Schutz vor Buffer-Overflow und Format-String Schwachstellen

Jan Kohlrausch

DFN-CERT GmbH

Oberstraße 14b

20144 Hamburg

kohlrausch@cert.dfn.de

DFN CERT

Gliederung

- Einleitung und Motivation
- Buffer-Overflow und Format-String Schwachstellen
- Präventive Schutzmöglichkeiten
- Zusammenfassung und Ausblick

DFN CERT

Einleitung und Motivation

- Buffer-Overflow Schwachstellen
  - Durch unsichere Speicherverwaltung
  - Lange bekannt (z.B. Ausnutzung durch Morris Wurm 1988)
  - Immer noch aktuell
- Format-String Schwachstellen
  - Betreffen unsichere Verwendung von Ausgabebefehlen
  - Ausnutzung bekannt seit Mitte 2000
  - Weniger bekannt als Buffer-Overflows

Einleitung und Motivation

- Wichtige sicherheitsrelevante Fehlerklassen in C-Programmen
- Teilweise über das Netzwerk ausnutzbar
- Viele verletzliche Programme
  - Buffer-Overflow Schwachstellen z.B. in
    - Microsoft Internet Information Server (IIS)
    - Nameserver Bind
    - Verschiedene Dienste (z.B. Solaris dtspcd)
    - Sshd Server
  - Format-String Schwachstellen z.B. im Wu-ftpd Server

Einleitung und Motivation

- Reaktive Maßnahmen
  - Einspielen von Patches
  - Workarounds

⇒ Kein Schutz vor noch nicht veröffentlichten oder unbekannten Schwachstellen

⇒ Präventive Maßnahmen notwendig:

  - Härten des Systems
  - Schutz der SetUID/SetGID Programme
  - Schutz der über das Netzwerk angebotenen Dienste

Buffer-Overflow Schwachstellen

- Speicherbelegung von Prozessen:
  - Codesegment: für ausgeführten Programmcode
  - Heapsegment: globale Variablen, dynamisch angelegte Variablen und Konstanten
  - Stack-Segment: Aktivierungssegmente für Funktionsaufrufe
    - Lokale Variablen
    - Übergebene Argumente
    - Rücksprungadresse

Buffer-Overflow Schwachstellen

- Ablauf einer Funktion in C:
  - Sichern der Argumente auf dem Stack
  - Sichern der Rücksprungadresse auf dem Stack
  - Sprung in Funktion
  - Für lokale Variablen wird Speicherplatz auf dem Stack reserviert
  - Zurückspringen an die Rücksprungadresse bei Beendigung der Funktion

⇒ viele sicherheitsrelevante Daten auf dem Stack (insbesondere die Rücksprungadresse) !

Buffer-Overflow Schwachstellen

- Buffer-Overflow ist Überschreiben der Grenze eines Speicherbereichs (buffer)
- Buffer-Overflow Schwachstellen möglich durch:
  - Kein Schutz der Speichergrenzen von Variablen in C-Programmen
  - Fehlerquellen durch unsichere String-Operationen
    - z.B. `strcpy(buf1,buf2)`
  - Keine Erkennung von unsicheren Pointeroperationen
- Beispiel: Überschreiben der Rücksprungadresse auf dem Stack (*Stack-Smashing*)

DFN CERT

Buffer-Overflow Schwachstellen

```
char *helloworld = "hello world \n";

void vulnerable(char *str)
{
  char *str2;
  unsigned int a = 65534;
  char buf[32];
  unsigned int b = 65535;

  strcpy(buf,str);
  printf(buf); printf("\n");
}
```

