[user[$ gdb -q funktions_zeiger
(gdb) list 25
20                      exit (EXIT_FAILURE);
21              }
22
23              zgr = (void (*)(char *s)) funktion;
24              printf ("vorher : zgr -> %p\n", zgr);
25
26              strcpy (buff, argv[1]);
27              printf ("nachher: zgr -> %p\n", zgr);
28
29              (void)(*zgr)(argv[2]);

(gdb) break 25
```

```
Breakpoint 1 at 0x804886d: file funktions_zeiger.c, line 25.

(gdb) run A B
Starting program: /home/user/funktions_zeiger A B
vorher : zgr -> 0x80487f0

Breakpoint 1, main (argc=3, argv=0xbffffdf4) at funktions_zeiger.c:26
26              strcpy (buff, argv[1]);

(gdb) print zgr
$1 = (void (*)(char *)) 0x80487f0 <funktion>

(gdb) print &zgr
$2 = (void (**)(char *)) 0xbffffc70

(gdb) print &buff
$3 = (char (*)[256]) 0xbffffc74
```

Wie sich den Debugger-Ausgaben entnehmen lässt, befindet sich der Funktionszeiger in diesem Beispiel an der Adresse `0xbffffc70` auf dem Stack und verweist wie erwartet auf die Unterfunktion `funktion()`. Der Überlaufpuffer `buff` befindet sich dagegen an Adresse `0xbffffc74` auf dem Stack. Aufgrund dieser Adressverteilung, lässt sich folgendes Stack Layout ableiten.

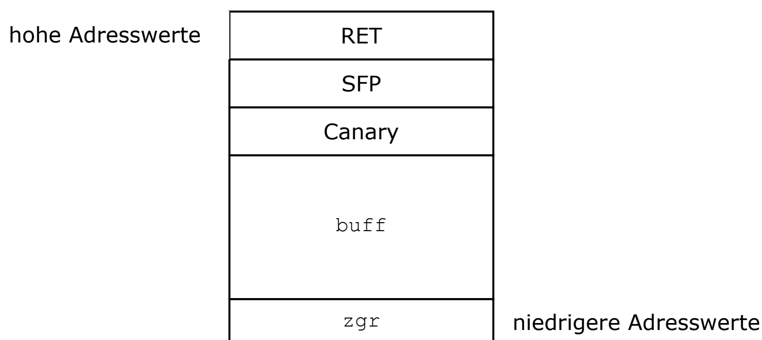

Abb. 3: Stack Layout – Funktionszeiger

Unmittelbar nach dem Puffer `buff` befindet sich also das Canary von SSP. Übergibt man dem durch SSP umstrukturierten Programm nun ein überlanges Kommandozeilenargument, so wird der Programmfluss erfolgreich durch den Canary-Mechanismus terminiert.

```
[user]$ ./funktions_zeiger `perl -e 'print "A"x280'` BB
vorher : zgr -> 0x80487f0
nachher: zgr -> 0x80487f0

