Gefahrenreduzierung nach stackbasierten BufferOverflows

Lars Knösel

22. Dezember 2005

Wichtiger Hinweis

Dieses Dokument und die Live-Demonstration liefern Informationen, mit deren Hilfe es möglich wird, stackbasierte Pufferüberläufe dahingehend auszunutzen, unautorisierten Zugriff auf Rechnern zu erlangen.

Sämtliche Informationen, die in diesem Dokument zur Verfügung gestellt werden, dienen ausschließlich der Lehre. Dieser Sachverhalt gilt ebenfalls für die Live-Demonstration.

Der Autor haftet in **keinster** Weise für den Missbrauch dieser Informationen.

Motivation

in-depth summary of what's happened in the world of Debian.

Security Advisories

[21 Dec 2005] DSA-924 nbd - buffer overflow
[19 Dec 2005] DSA-923 dropbear - buffer overflow
[14 Dec 2005] DSA-922 kernel-source-2.6.8 - several vulnerabilities
[14 Dec 2005] DSA-921 kernel-source-2.4.27 - several vulnerabilities
[13 Dec 2005] DSA-920 ethereal - buffer overflow
[12 Dec 2005] DSA-919 curl - buffer overflow
[09 Dec 2005] DSA-918 osh - programming error
[08 Dec 2005] DSA-917 courier - programming error
[07 Dec 2005] DSA-916 inkscape - buffer overflow
[02 Dec 2005] DSA-915 helix-player - buffer overflow

For older security advisories see the Security Page. If you would like to rece
advisories as soon as they're announced, subscribe to the debian-security-a

This page is also available in the following languages:

URL:http://www.debian.org/ (Stand: 21.12.2005)

Überblick

- Software
- Speicher
- Prozessor
- Stack
- Debugger *gdb*
- BufferOverflow
- Exploit
- Demo
- Maßnahmen
- Fazit

Software - Sicherheit

- „Almost all the security problems that happen in Software, like probably 95 percent of them, are low-level programmer errors.” Theo de Raadt
- Ein Angreifer nutzt die Seiteneffekte, die durch einen Softwarefehler entstehen, weil er sie versteht
- Der Angreifer erlangt Privilegien, da sich Systeme immer wieder gleich verhalten
- Es gibt grundsätzlich zwei Möglichkeiten, dies zu unterbinden:
  - Fehler in der Software beseitigen
  - Das System so verändern, dass es sich nicht mehr auf herkömmliche Weise für Angriffe nutzen lässt

Grundlagen Speicher – Segmente

- Text-Segment (ro)
- Data-Segment
  - BSS

  (Block Started by Symbol)
- Heap
- Stack

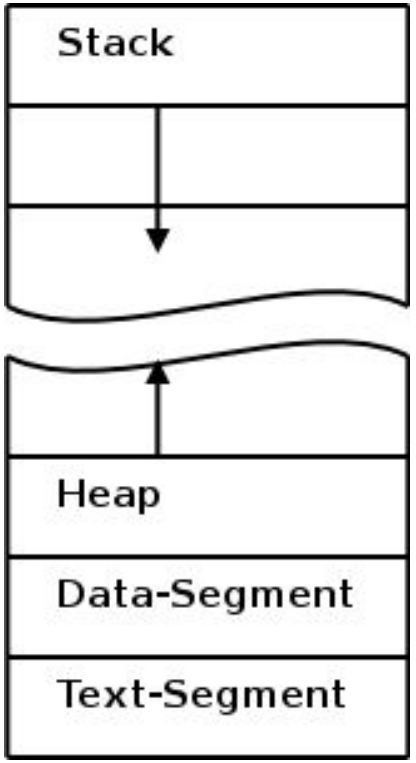


Abbildung 1: (Speicher-) Segmente

Grundlagen Prozessor – Register

- x86/IA-32 16 Basisregister
- Zwischenspeicher
- Extended Instruction Pointer (EIP)
- Extended Base Pointer (EBP)
- Extended Stack Pointer (ESP)

Stack – Funktionsweise

- Zwischenspeicher für lokale Variablen
- Beginnt am oberen Ende des Adressraums
- Wächst nach unten (LIFO)
- Es gibt zwei Basisoperationen
  - `push()`
  - `pop()`