Buffer-Overflow Schwachstellen

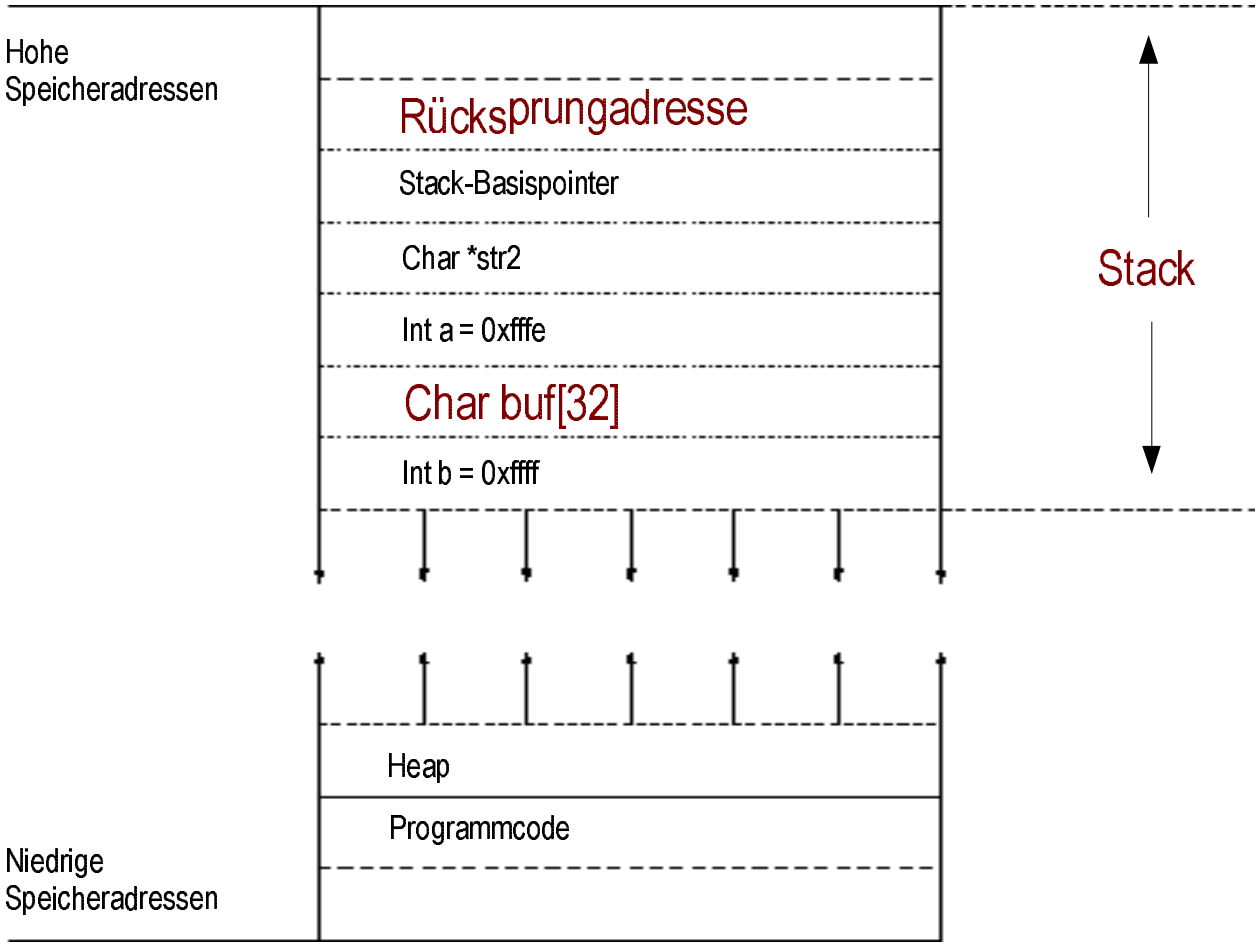
- Buffer-Overflow durch unsichere Verwendung der Funktion `strcpy(buf,str);`
- `Buf` ist auf 32 Bytes begrenzt
- Keine Längenbegrenzung für `str`
- Benutzerargument in `str` wird in `buf` kopiert
- `strcpy` berücksichtigt nicht die begrenzte Größe von `buf`

Buffer-Overflow Schwachstellen

- Überschreiben der Rücksprungadresse mit beliebigen Zeichen (Denial of Service Angriff):

```
marvin:~ > ./testprogramm AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA
Adresse der Variablen "helloworld":8048424
Speicherzugriffsfehler (core dumped)
```