Parameter: BB
funktions_zeiger: stack smashing attack in function mainAborted
```

Fazit – Schutz lokaler (Objekt-)Zeiger und Variablen

Aufgrund der zusätzlichen Möglichkeit von SSP, bei Bedarf eine gezielte Anpassung des vorherrschenden Stack Layouts vorzunehmen, lassen sich gezielte Manipulationen lokaler Variablen weitgehend erfolgreich unterbinden.

5 SSP – Schwachstellen und Grenzen

Bei näherer Betrachtung der durch SSP verwirklichten Schutzfunktionen lässt sich nachvollziehen, dass damit ein Großteil Stack-basierter Buffer-Overflow-Exploit-Techniken erfolgreich unterbunden werden können. Nichtsdestotrotz verfügt SSP ebenfalls über einige Grenzen bzw. Schwachstellen innerhalb der verwirklichten Implementation, welche unter bestimmten Umständen wiederum dazu ausgenutzt werden können, um den Schutzmechanismus gezielt zu umgehen. Im Anschluss sollen einige dieser Implementationsschwachstellen näher beschrieben werden.

5.1 Schwachstellen und Grenzen 1: SUSPICIOUS_BUF_SIZE

Der Schutz von SSP beschränkt sich per Default lediglich auf Puffer mit einer statischen Elementgröße von mindestens 8 Byte. Bei Puffern die mit einer Größe <= 8 Byte deklariert werden, wird auf eine Implementation der geschilderten Schutzmechanismen verzichtet. Dazu ein Auszug von der SSP-Webseite [7]:

„*Protect what?*

more than N length array of 'char', 'unsigned char', 'signed char', and its derivatives used as a local variable and a function argument.

What is the number of N?

N is the experimental constant, and it is seven. I examined every program used in RedHat Linux 6.2 how many strings are declared and what is the size. I remember the ratio of the existence of less than 8 characters of string is small (10-20%, sorry I forgot the exact figure). I think, even if N is one, the overhead of protected instruments is not increased so much.“

Im Anschluss soll dieses Verhalten näher untersucht werden.

```
01 #include <stdio.h>
02 #include <string.h>
03
04 void
05 funktion (char *args)
06 {
07         char    buff[7];
08         strcpy (buff, args);
09 }
10
11 int
12 main (int argc, char *argv[])
13 {
14         if (argc > 1)
15         {
16                 funktion (argv[1]);
17         } else
18                 printf ("Kein Argument!\n");
19
20         return 0;
21 }
```

Listing 3: *stack_bof2.c*

Wie sich dem Listing 3 entnehmen lässt, wird in Zeile 8 eine klassische Buffer-Overflow-Schwachstelle verursacht. Der Puffer, welcher als Folge daraus erfolgreich überfüllt werden kann, besitzt eine statische Größe von 7 Byte (Zeile 7). Gemäß dem geschilderten Verhalten von SSP sollten in diesem Fall nun keine entsprechenden Schutzmechanismen für die Unterfunktion `funktion()` verwirklicht werden:

```
[user]$ gcc -fstack-protector -S -o stack_bof2.s stack_bof2.c
```

```
[user]$ cat stack_bof2.s

[...]
funktion:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    -24(%ebp), %eax
        movl    %eax, (%esp)
        call    strcpy
        leave
        ret
[...]
```

Wie sich dem Assembler-Code entnehmen lässt, werden in der Tat keinerlei Schutzmechanismen durch SSP innerhalb dieser Unterfunktion verwirklicht. Damit lässt sich die entsprechende Buffer-Overflow-Schwachstelle erfolgreich ausnutzen, um den Programmfluss gezielt zu manipulieren.

```
[user]$ gcc -fstack-protector -o stack_bof2 stack_bof2.c
[user]$ gdb -q stack_bof2
(gdb) run `perl -e 'print "A"x50'`
Starting program: /home/user/stack_bof2 `perl -e 'print "A"x50'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) info reg eip
eip            0x41414141       0x41414141
```

Dieses geschilderte Verhalten von SSP lässt sich jedoch beheben, indem man innerhalb der Datei *protector.c* die Konstante `SUSPICIOUS_BUF_SIZE` entsprechend anpasst. Per Standardeinstellung wird dieser Konstanten der Wert 8 zugewiesen.

```
112     #define SUSPICIOUS_BUF_SIZE 8

[...]

