Karlsruher Institut für Technologie

Einführung

Binary Exploitation 2

KITCTF

```
1 char sc[] = "\x6a\x0b"   // push byte +0xb
2 "\x58"                   // pop eax
3 "\x99"                   // cdq
4 "\x52"                   // push edx
5 "\x68\x2f\x2f\x73\x68"   // push dword 0x68732f2f
6 "\x68\x2f\x62\x69\x6e"   // push dword 0x6e69922f
7 "\x89\xe3"               // mov ebx, esp
8 "\x31\xc9"               // xor ecx, ecx
9 "\xcd\x80";              // int 0x80
```

Wiederholung: Canaries und ASLR

- ASLR
  - randomisiert Adressen von Heap, Stack und mapped memory
  - erschwert z.B. Return-Oriented-Programming

Wiederholung: Canaries und ASLR

- ASLR
  - randomisiert Adressen von Heap, Stack und mapped memory
  - erschwert z.B. Return-Oriented-Programming

- Canaries
  - Zufallswert vor Return-Adresse
  - Wird auf Veränderung überprüft bevor eine Funktion verlassen wird
  - Wird beim Programmstart festgelegt

printf

- Erstes Argument ist Vorlage
- Weitere Argumente werden formatiert und Vorlage eingebettet

```
printf("%d\n", 42);          // Ausgabe: 42
printf("%x\n", 42);          // Ausgabe: 2a
printf("%.2f\n", 3.14159);   // Ausgabe: 3.14
printf("%3$d %1$d\n", 0, 1, 2); // Ausgabe: 2 0
```

Format String Bugs

```
int main(int argc, char **argv)
{
   printf("%x\n");
}
```

- Was passiert hier?

Format String Bugs

```
int main(int argc, char **argv)
{
   printf("%x\n");
}
```

- Was passiert hier?
- → printf liest von dort, wo das nächste Argument stehen sollte
- Falls man das erste Argument kontrollieren kann, kann man beliebig vom Stack lesen

Format String Bugs

```
int main(int argc, char **argv)
{
   printf("%x\n");
}
```

- Was passiert hier?
- → printf liest von dort, wo das nächste Argument stehen sollte
- Falls man das erste Argument kontrollieren kann, kann man beliebig vom Stack lesen

- Was lesen?

Beispiel

```c
#include <stdio.h>

#define BUFSIZE 100

void vuln()
{
   char buf[BUFSIZE];
   fgets(buf, BUFSIZE, stdin);

   printf(buf);
}

int main(int argc, char **argv)
{
   vuln();
}
```

Schreiben

- printf kann auch in Variablen schreiben
- Formatparameter `%n` schreibt die Anzahl ausgegebener Zeichen in die Variable (4-Byte Wert)
- `%hn` schreibt einen 2-Byte Wert
- `%hhn` schreibt einen 1-Byte Wert
- `%<N>x` padded Ausgabe auf N Zeichen

Beliebige Adressen

- Bisher: Nur Stack kann gelesen und geschrieben werden
- Ziel: Zugriff auf beliebige Speicherbereiche
- $\rightarrow$ Zieladresse muss auf dem Stack platziert werden
- Ablegen der Adresse im Format-String

Beliebige Adressen

- Bisher: Nur Stack kann gelesen und geschrieben werden
- Ziel: Zugriff auf beliebige Speicherbereiche
- $\rightarrow$ Zieladresse muss auf dem Stack platziert werden
- Ablegen der Adresse im Format-String

- Was kann man überschreiben?

Ergänzung: Wie funktioniert ASLR?

- Problem: Wohin springen, wenn Adressen randomisiert werden?

Ergänzung: Wie funktioniert ASLR?

- Problem: Wohin springen, wenn Adressen randomisiert werden?
- Lösung: Dynamischer Linker findet die richtige Adresse zur Laufzeit

Ergänzung: Wie funktioniert ASLR?

- Problem: Wohin springen, wenn Adressen randomisiert werden?
- Lösung: Dynamischer Linker findet die richtige Adresse zur Laufzeit
- Lazy Binding:
  - Linker bei jedem Aufruf wäre zu langsam
  - Beim ersten Aufruf der Funktion findet Linker Adresse und schreibt sie in Global Offset Table (GOT)
  - Anfangs steht in der GOT die Adresse vom Linker-Aufruf
  - Konsequenz: GOT ist schreibbar

Demo

Karlsruher Institut für Technologie

Demo

Integer Bugs

- Overflow, Underflow
  - 2147483647 + 1 == -2147483648
  - -2147483648 - 1 == 2147483647
- Comparison Bugs
  - Werte werden vor Vergleich auf gleichen Datentyp gecastet
  - Nicht immer intuitives Verhalten
- Größe von Datentyp ist plattformabhängig, häufig:
  - `char`: 8 Bit
  - `short`: 16 Bit
  - `int`: 32 Bit
  - `long`: 32 Bit oder 64 Bit
  - `long long`: 64 Bit

Casting Regeln

- `char` und `short` werden immer zu `int` gecastet
- Ist einer von beiden Werten `unsigned`, wird der andere auch zu `unsigned` gecastet
- Bei `int` und `long long` wird zu `long long` gecastet

```
(unsigned int) 1 >= (int) -1 // false
(uint8_t) 1 >= (int8_t) -1 // true
```

Weitere Vulnerabilities

- Timing Attacks
  - z.B.: Passwort wird Zeichen für Zeichen überprüft
- Race Conditions
- .GOT, .PLT overwrites
- Heap Bugs
  - Use-after-free
  - Funktionsweise von malloc/free
  - https://github.com/shellphish/how2heap

Tools / Ressourcen

- gdb
  - peda
  - gef
- python
  - pwntools (`https://docs.pwntools.com/en/stable/`)
  - `https://github.com/saelo/ctfcode/blob/master/pwn.py`
- ltrace / strace
- nc
- checksec

- Buch: 'Hacking - the Art of Exploitation'
- `http://phrack.org/issues/49/14.html`
- `http://www.myne-us.com/2010/08/from-0x90-to-0x4c454554-journey-into.html`
- `http://liveoverflow.com/binary_hacking/`

Aufgaben

- https://picoctf.com/
- http://overthewire.org/wargames/leviathan/
- http://overthewire.org/wargames/narnia/
- http://overthewire.org/wargames/behemoth/
- https://exploit-exercises.com/protostar/
- https://microcorruption.com/
- https://pwnable.kr/
- https://pwnable.xyz/
- http://smashthestack.org/wargames.html