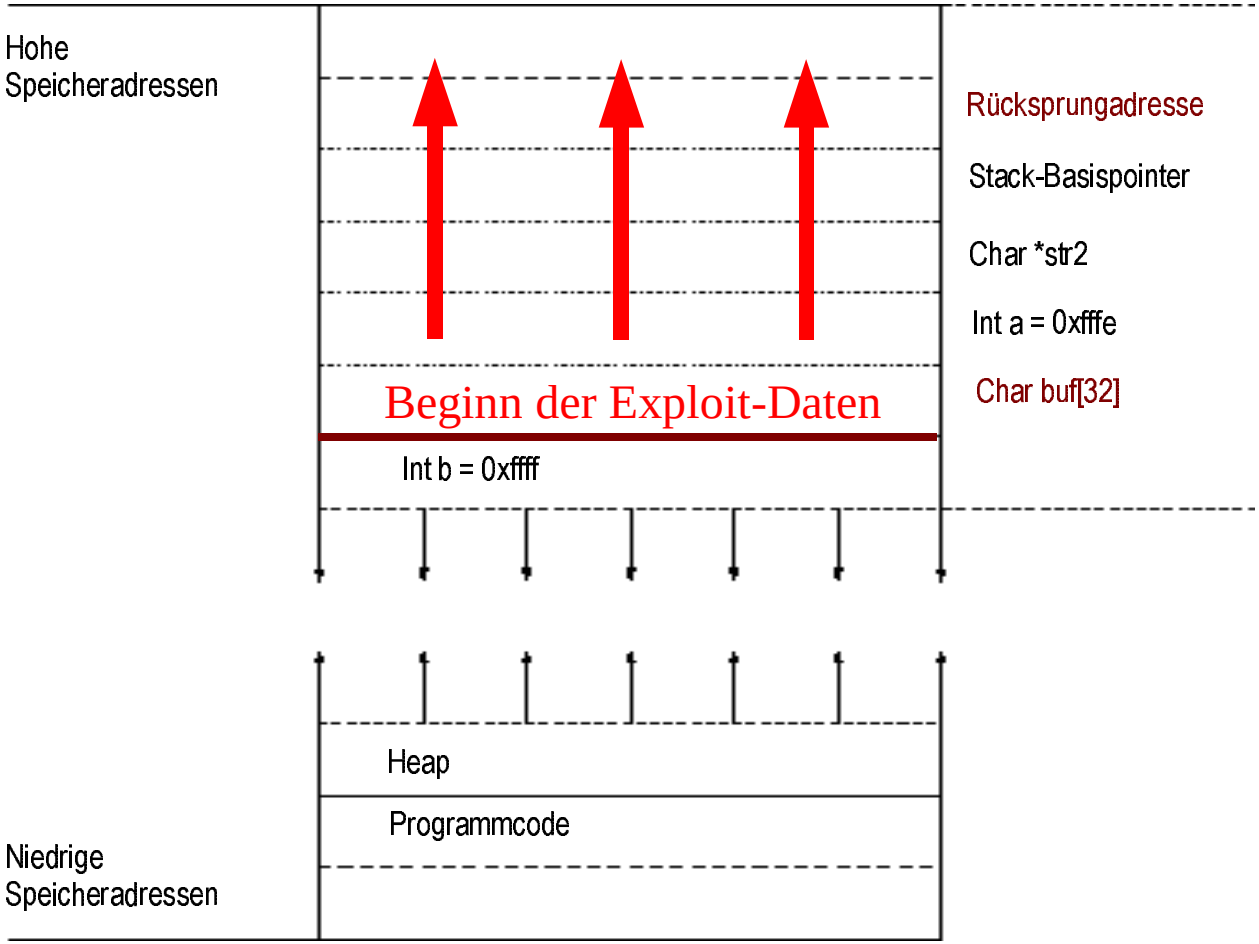
Buffer-Overflow Schwachstellen



Speicherbelegung des Testprogramms

DFN CERT

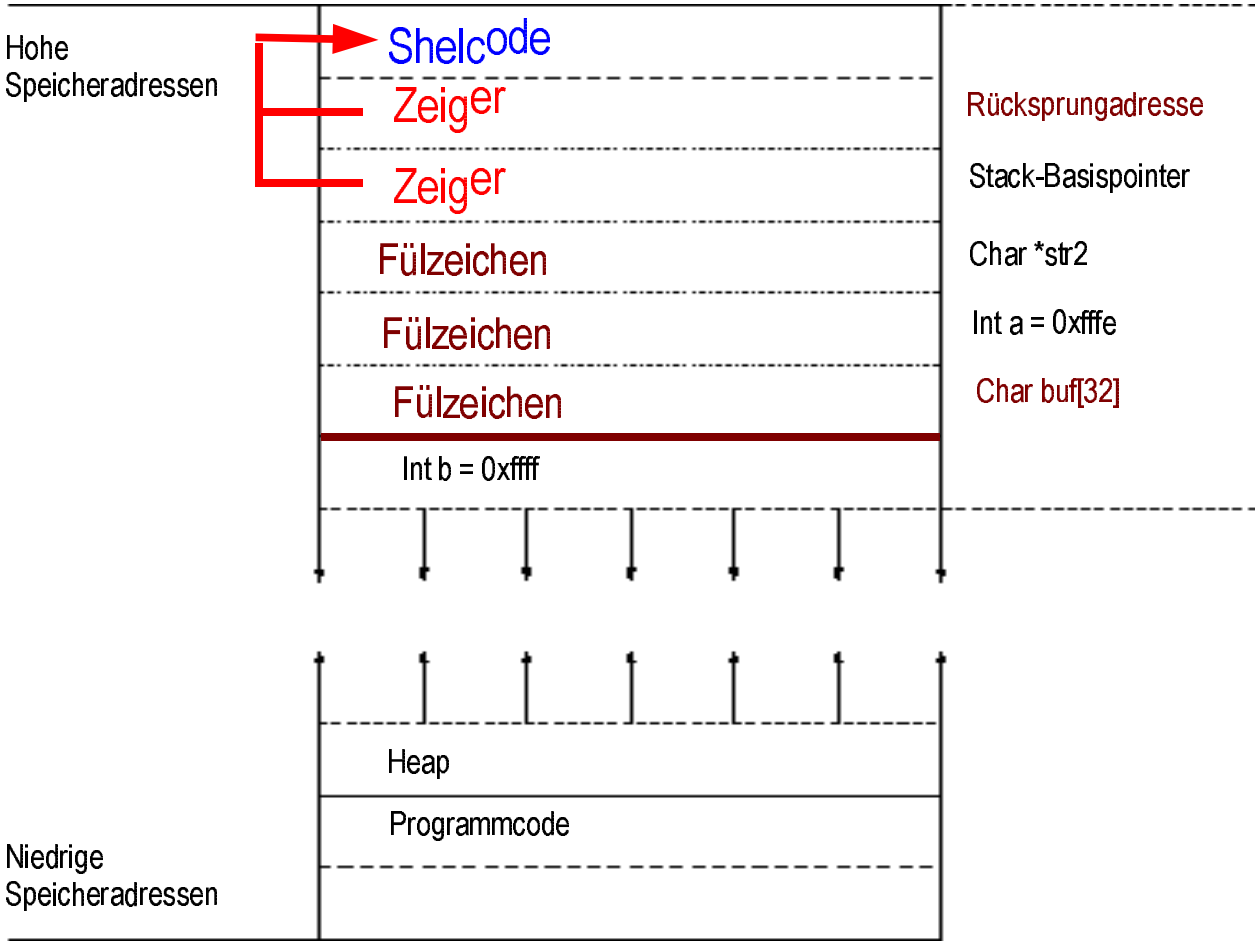
Buffer-Overflow Schwachstellen



Überfluten von `buf` durch `strcpy(buf,str)`

DFN CERT

Buffer-Overflow Schwachstellen



Vorgehensweise eines Exploits

DFN CERT

Buffer-Overflow Schwachstellen

- Vorgehensweise des “Stack-Smashings”
  - Ablegen von Shellcode im Prozeßspeicher
  - Überschreiben der Rücksprungadresse mit Zeiger auf Shellcode beim Buffer-Overflow
  - Nach Ablauf der Funktion wird der Shellcode ausgeführt
  - Shellcode startet häufig Betriebssystem-Shell durch die der Angreifer beliebige Befehle ausführen kann