389           /* Check if the string size is greater than SUSPICIOUS_BUF_SIZE */
390           if (TYPE_MAX_VALUE (TYPE_DOMAIN (type)) != 0
391               && TREE_INT_CST_LOW(TYPE_MAX_VALUE(TYPE_DOMAIN(type)))+1 >= SUSPICIOUS_BUF_SIZE)
392             return TRUE;
```

Auszug aus der Datei *protector.c*

Um beispielsweise sämtliche Puffergrößen in das Schutzkonzept von SSP zu integrieren, sollte man den Wert der Konstanten auf 1 ändern:

```
112     #define SUSPICIOUS_BUF_SIZE 1
```

Damit die Änderung übernommen wird ist es notwendig den GCC neu zu patchen und zu erstellen. Kompiliert man das Programm aus Listing 3: *stack_bof2.c* nun erneut mittels des neu erstellten GCCs, so ergibt sich folgendes Ergebnis hinsichtlich dem Versuch einer gezielten Ausnutzung der beinhalteten Schwachstelle:

```
[user]$ gcc -fstack-protector -g -o stack_bof2 stack_bof2.c
[user]$ gdb -q stack_bof2
(gdb) break funktion
Breakpoint 1 at 0x80487e4: file stack_bof2.c, line 8.

(gdb) run `perl -e 'print "A"x50'`
Starting program: /home/user/stack_bof2 `perl -e 'print "A"x50'`
```

```
Breakpoint 1, funktion (args=0xbffffeaa 'A' <repeats 50 times>) at stack_bof2.c:8
8             strcpy (buff, args);

(gdb) print &buff
$1 = (char (*)[7]) 0xbffffd00

(gdb) cont
Continuing.
stack_bof2: stack smashing attack in function funktion
Program received signal SIGABRT, Aborted.
0x40040781 in kill () from /lib/libc.so.6
```

Wie sich den Debugger-Ausgaben entnehmen lässt, werden nun ebenfalls solche Puffer von dem SSP-Schutzkonzept berücksichtigt, welche eine statische Größe <= 7 Byte besitzen.

5.2 Schwachstellen und Grenzen 2: Gezielte Manipulation von Pufferinhalten

Werden mehrere Puffer statischer Elementgröße innerhalb einer Unterfunktion deklariert, so werden diese durch die gezielte Umstrukturierung des Stacks direkt aufeinander folgend unterhalb (in Richtung niedrigerer Adresswerte) des Canaries platziert.

Der Abb. 4 lässt sich beispielsweise entnehmen, dass es durch die gezielte Überfüllung von `buff2` in diesem Beispiel möglich ist, den Inhalt von `buff1` gezielt zu manipulieren, ohne dass dies erfolgreich durch SSP unterbunden werden kann.

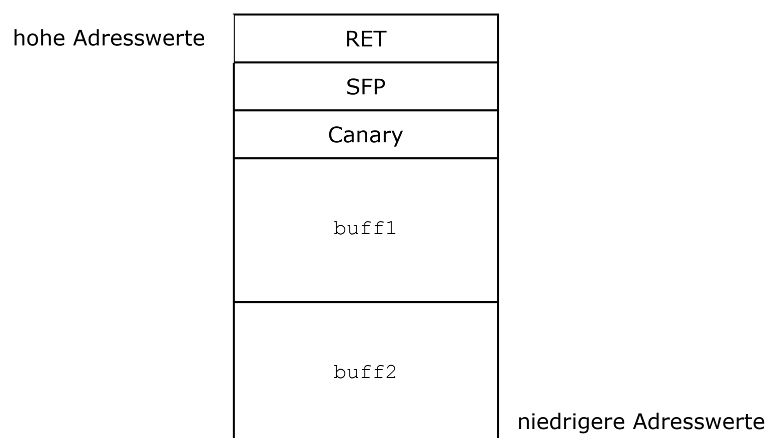

Abb. 4: Manipulation von Pufferinhalten

5.3 Schwachstellen und Grenzen 3: Strukuren

Eine weitere Schwachstelle innerhalb des durch SSP verwirklichten Schutzkonzeptes betrifft die gleichzeitige Verwendung von Zeigern und Puffer statischer Elementanzahl innerhalb einer Struktur. So ist SSP nicht dazu in der Lage, zum Schutz eines solchen Zeigers eine gezielte Umstrukturierung des vorherrschenden Stack-Layouts vorzunehmen. „*If a structure includes both a pointer variable and a character array, the pointer can't be protected, because changing the order of structure elements is prohibited*" (aus [7]). Dieses Verhalten soll im Anschluss mit Hilfe eines konkreten Beispiels veranschaulicht werden.