Stack – Steuerung

- StackFrame
- Inhalt des StackFrames
- Call Stack

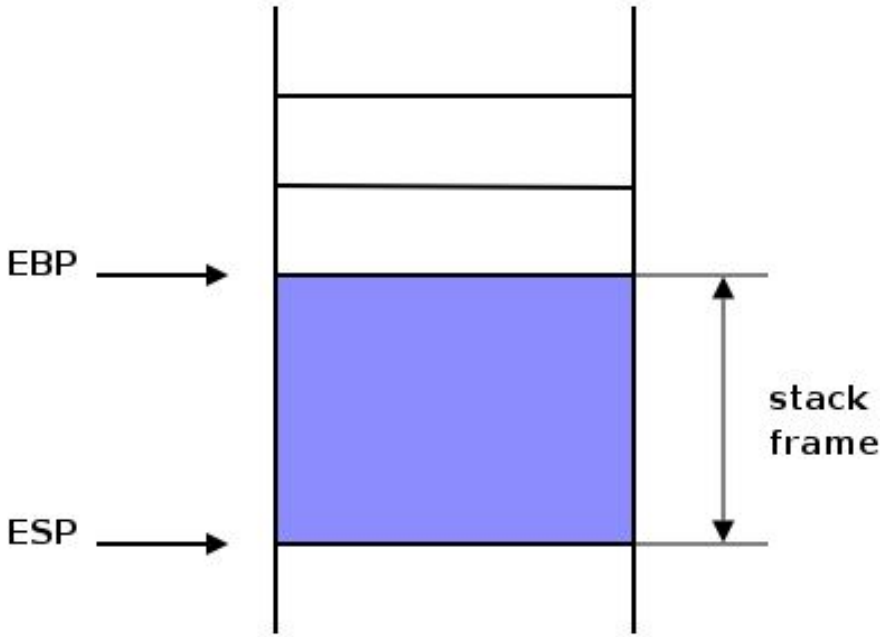


Abbildung 2: (Stack-) Frame

Stack – Funktionen

- Funktions-Prolog
  - Zustand des Stacks sichern
  - Speicher reservieren
- Funktions-Epilog
  - Zustand des Stacks wiederherstellen

Stack – Aufbau

```
01  void
02  function(const char* msg)
03  {
04      int     var1;
05      char    buf[80];
06      int     var2;
07
08      strcpy(buf, msg);
09  }
10  int
11  main(int argc, char** argv)
12  {
13      function(argv[1]);
14      return (0);
15  }
```

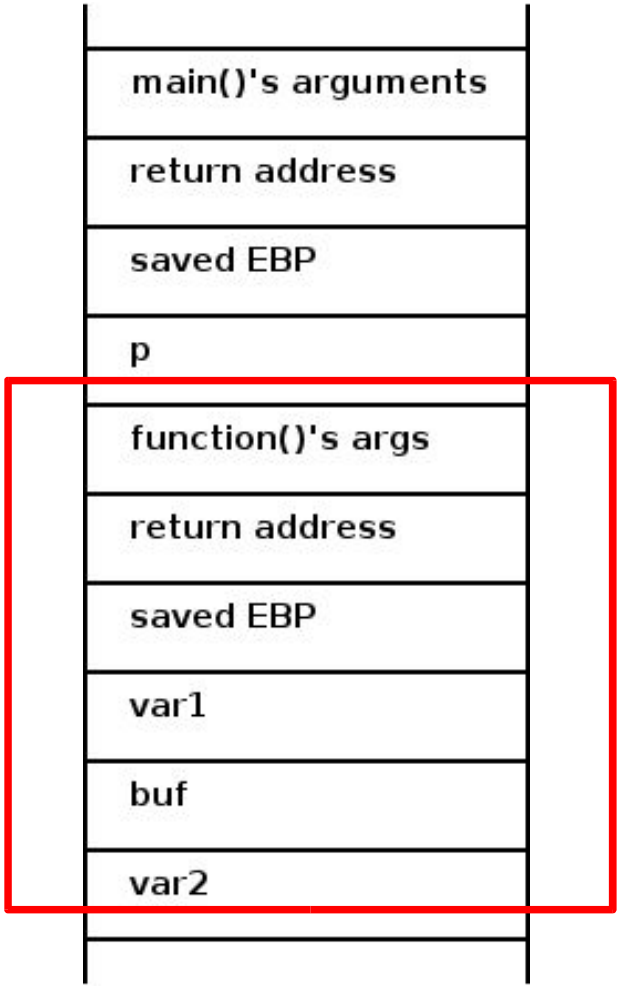


Abbildung 3: Stack. Vgl. [Richarte2002]

Angriffsmöglichkeiten durch BufferOverflows

- Denial of Service
- Modifikation des Programmflusses
- Ausführen eingeschleusten Programmcods
- Frame Pointer Overwriting