Format-String Schwachstellen

- Format-Strings und Formatangaben:
  - `printf(Format-String, Argument1...);`
  - z.B. `printf("Text: %s", msg);`
    - `%s` ist Formatangabe mit Argument `msg`
  - Für jede Formatangabe (%x, %s, %n) im Format-String muß ein Argument angegeben werden
- Formatangaben:
  - %x: Integervariable, Argument `int`
  - %s: Stringvariable, Argument `char *`
  - %n: Anzahl ausgegebener Zeichen, Argument `int *`

Format-String Schwachstellen

- Schwachstelle, wenn Format-String ohne explizite Argumente angegeben ist, z.B.:
  - `printf(name)` oder `syslog(msg)`
- Ausnutzbar, wenn Angreifer Format-String beeinflussen kann.
- In Programmen häufig verwendet:
  - `syslog(msg)` statt `syslog("%s", msg)`
  - Ergibt gewünschte Log-Meldung ?
  - Wie wird z.B. `printf("Fehler:%s")` ausgewertet ?
- Formatangaben (z.B. %x, %s, %n) werden auch ohne explizite Argumente ausgewertet:
  - Es wird jeweils der oberste Wert vom Stack verwendet

DFN CERT

Format-String Schwachstellen

```
char *helloworld = "hello world \n";

void vulnerable(char *str)
{
  char *str2;
  unsigned int a = 65534;
  char buf[32];
  unsigned int b = 65535;

  strcpy(buf,str);
  printf(buf); printf("\n");
  /* sicher wäre printf(“%s”,buf) */
}
```


DFN CERT

Format-String Schwachstellen

- Ausnutzen: Auslesen des Stackinhaltes

```
jan@marvin:~ > ./testprogramm "%x %x%x%x%x%x%x%x%x %x %x %x %x"
Adresse der Variablen "helloworld":8048424
ffff 252078252578257825782578257825782078257825207825782520787825
2520 fffe bffff874 bffff8a8 80483b4
```

- Interpretierung der Ausgabe:
  - `25207825257...` Ist Zeichenkette `"%x %x%x%x%..."` in kodierter Form
  - `80483b4` ist die Rücksprungadresse