```
01 #include <stdio.h>
02 #include <stdlib.h>
```

```
03 #include <string.h>
04
05 char    c1[] = "Normale Ausgabe";
06 char    c2[] = "!!!! BINGO !!!!";
07
08 int
09 main (int argc, char *argv[])
10 {
11        struct bla {
12                char    buff[12];
13                char *  ptr;
14                int     dummy;
15        } vuln;
16
17        if (argc < 2) {
18                printf ("Kein Argument!\n");
19                exit (EXIT_FAILURE);
20        }
21
22        vuln.dummy = 0x11111111;
23        vuln.ptr = c1;
24        strcpy (vuln.buff, argv[1]);
25
26        printf ("c1   (%p) : %s\n", c1, c1);
27        printf ("c2   (%p) : %s\n\n", c2, c2);
28        printf ("ptr  (%p): %s (-> %p)\n", &vuln.ptr, vuln.ptr, vuln.ptr);
29        printf ("buff (%p): [ %s ]\n", vuln.buff, vuln.buff);
30
31        return (EXIT_SUCCESS);
32 }
```

Listing 4: *ssp_struct.c*

Wie sich dem Listing 4: *ssp_struct.c* entnehmen lässt, besitzt die Struktur `bla` neben einem Zeiger u.a. einen Puffer mit einer statischen Elementanzahl als Element (siehe Zeile 11-15). In Zeile 23 wird der Zeiger innerhalb der Struktur mit dem String „Normale Ausgabe" initialisiert. In der darauf folgenden Zeile wird dann der Inhalt des ersten Kommandozeilenarguments mittels `strcpy(3)` in den Puffer `buff` kopiert. Da durch `strcpy(3)` keinerlei Prüfung hinsichtlich der Anzahl der zu kopierenden Bytes durchgeführt wird, kommt es zu einer Buffer-Overflow-Schwachstelle. SSP ist es wie bereits erwähnt nun nicht möglich, die Elemente einer Struktur zum Schutz eventuell vorhandener Zeiger gezielt umzustrukturieren. Aufgrund dieser Tatsache lässt sich die Buffer-Overflow-Schwachstelle innerhalb des Listings 4 erfolgreich dazu ausnutzen, um den Zeiger `ptr` gezielt zu manipulieren. Dies soll im Folgenden dazu genutzt werden, um den Zeiger nicht auf den String „Normale Ausgabe" sondern vielmehr auf den String „!!!! BINGO !!!!" verweisen zu lassen.

```
[user]$ gcc -Wall -fstack-protector -o ssp_struct ssp_struct.c
[user]$ ./ssp_struct AAAA
c1   (0x8049d74) : Normale Ausgabe
c2   (0x8049d84) : !!!! BINGO !!!!

ptr  (0xbffffd40): Normale Ausgabe (-> 0x8049d74)
buff (0xbffffd34): [ AAAA ]
```

Wie sich der Ausgabe des Programms aus Listing 4 entnehmen lässt, verweist der Zeiger `ptr` auf den String „Normale Ausgabe", welcher sich an der Adresse `0x8049d74` innerhalb des Prozessspeicherbereichs befindet. Im Anschluss soll der Zeiger nun gezielt mit der Adresse des Strings „!!!! BINGO !!!!" überschrieben werden. Um dies zu erreichen sollen dem verwundbaren Programm zunächst 12 Dummy-As übergeben werden, um dadurch den Puffer `buff` vollständig zu füllen. Anschließend wird dann die Adresse des Strings „!!!! BINGO !!!!" in Little-Endian-Byte-Ordering an das Programm übergeben, um damit den Zeiger gezielt zu überschreiben.