Angriffsziele auf dem Stack

- Rücksprungadresse
- Lokale Variablen
- Argument Variablen
- Svd. Frame Pointer

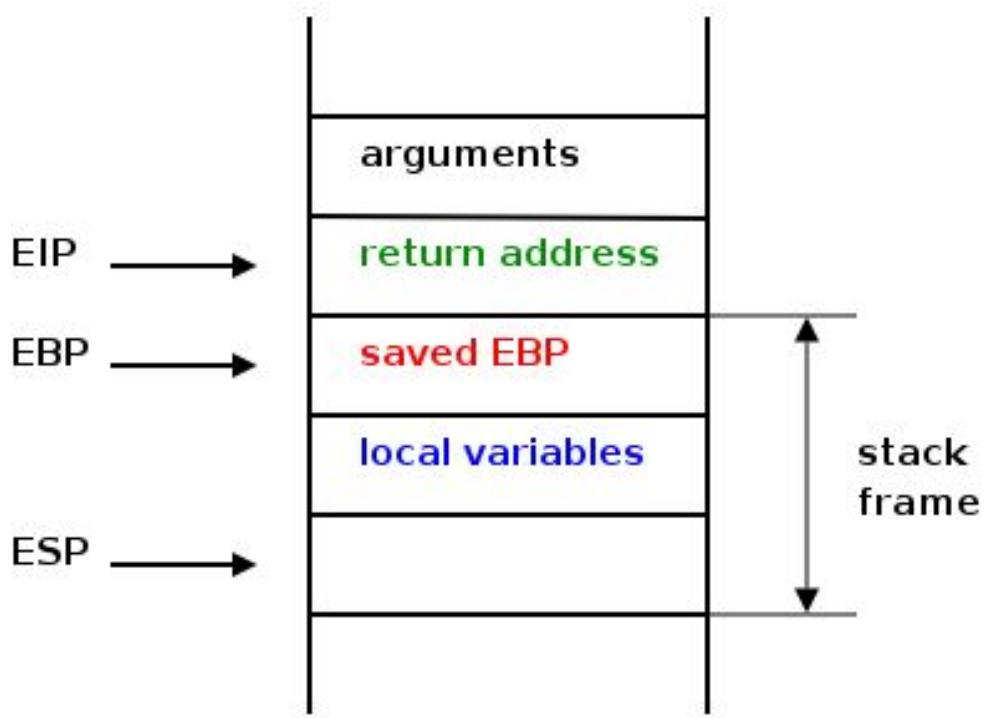
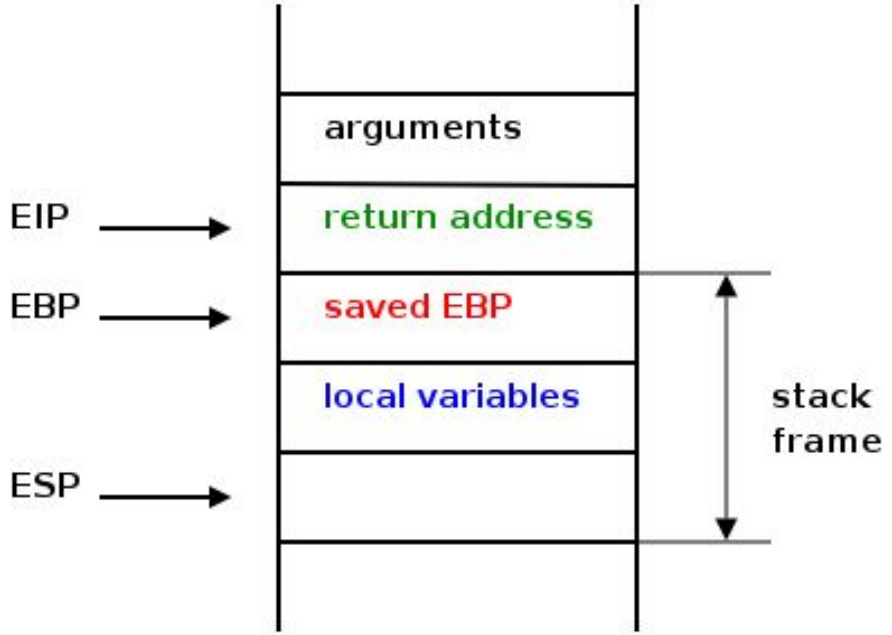


Abbildung 4: Angriffsziele (Stack)

Grundlagen GNU Debugger *gdb*

Abbildung 5: Stack




```
(gdb) info frame
Stack level 0, frame at 0xbfbaa890:
 eip = 0x8048360 in function (test3.c:4); saved eip 0x804837e
 called by frame at 0xbfbaa8a0
 source language c.
 Arglist at 0xbfbaa888, args: a=255
 Locals at 0xbfbaa888, Previous frame's sp is 0xbfbaa890
 Saved registers:
  ebp at 0xbfbaa888, eip at 0xbfbaa88c
```

Beispiel Stackinhalt

```
01  void
02  function(int a)
03  {
04      int var = a;
05  }
06
07  int
08  main(void)
09  {
10      function(255);
11      return (0);
12  }
```