DFN CERT

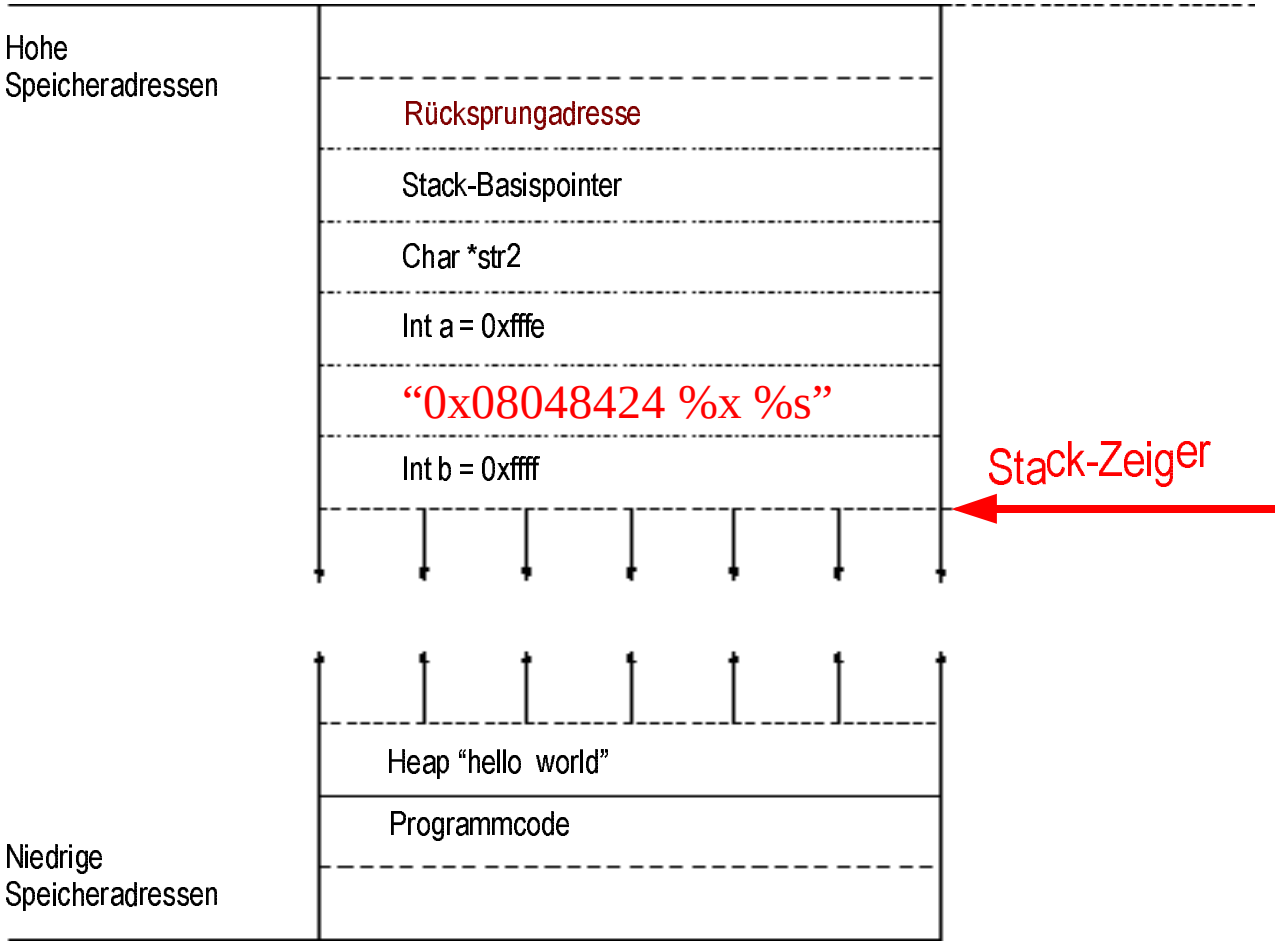
Format-String Schwachstellen

- Auslesen von beliebigen Speicherinhalten: Beispielsweise die globale Variable `helloworld` auf dem Heap

```
jan@marvin:~ > ./testprogramm "\x24\x84\x04\x08%x%s"
Adresse der Variablen "helloworld":8048424
\x24\x84\x04\x08ffffhello world
```

- Im Folgenden: Wie wird die Eingabe `"\x24\x84\x04\x08%x%s"` ausgewertet ?