```
[user]$ ./ssp_struct $(perl -e 'print "A"x12 . "\x84\x9d\x04\x08"')
c1   (0x8049d74) : Normale Ausgabe
c2   (0x8049d84) : !!!! BINGO !!!!
```

```
ptr  (0xbffffd30): !!!! BINGO !!!! (-> 0x8049d84)
buff (0xbffffd24): [ AAAAAAAAAAAA
```

Anhand der Ausgabe des verwundbaren Programms lässt sich entnehmen, dass der Zeiger in der Tat trotz des vorhandenen SSP-Schutzkonzeptes erfolgreich manipuliert wurde.

SSP ist somit nicht dazu in der Lage, einzelne Strukturelemente in das verwirklichte Schutzkonzept zu integrieren.

5.4 Schwachstellen und Grenzen 4: Trampoline Code

Neben den bereits beschriebenen Schwachstellen hat SSP darüber hinaus Probleme bei der gezielten Absicherung solcher Funktionen, welche so genannten Trampoline Code aufrufen. „*A trampoline is a small piece of code that is created at run time when the address of a nested function is taken. It normally resides on the stack, in the stack frame of the containing function.*" (aus [2]).

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <string.h>
04
05 void
06 call (void (*tramp_funk)(void))
07 {
08         printf ("Trampoline call\t");
09
10         tramp_funk();
11 }
12
13 void
14 trampoline (char *arg)
15 {
16         char    buff[12];
17
18         memset (buff, 'i', 12);
19         strcpy (buff, arg);
20
21         void
22         nested_funk (void)
23         {
24                 printf ("[ OK ]\n\n");
25         }
26
27         call (nested_funk);
28 }
29
30 int
31 main (int argc, char *argv[])
32 {
33         if (argc < 2) {
34                 printf ("Kein Argument!\n");
35                 exit (EXIT_FAILURE);
36         }
37
38         trampoline (argv[1]);
39
40         return (EXIT_SUCCESS);
41 }
```

Listing 5: *ssp_trampoline.c*

Innerhalb der Unterfunktion `trampoline()` des Listings 5, wird eine verschachtelte (*nested*) Funktion deklariert und anschließend via der `call()`-Funktion zur Ausführung gebracht. Zudem lässt sich dem

Quellcode entnehmen, dass mittels der strcpy(3)-Funktion in Zeile 19 das erste Kommandozeilenargument ohne Längenprüfung in den Puffer buff kopiert wird, welcher über eine statische Anzahl von lediglich 12 Elementen verfügt. Es handelt sich dabei um eine klassische Stack-basierte Buffer-Overflow-Schwachstelle.

```
[user]$ gcc -Wall -fstack-protector -g -o ssp_trampoline ssp_trampoline.c
```

Nachdem das fehlerhafte Programm aus Listing 5 mittels SSP-Unterstützung erstellt wurde, soll der Puffer im Folgenden gezielt überfüllt werden.

```
[user]$ gdb -q ssp_trampoline
(gdb) run $(perl -e 'print "A"x28 . "B"x4')
Starting program: /home/user/ssp_trampoline $(perl -e 'print "A"x28 . "B"x4')
Trampoline call [ OK ]

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Wie sich der Ausgabe des Debuggers entnehmen lässt, ist es trotz vorhandenem SSP-Schutzmechanismus in diesem Fall möglich, die Rücksprungadresse der trampoline()-Funktion erfolgreich zu manipulieren. Dies lässt sich nun wiederum beispielsweise dazu ausnutzen, um zuvor eingeschleusten Programmcode erfolgreich zur Ausführung zu bringen.

```
[user]$ ulimit -c 100000
[user]$ ./ssp_trampoline $(perl -e 'print "A"x28 . "B"x4')
Trampoline call [ OK ]

Segmentation fault (core dumped)
```

Nachdem einen *Coredump* erzeugt hat, soll dieser im Anschluss mittels des Debuggers analysiert werden.