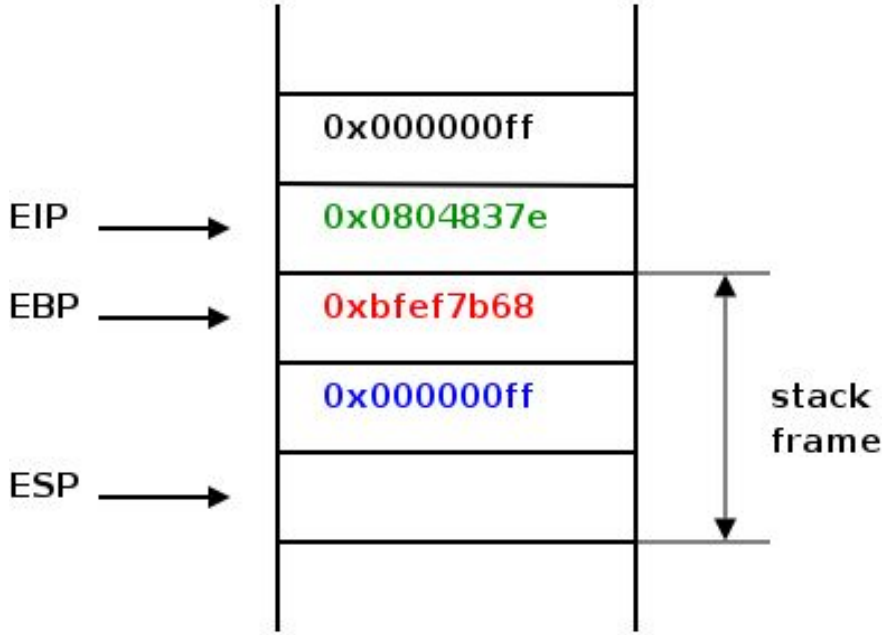


Abbildung 6: Stack

Ausgabe *gdb*

```
(gdb) x/4x &var
0xbfef7b54: 0x000000ff 0xbfef7b68 0x0804837e 0x000000ff
```

Klassischer BufferOverflow

```
01  void
02  function(const char* args)
03  {
04      char buffer[4];
05      strcpy(buffer, args);
06  }
07
08  int
09  main(int argc, char** argv)
10  {
11      function(argv[1]);
12      return (0);
13  }
```

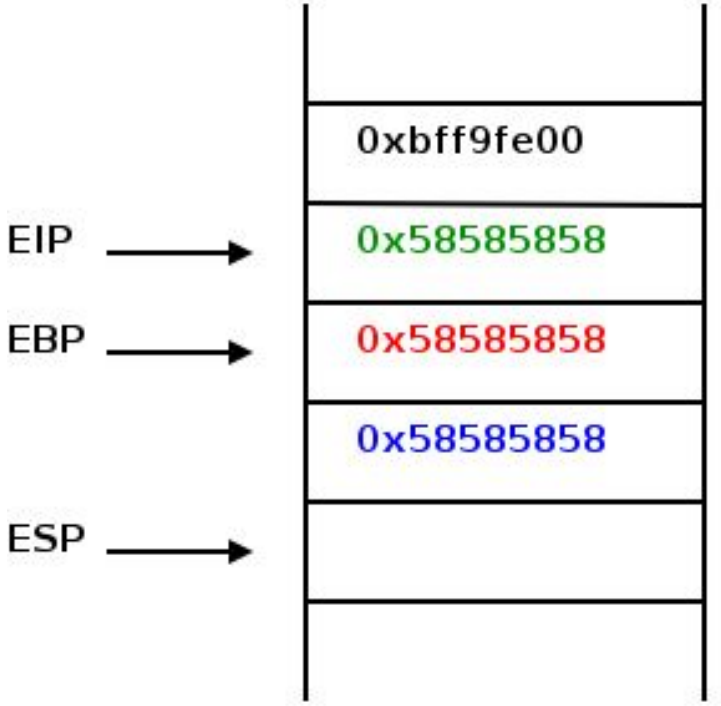


Abbildung 7: Stack - BufferOverflow

```
./faulty `perl -e 'print "X" x 12'`
```

Ausgabe *gdb*

```
(gdb) x/4x buffer
0xbfef7b54: 0x58585858 0x58585858 0x58585858 0xbff9fe00
```

BufferOverflow Bug ausnutzen

- Aufdecken des Fehlers
- Ziel festlegen
- Shellcode (Payload) erstellen
- Exploit schreiben
- Bug mit Hilfe des Exploits ausnutzen
- :-)

Shellcode erstellen – Beispiel *exit()*

Assembler Code

```
mov $1,%al
xorl %ebx,%ebx
int $0x80
```

Assemblieren

```
$ as -o exit.o exit.asm
```

Linken

```
$ ld -o exit exit.o
```

Disassemblieren des Maschinencodes

```
$ objdump -d exit
8048074: b0 01    mov $0x1,%al
8048076: 31 db    xor %ebx,%ebx
8048078: cd 80    int $0x80
```

Shellcode

```
\xb0\x01\x31\xdb\xcd\x80
```

Funktionalität eines Exploits (Beispiel)

- Feststellen des aktuellen Stack Pointers (ESP)
  - Trick:

```
01  unsigned long
02  getESP(void)
03  {
04      __asm__("movl %esp,%eax");
05  }
```