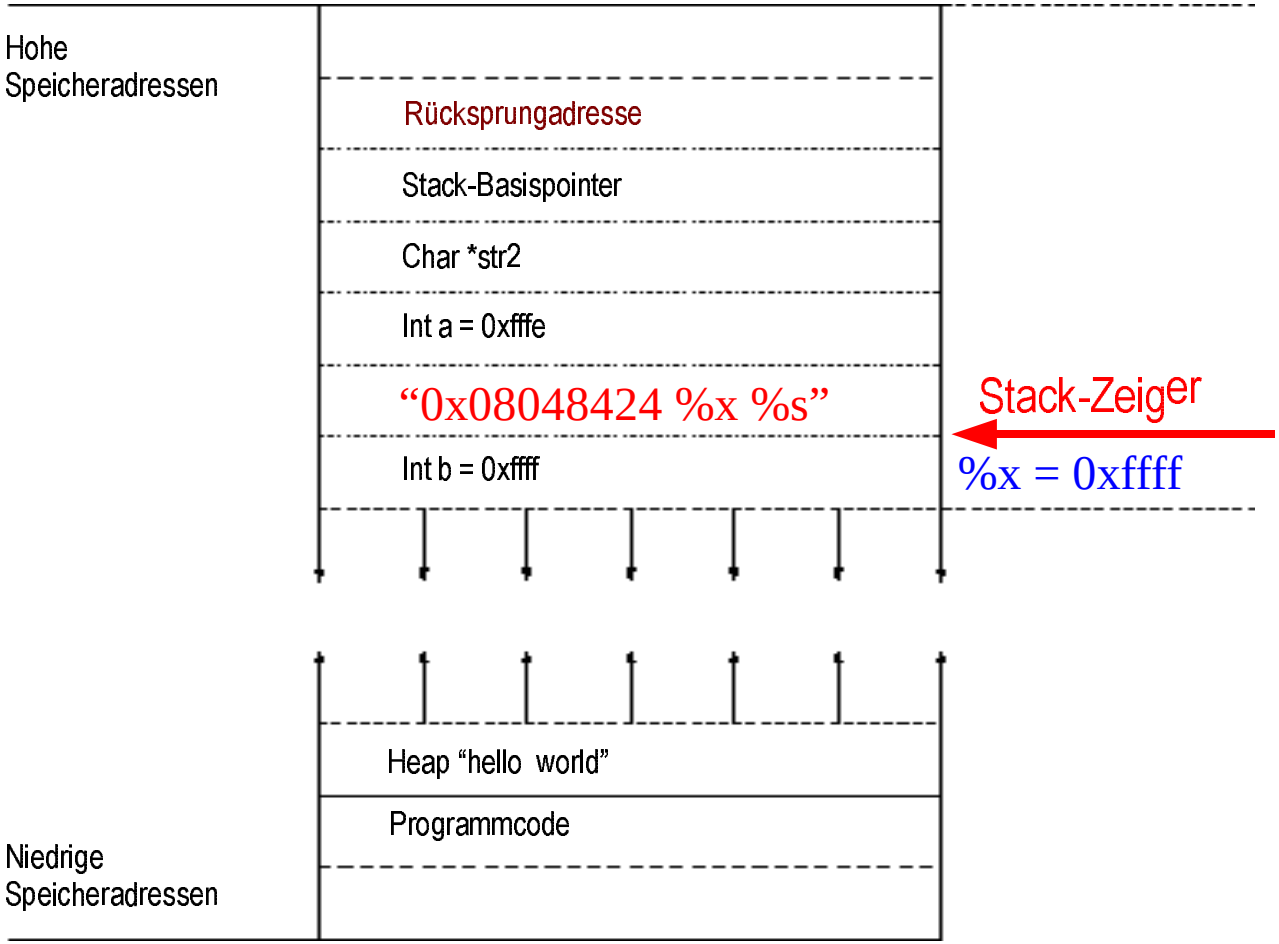
Format-String Schwachstellen



Ausführung von `strcpy(buf,str)`

DFN CERT

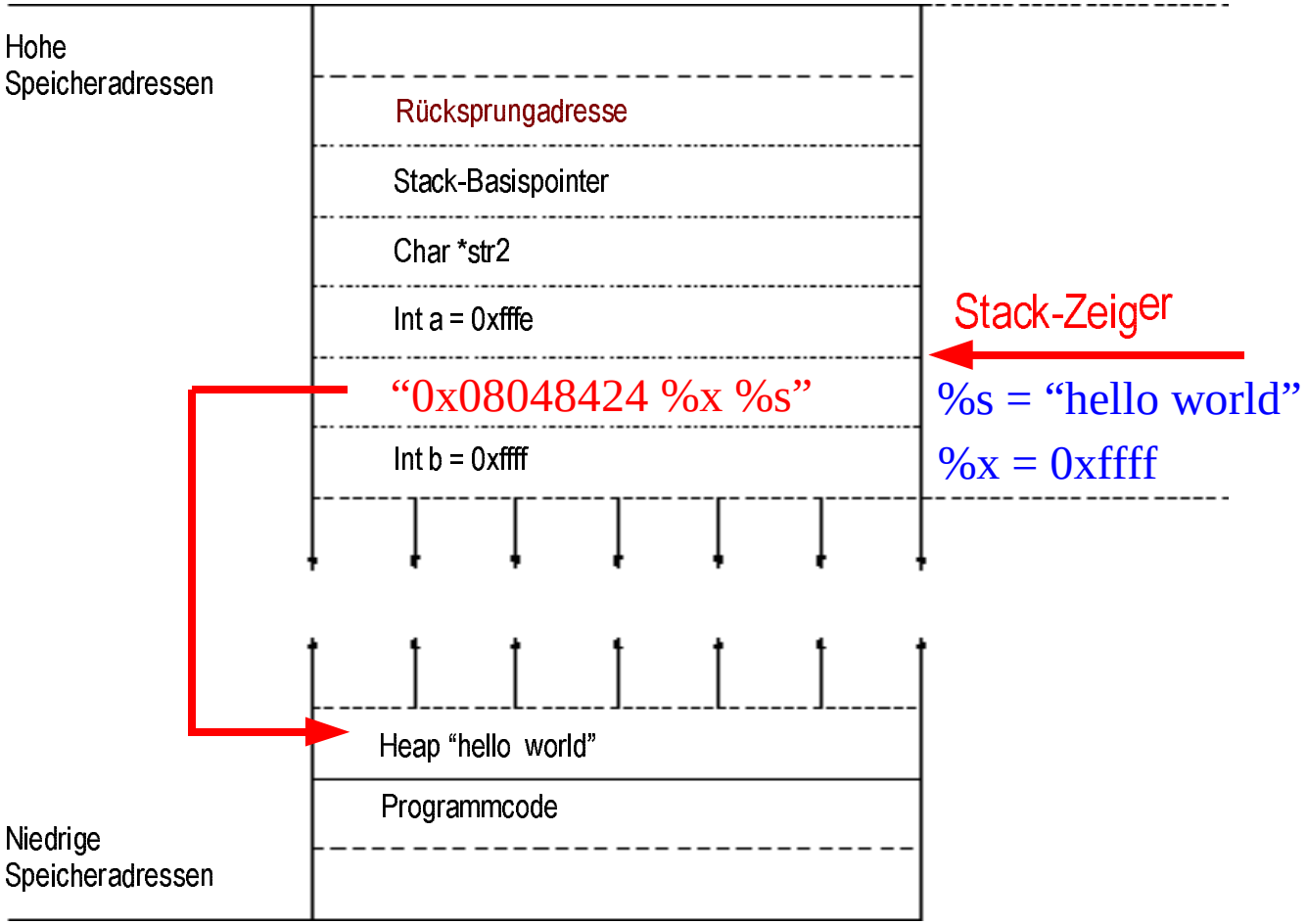
Format-String Schwachstellen



Ausführung von `printf(str)`

DFN CERT

Format-String Schwachstellen



Ausführung von `printf(str)`

DFN CERT

Format-String Schwachstellen

- Wiederholung: Auslesen der globalen Variable `helloworld`
- Ausgabe des Testprogramms:

```
jan@marvin:~ > ./testprogramm "\x24\x84\x04\x08%x%s"
Adresse der Variablen "helloworld":8048424
\x24\x84\x04\x08ffffhello world
```


DFN CERT

Format-String Schwachstellen

- Überschreiben beliebiger Zeiger mit %n möglich
- %n schreibt die Anzahl der ausgegebenen Zeichen in Integervariable, deren Adresse als Argument angegeben wird.
- Häufig wird Rücksprungadresse auf dem Stack überschrieben

⇒ Ausnuztung ähnlich wie bei Buffer-Overflows:

  – Überschreiben der Rücksprungadresse
  – Ausführung von Shellcode nach dem Verlassen der Funktion

Präventive Schutzmöglichkeiten

- Sicherer Programmierstil
- Manuelle oder automatische Analyse des Quelltextes
- Schutz auf Betriebssystemebene
- Schutz während der Programmausführung

