Softwaresicherheit

Tobias Maier & Jens Willmer

Vorlesung {IT|Daten}sicherheit

Inhalt

Guidelines zur sicheren Entwicklung
- Grundsätze
- Security Development Lifecycle

Sicherheitslücken in Software
- Bufferoverflow
- Format String Attack

Grundsätze

Secure by design
- Sicherheit beginnt beim Entwurf
- Sicherheitslücken frühzeitig erkennen
- Nachbesserung aufwändiger und teurer

Grundsätze

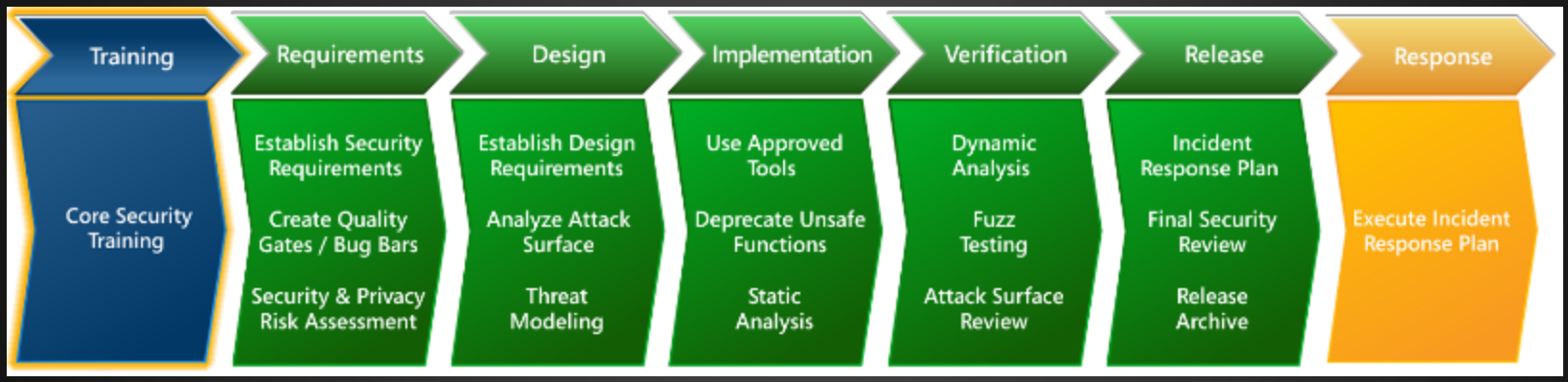
Secure by default
- Annahme: Es gibt immer Sicherheitslücken.
- Prinzip der geringsten Rechte
- Ungenutze Features deaktivieren
- "Sichere" Standard-Einstellungen wählen

Grundsätze

Secure in deployment
- Umfassende Dokumentation der Software
- Sicherheitshinweise an Administratoren
- Offener Umgang mit Sicherheitslücken

Security Development Lifecycle

SDL wird 2004 von Microsoft veröffentlicht



Security Development Lifecycle

Training
- Schulung der Entwickler
- Sensibilisierung für Sicherheitslücken

Security Development Lifecycle

Requirements
- Aufstellen Mindestsicherheitsanforderungen
- Definition schutzbedürftiger Daten
- Bug Bar / Null-Fehler-Tor
- Detaillierte Risikobewertungen

Security Development Lifecycle

Design
- "Secure by design"
- Umsetzung der Anforderungen
- Angriffsflächen reduzieren
- "Threat modelling": Schutzziele definieren

Security Development Lifecycle

Implementation
- Verwendung sicherer Funktionen
- Regelmäßige Reviews
- Evtl. Unterstützung durch Analyse-Tools

Security Development Lifecycle

Verification
- Test des laufenden Programms
- Analyse bezgl. Speicher, Rechte etc.
- "Fuzz Testing"
- Bezug zu früheren Phasen
- Kontrolle, ob neue Lücken entstanden sind

Security Development Lifecycle

Release
- Bennenung von Ansprechpartnern
- Finales Sicherheitsreview
- Archivierung von Erkenntnissen