- „NOPs" einfügen
- Shellcode (Payload) einfügen
- Rücksprungadresse(n) einfügen

Funktionalität eines Exploits mit NOP-Sliding

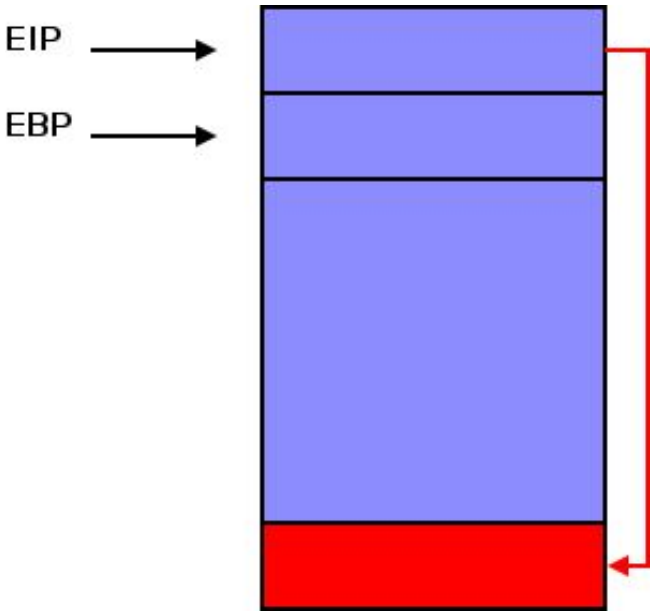


Abbildung 8: Stack nach Exploit
Klassische Methode. Vgl. [Klein2004]

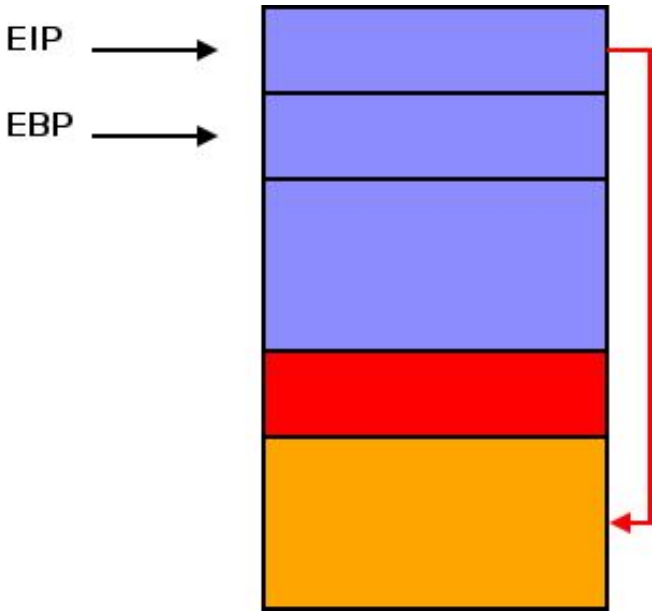


Abbildung 9: Stack nach Exploit
mit NOP-Sliding. Vgl. [Klein2004]

Zwischenzusammenfassung

- Register EIP, EBP, ESP
- Rücksprungadresse
- Payload (Shellcode)
- NOPs

System für Demo

- x86
- LINUX
- gcc – Version 2.95.2
- Kernel 2.2.16

Maßnahmen zur Gefahrenreduzierung

- Compiler-Erweiterungen
  - StackGuard
  - SSP
  - StackGhost (sparc)
  - Bounds Checking
- Wrapper für Bibliotheksfunktionen
  - Libsafe
- Modifikation der Prozessumgebung
  - Non-executable Stack
  - PaX

StackGuard – Überblick

- StackGuard schützt (bedingt) vor Angriffen
- Seit 1998 als Erweiterung für *gcc* verfügbar
- Der Verlust an Performance ist gering, aber vorhanden
- StackGuard verwendet eine spezielle Umgebung, die *Canary* genannt wird

StackGuard – Entwurf

- Es gibt drei verschiedene Arten von Canaries:
  - terminator canary (CR, EOF, LF, Null)
  - random canary
  - random xor canary
- StackGuard verwendet terminator canary

StackGuard – Implementierung

- Schutz der Rücksprungadresse
- Dieser Speicherbereich wird durch einen (terminator) Canary geschützt
- Überprüfung des Canarys
- Canary – Wert:
  - Falsch
  - Richtig