Präventive Schutzmöglichkeiten

- Sicherer Programmierstil
  - Vorteile:
    - Vermeiden von Schwachstellen
  - Nachteile:
    - Hilft nicht, wenn fertige Programme installiert werden (es sei denn Vertrauen in Programmierer ist vorhanden (Open Source))
    - Übersehen von Fehlern möglich

Präventive Schutzmöglichkeiten

- Manuelle oder automatische Analyse des Quelltextes
  - Vorteile:
    - Vermeiden von Schwachstellen
  - Nachteile:
    - Technisches Wissen notwendig
    - Eigene Erstellung von Patches erforderlich
    - Programme häufig zu komplex (manuelle Analyse)
    - Falschmeldungen für sichere Codesegmente und Übersehen von unsicheren Codesegmenten nicht auszuschließen (automatische Analyse)

Präventive Schutzmöglichkeiten

- Schutz auf Betriebssystemebene (1)
  - Einschränkung der Privilegien des Programms (*least privilege*): Firewall, Deaktivieren von Diensten
  - Vorteile:
    - Keine Änderung der Programme notwendig
    - Generischer Ansatz
  - Nachteile:
    - Ausnutzen der Schwachstelle wird nicht verhindert
    - Angreifer kann immer noch Schaden anrichten (z.B. Verändern der Homepage...)

Präventive Schutzmöglichkeiten

- Schutz auf Betriebssystemebene (2)
  - Verhindern des Ausführens von Code auf dem Stack und Heap
  - Vorteile:
    - Keine Änderungen an Programmen notwendig
    - Generischer Ansatz
    - Exploits können gestoppt werden
  - Nachteile:
    - Kein vollständiger Schutz
    - Einige Programme benötigen ausführbaren Stack

Präventive Schutzmöglichkeiten

- Schutz während der Programmausführung (1)
  - Schutz der Speichergrenzen von Objekten (Variablen)
  - Vorteile:
    - Erkennen von Buffer-Overflows während der Programmausführung
  - Nachteile:
    - Programm muß neu übersetzt werden
    - Deutliche Performanzeinbußen (Faktor mindestens 3x)
    - Kompatiblitätsprobleme möglich

Präventive Schutzmöglichkeiten

- Schutz während der Programmausführung (2)
  - Schutz vor dem Überschreiben der Rücksprungadresse
  - Vorteile:
    - Erkennen von Exploits während der Programmausführung
  - Nachteile:
    - Programm muß z.T. neu übersetzt werden
    - Kein vollständiger Schutz

Präventive Schutzmöglichkeiten

- Im Folgenden: Diskussion der Ansätze
  - Verhindern des Ausführens von Code auf dem Stack und Heap
  - Schutz vor dem Überschreiben der Rücksprungadresse
- Motivation:
  - Schutz vor dem Ausnutzen von Buffer-Overflow und Format-String Schwachstellen
  - Nur geringe Performanzeinbußen
  - Wenig technisches Wissen notwendig

Präventive Schutzmöglichkeiten

Verhindern des Ausführens von Code

- Ansätze für Stack und Heap:
  - “non-executable” Stack ab Solaris 2.6
  - Linux: Kernelpatch von “Solar designer” realisiert non-executable” Stack
  - PaX für: Verhinderung der Codeausführung in beliebigen Speichersegmenten (Stack und Heapsegment)
- Schutz:
  - Alle Exploits scheitern, die Shellcode auf dem Stack oder Heap ausführen

Präventive Schutzmöglichkeiten

Verhindern des Ausführens von Code

- Grenzen der Ansätze:
  - “non-executable” Stack läßt sich leicht umgehen (“security by obscurity”)
  - Klasse der “return-into-libc” Exploits läßt sich durch diese Ansätze nicht vollständig stoppen (keine Ausführung von Shellcode, sondern direkte Verwendung von libc Funktionen)
    - PaX und Patch von “Solar desiger” erschweren “return-into-libc” Exploits
  - Kein Verhindern von Denial of Service Angriffen

Präventive Schutzmöglichkeiten

Überschreibungsschutz von Zeigern

- Ansatz:
  - GCC Compiler-Patches StackGuard und Stackshield
- Vorgehensweise:
  - Absicherung der Rücksprungadresse durch zusätzlichen Code, der vor dem Aufruf bzw. Verlassen einer Funktion ausgeführt wird
  - Vor dem Verlassen der Funktion wird Integrität der Rücksprungadresse überprüft