```
[user]$ gdb -q ssp_trampoline core
Core was generated by `./ssp_trampoline AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x42424242 in ?? ()

(gdb) x/10x $esp + 386
0xbffffe92:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffea2:     0x41414141      0x41414141      0x41414141      0x42424242
0xbffffeb2:     0x44575000      0x6f682f3d
```

Wie sich der Debugger-Ausgabe entnehmen lässt, befindet sich der Anfang des Puffers buff in diesem Beispiel an Adresse 0xbffffe92. Nun soll der Puffer mittels einer Reihe von NOPs, sowie einer Endlosschleife in Form der Sprunganweisung 0xeb 0xfe gefüllt werden. Überschreibt man darüber hinaus die Rücksprungadresse mittels der soeben ermittelten Adresse des Überlaufpuffers, so sollte der Code der Endlosschleife erfolgreich zur Ausführung gebracht werden.

```
[user]$ ./ssp_trampoline $(perl -e 'print "\x90"x26 . "\xeb\xfe" . "\x92\xfe\xff\xbf"')
Trampoline call [ OK ]

^C
```

Die Endlosschleife wird in der Tat so lange ausgeführt, bis diese mittels *CTRL+C* unterbrochen wird.

Wie sich diesem Beispiel entnehmen lässt, ist SSP in der Tat nicht dazu in der Lage solche Funktionen erfolgreich in das verwirklichte Schutzkonzept zu integrieren, welche Trampoline Code aufrufen.

5.5 Schwachstellen und Grenzen 5: BSS und Heap Overflows

Aufgrund der geschilderten Schutzmechanismen, welche durch SSP verwirklicht werden, lässt sich feststellen, dass BSS- sowie Heap Overflows in keinster Weise durch SSP adressiert werden. Diese Buffer-Overflow-Generationen lassen sich somit nicht erfolgreich mittels SSP unterbinden.

6 Gesamtfazit

SSP verwirklicht durch die Kombination des klassischen Canary-Mechanismus mit zusätzlichen innovativen Techniken eine leistungsfähige Möglichkeit, um die gezielte Ausnutzung Stack-basierter Buffer-Overflow-Schwachstellen erfolgreich zu erschweren. Nichtsdestotrotz besitzt diese Compiler Erweiterung einige Defizite, welche es unter Umständen erlauben, den verwirklichten Schutzmechanismus gezielt zu umgehen. Darüber hinaus konzentrieren sich die verwirklichten Schutzkonzepte lediglich auf das Stack-Segment, wobei BSS- sowie Heap Overflows in keinster Weise berücksichtigt werden.
Im direkten Vergleich zu anderen Compiler-Erweiterungen – wie beispielsweise StackGuard, StackShield sowie der /*GS*-Option von Visual Studio .NET 2002 – ist SSP dennoch die im Moment leistungsfähigste und ausgereifteste Implementation. Gerade in Kombination mit anderen Schutzmechanismen, wie beispielsweise PaX oder der vergleichbaren *W^X*-Implementation von *OpenBSD* kann SSP einen durchaus effektiven Beitrag zu einem Gesamtschutzkonzept beitragen. So wird eine entsprechende Kombination dieser beiden unterschiedlichen Schutzmechanismen bereits durch verschiedene Linux- sowie BSD-Varianten entsprechend umgesetzt (siehe u.a. [3], [4] und [5]).

7 Literatur

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

[2] GCC Internals Manual, http://gcc.gnu.org/onlinedocs/gccint/

[3] OpenBSD, http://www.openbsd.org/

[4] Adamantix, http://www.adamantix.org/

[5] Hardened Gentoo, http://www.gentoo.org/proj/en/hardened/

[6] T. Klein: *GCC 3.x Stack Layout - Auswirkungen auf Stack-basierte Exploit-Techniken*, http://www.trapkit.de, 2003.

[7] SSP-Webseite, http://www.trl.ibm.com/projects/security/ssp/