StackGuard – Funktionsweise

- Prolog der Funktion:
  - Canary anlegen

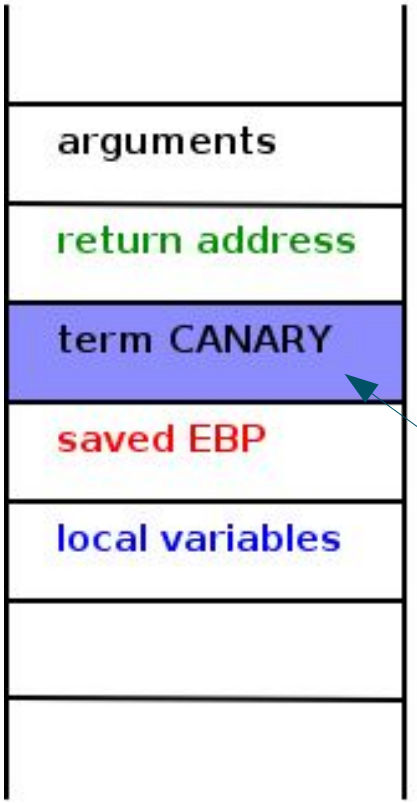
```
function_prologue:
    pushl   $0x000aff0d
    pushl   %ebp
    mov     %esp, %ebp
    subl    $108, %esp

(function body)
```

- Epilog der Funktion:
  - Canary kontrollieren

```
function_epilogue:
    leave
    cmpl    $0x000aff0d,(%esp)
    jne     canary_changed
    addl    $4, %esp
    ret
```

StackGuard – Stackansicht



(Terminator) – Canary Bereich

Abbildung 11: Stack mit (Terminator) Canary

Stack Smashing Protector – Überblick

- Als Erweiterung für *gcc* verfügbar
- Neue Idee, um bekannte Methoden (StackGuard) zu verbessern
- SSP verwendet unter anderem ebenfalls einen Canary-Bereich
- Schützt vor den gängigsten Angriffsmethoden
- SSP ist deutlich performanter als StackGuard
- Seit Dezember 2002 in OpenBSD integriert

Stack Smashing Protector – Entwurf

- Verwendung eines Canary-Bereichs (random)
- Reorganisation von lokalen Variablen
- Kopieren von Funktionsargumenten

Stack Smashing Protector – Implementierung

- Der Focus von SSP liegt auf folgenden Bereichen:
  - Lage der Argumente der Funktion
  - Rücksprungadresse
  - Gesicherter Frame Pointer (EBP)
  - Lokale Variablen
- Ein Canary-Bereich schützt die ersten drei aufgeführten Bereiche
- Die Stackbenutzung wird restriktiver behandelt (Reorganisation)

Stack Smashing Protector – Funktionsweise

- Prolog der Funktion

```
function_prologue:
    pushl   %ebp
    mov     %esp,%ebp
    subl    $272,%esp
protection_prologue:
    movl    __guard, %eax
    movl    %eax, -24(%ebp)

(function body)
```

- Epilog der Funktion

```
protection_epilogue:
    movl    -24(%ebp), %edx
    cmpl    __guard, %edx
    je      standard_epilogue

    movl    -24(%ebp), %eax
    pushl   %eax
    pushl   $function_name
    call    __stack_smash_handler
    addl    $8, %esp
standard_epilogue:
    ...
```