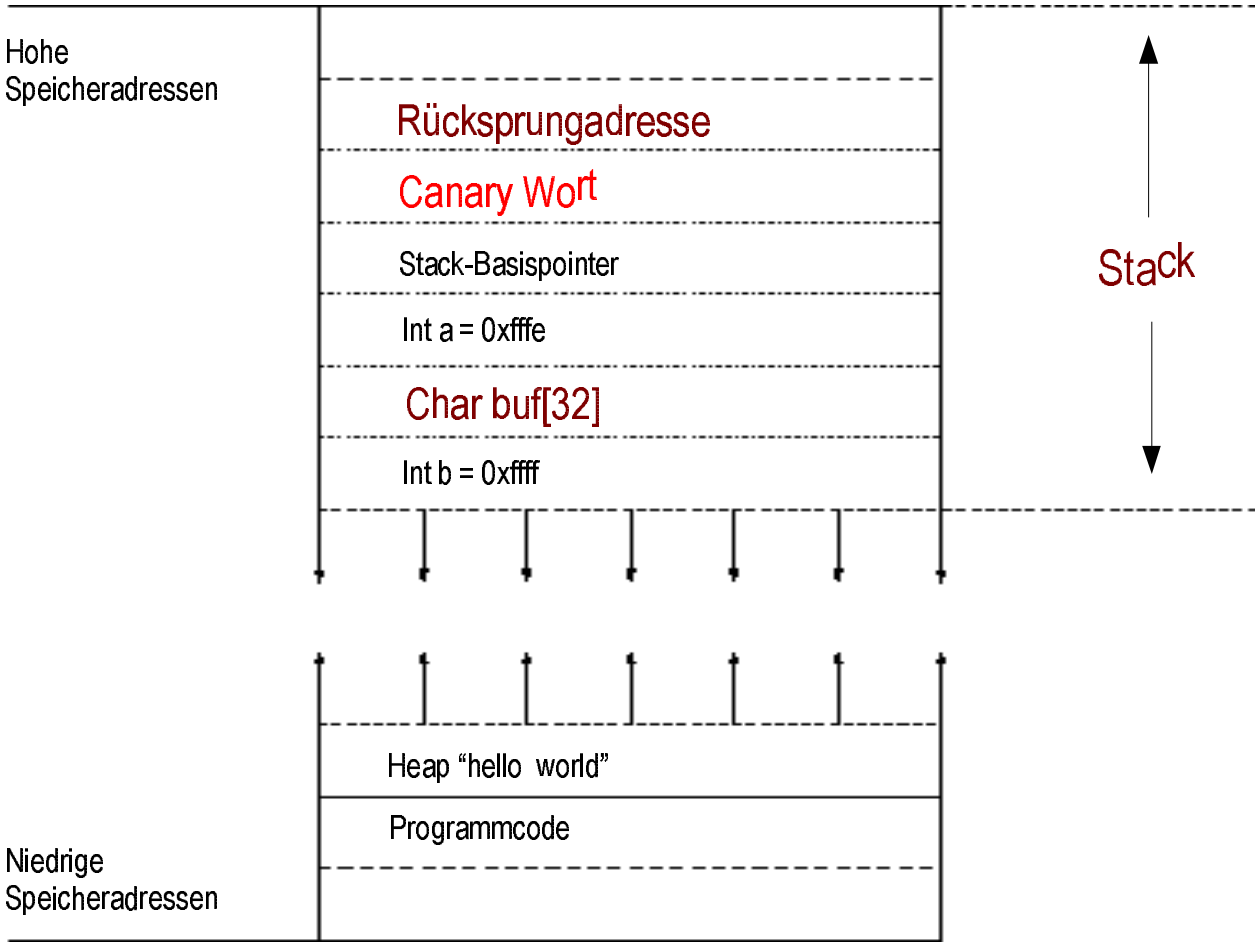
Präventive Schutzmöglichkeiten

Überschreibungsschutz von Zeigern

- Stackguard:
  - Einfügen eines *Canary* Wortes direkt vor Rücksprungadresse im Speicher
    - Annahme: Angreifer kann *Canary* nicht erraten
    - Exploit überschreibt *Canary* Wort beim Buffer-Overflow
- Zusätzlich: Schutz vor dem Ausnutzen von Format-String Schwachstellen notwendig
  - FormatGuard: glibc Patch

Präventive Schutzmöglichkeiten

Überschreibungsschutz von Zeigern



Speicherbelegung des Testprogramms mit StackGuard

DFN CERT

Präventive Schutzmöglichkeiten
Überschreibungsschutz von Zeigern

- Ansatz:
  - Libsafe und Libverify für Linux
- Vorgehensweise:
  - Verwendung von dynamischen Bibliotheken zur Absicherung der Rücksprungadresse
  - Libsafe: Ersetzen der unsicheren libc Funktionen durch sichere Funktionen (z.B. `strcpy`)
  - Libverify: Einfügen von Code zum Schutz der Rücksprungadresse während der Initialisierung des Prozesses (*binary rewriting*)

Präventive Schutzmöglichkeiten

Überschreibungsschutz von Zeigern

- Grenzen der Ansätze:
  - Zustandsänderungen auf dem Stack weiterhin möglich (Änderungen von lokalen Variablen und Zeigern)
  - Kein Verhindern von Denial of Service Angriffen
  - Kein Schutz vor Buffer-Overflows im Heapsegment
  - Fehler in Bibliotheken können weiterhin ausgenutzt werden

Zusammenfassung

- Präventiver Schutz vor dem Ausnutzen von Schwachstellen wichtig
- Ansätze zum Schutz vor dem Ausführen von Code in Speichersegmenten und vor dem Überschreiben von Zeigern empfehlenswert, weil
  - Keine Änderung des Quelltextes
  - Einfach einzusetzen
  - Keine bzw. geringe Performanzeinbußen
  - Aber Restrisiken vorhanden

Ausblick

- Exploits werden aufwendiger (z.B. Klasse der “return-into-libc” Exploits, sshd Exploit)
- Beiträge, wie Schwachstellen unter erschwerten Bedingungen (StackGuard) ausgenutzt werden können
- Verschleiern der Exploitprogramme (z.B. der sshd Exploit `x2` ist durch `burneye` verschluesselt worden)

⇒Forschungsthema: Sicherheit der Ansätze

⇒Forschungsthema: Verbesserungen der Ansätze

```
Segmentation fault
process terminated
core dumped
#
```


DFN CERT