Security Development Lifecycle

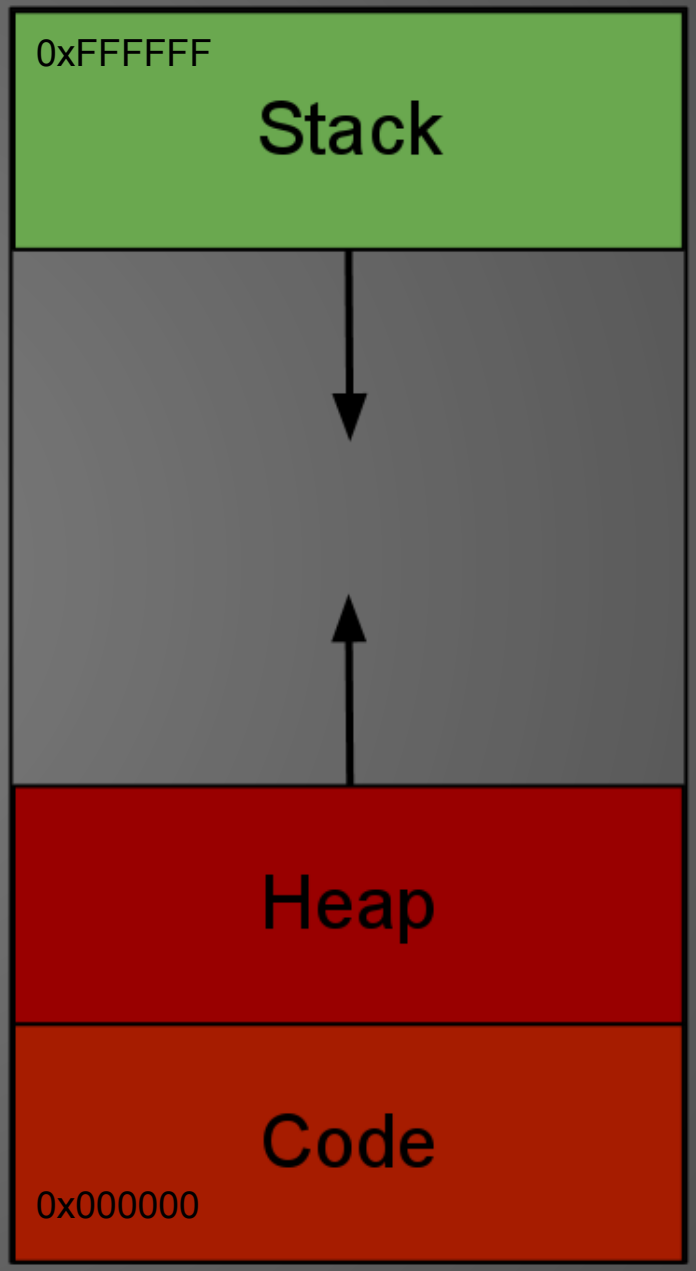
Response
- Betreuung während Betriebsphase
- Reaktion auf spätere Sicherheitslücken
- Nachbesserung per Updates

Buffer-Overflow

Grundlagen
- Was ist ein Buffer-Overflow?
- Wodurch entsteht er?
- Welche Sprache ist anfällig gegen Buffer-Overflows?