Stack Smashing Protector – Stackansicht

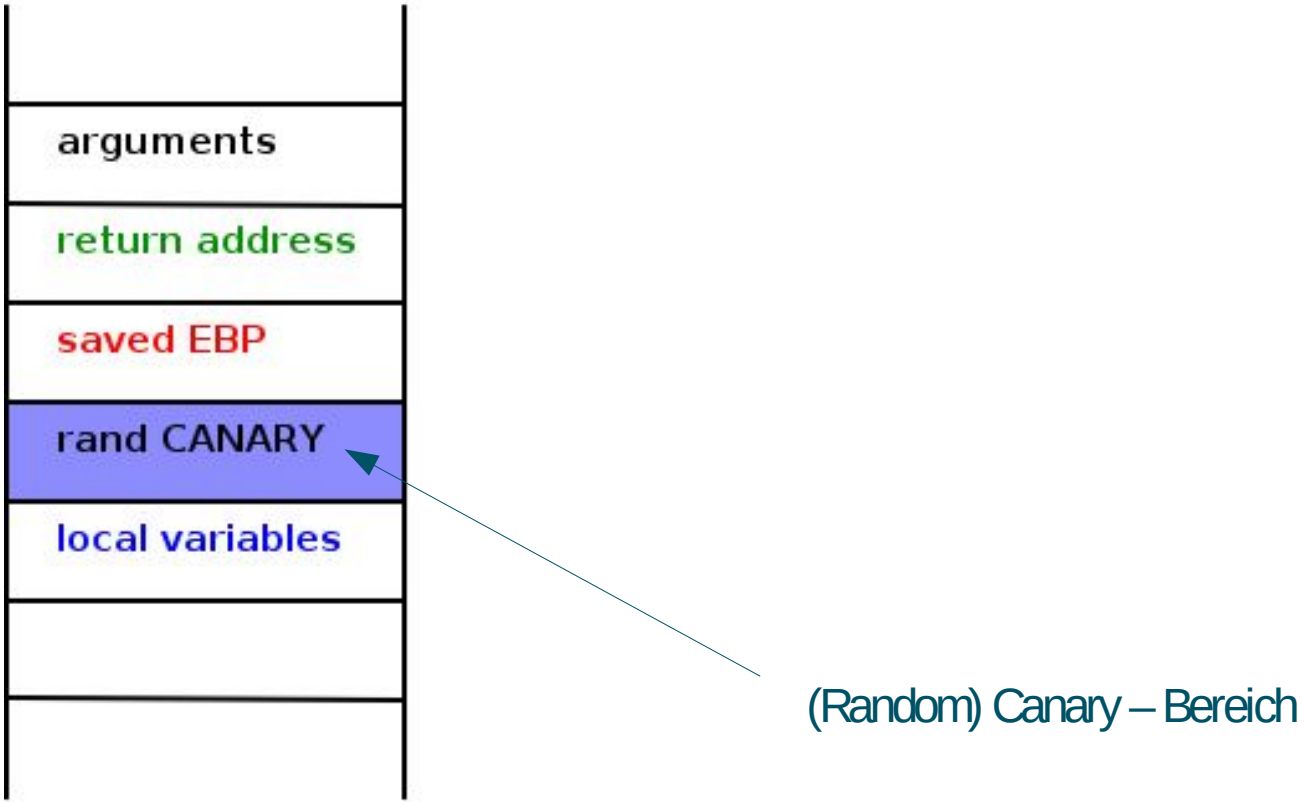


Abbildung 12: Stack mit (Random) Canary

Stack Smashing Protector – Reorganisation der Variablen

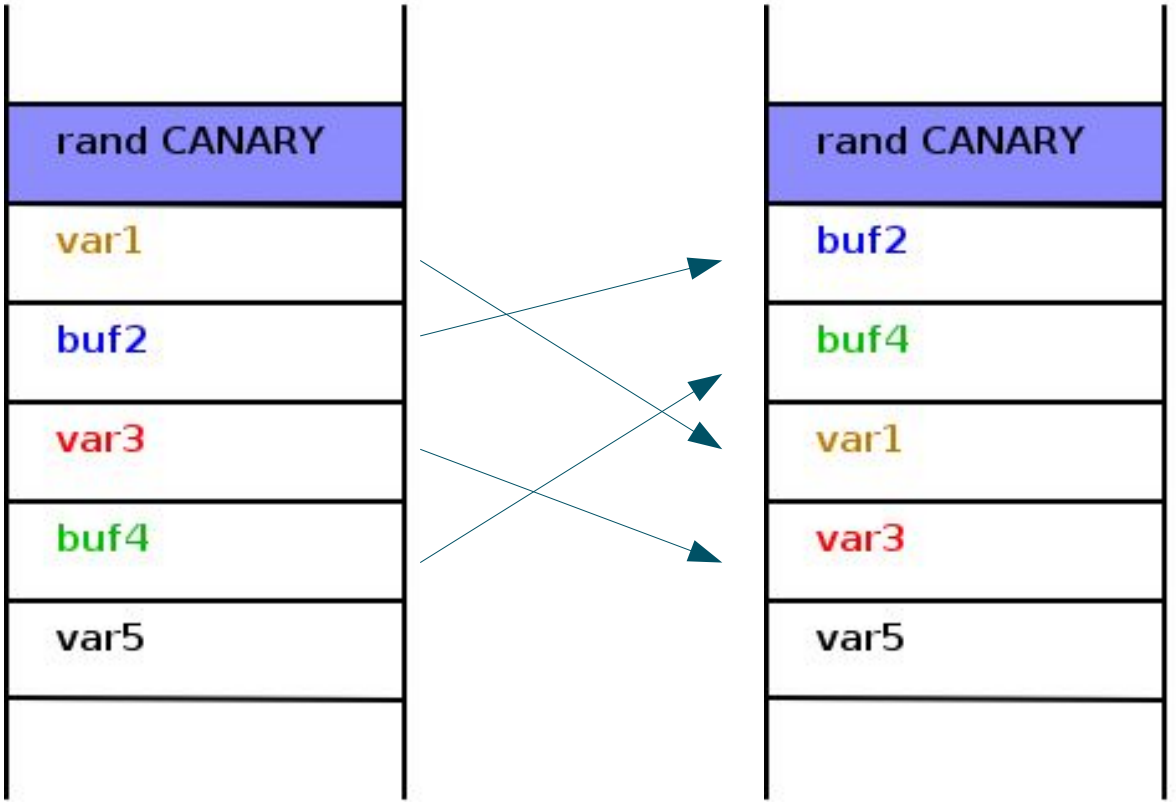


Abbildung13: Stack vor Reorganisation.
Vgl. [Richarte2002]

Abbildung14: Stack nach Reorganisation.
Vgl. [Richarte2002]