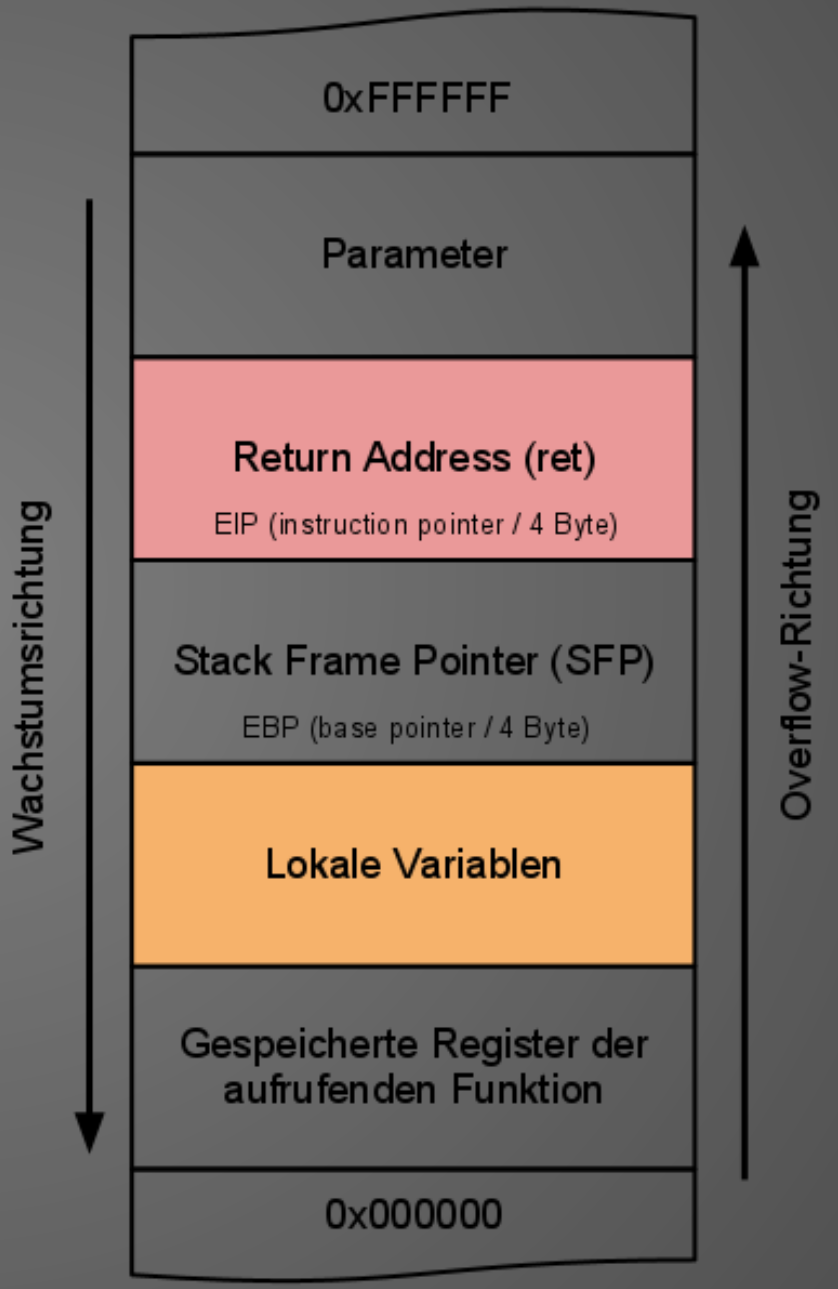
Buffer-Overflow

Speicherverwaltung
- Memory Management Unit
- Adressraum:
  - Stack
  - Heap
  - Codebereich



Buffer-Overflow

Speicherverwaltung
- Stack-Frame



Buffer-Overflow

Anwendungsbeispiel 1

Überschreiben einer Variablen

```c
void main()
{
    // integer allocates 4 Bytes in memory
    int authentication = 0;

    // 12 bytes in memory because we have 4 byte addressblocks!
    char cUsername[10];
    char cPassword[10];

    printf(" Enter Username: ");
    gets(cUsername);

    printf(" Enter Password: ");
    gets(cPassword);

    if(strcmp(cUsername, "admin") == 0 && strcmp(cPassword, "adminpass") == 0)
    {
      authentication = 1;
    }

    // checks if variable have value!
    if(authentication)
    {
        printf(" Access granted\n");
    }
    else
    {
        printf(" Wrong username and password\n");
    }

    return 0;
}
```

Buffer-Overflow

Victim

Anwendungsbeispiel 2

Verändern einer Rücksprungadresse

```
void PrivateFunction(void);
const char * const RepeatFunction(void);

int _tmain()
{
        printf("\n  You wrote: %s\n", RepeatFunction());
        return 0;
}

const char * const RepeatFunction(void)
{
        char buf[12];

        printf("\n  Write something: ");
        gets(buf);

        return strdup(buf);
}

void PrivateFunction(void)
{
        printf("\n\n  --> Output of the private function!\n");
        exit(0);
}
```

Buffer-Overflow

Exploit

Anwendungsbeispiel 2

Verändern einer Rücksprungadresse

```
void main()
{
    // fill buffer and base pointer
    // 12 Byte (buffer) + 4 Byte (EBP: base pointer)
    printf("0101010101AAAA");

    // new return address (EIP: instrction pointer)
    unsigned EIP = 0x01021400;

    // overwrite EIP
    fwrite(&EIP, 1, 4, stdout);

    return 0;
}
```

Buffer-Overflow

Gegenmaßnahmen

- Daten aus externen Quellen immer Validieren
- Funktionen ohne interne Validierung meiden
- Suchfunktion von Compiler nutzen
- Mit externen Programmen Anfälligkeiten testen
    - Fuzzer

Format String Attack

Problem bei ungefilterter Benutzereingabe
Beispiel in C:

```
printf(input)
Eingabe:  %s%s%s%s%s%s
-> würde versuchen, Stack auszugeben
-> führt evtl. zum Programmabsturz
Eingabe:  %s%n%s%n%s%n%s%n%s%n%s
-> überschreibt Speicherbereiche
```

Besser: `printf("%s", buffer)`