Stack Smashing Protector – Kopieren der Argumente

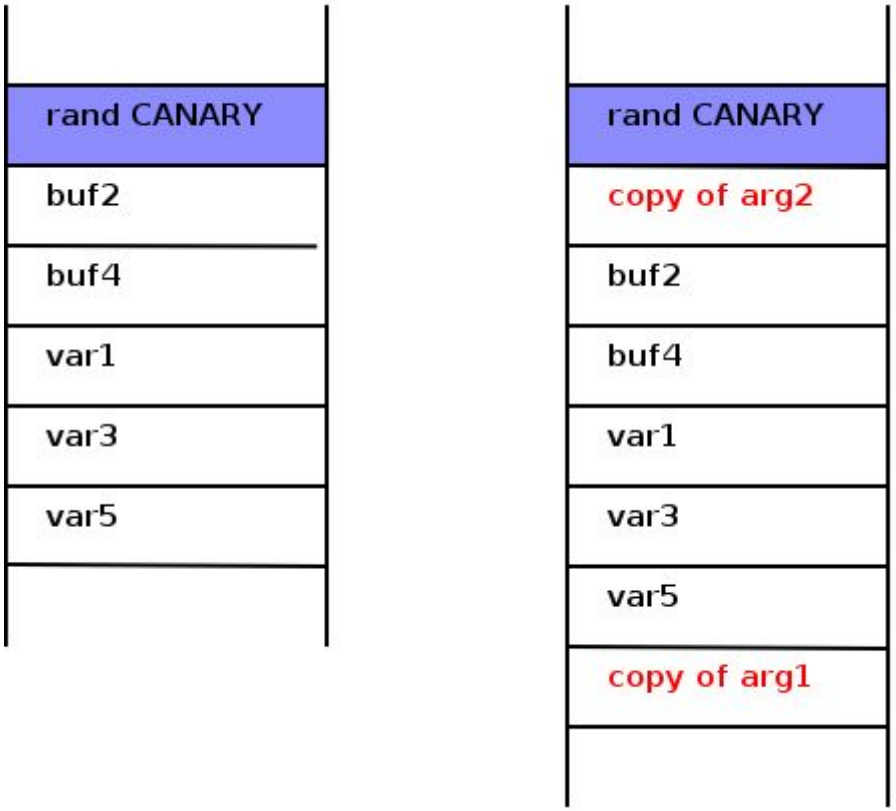


Abbildung15: Stack vor und nach Kopie. Vgl. [Richarte2002]

Fazit

- Sicherheitslücken durch BufferOverflows dürfen auf keinen Fall unterschätzt werden
- Die Ausnutzung einer solchen Schwachstelle ist jedoch nicht trivial
- Es existieren gute Schutzmechanismen, nutzen!

Literatur

Debugging with gdb
URL:https://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/gdb/stack.html
(Stand: November 2005).

[Etho2000] Etho, Hiroaki. Protecting from stack-smashing attacks.
URL:http://www.research.ibm.com/trl/projects/security/ssp/main.html (Stand: November 2005)

[Kallnik2001] Kallnik, Stephan; Pape, Daniel; Schrter, Daniel; Strobel, Stefan:
Das Sicherheitsloch. URL:http://www.heise.de/ct/01/23/216/ (Stand: November 2005).

[De Raadt2005] De Raadt, Theo: Exploit Mitigation Techniques.
URL:http://openbsd.informatik.uni-erlangen.de/papers/ven05-deraadt/index.html
(Stand: November 2005).

[Wagle] Wagle, Perry; Cowan, Crispin: StackGuard: Simple Stack Smash Protection for GCC.
URL:http://gcc.fyxm.net/summit/2003/Stackguard.pdf (Stand: November 2005).

[One] Aleph One: Smashing The Stack For Fun And Profit. Phrack Magazine #49 Artikel 14.
URL:http://www.phrack.org (Stand: November 2005)

Literatur

[Hildebrand2005] Hildebrand, Matthew: Interview with Theo de Raadt on Industry and Free Software. 05.07.2005, URL: http://www.theepochtimes.com/news/5-7-5/30084.html
(Stand: Dezember 2005).

[Klein2004] Klein, Tobias: Buffer Overflows und Format-String-Schwachstellen. dpunkt.verlag Heidelberg 2004. 1. Aufl.

[Richarte2002] Richarte, Gerardo: Four different tricks to bypass StackShild and StackGuard protection. 9.4.2002 – 3.6.2004, URL:http://www.coresecurity.com/files/files/11/StackguardPaper.pdf
(Stand: November 2005)

[De Raadt2004] De Raadt, Theo: OpenBSD's Theo de Raadt talks software security. 10.09.2004, URL:http://www.computerworld.com.au/index.php/id;1498222899;fp;16;fpid;0
(Stand: Oktober 2005)

Vielen Dank für die Aufmerksamkeit