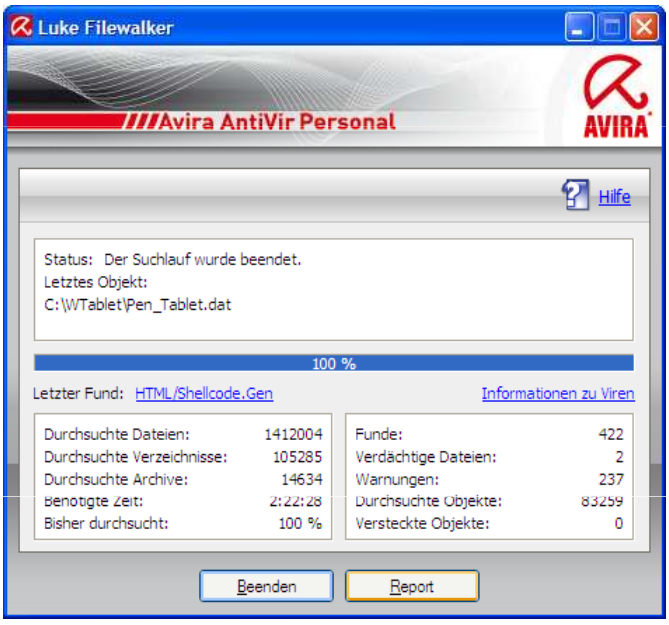
Vorlesung Systemsicherheit Wintersemester 2010/2011

Exkurs zum Thema Softwaresicherheit

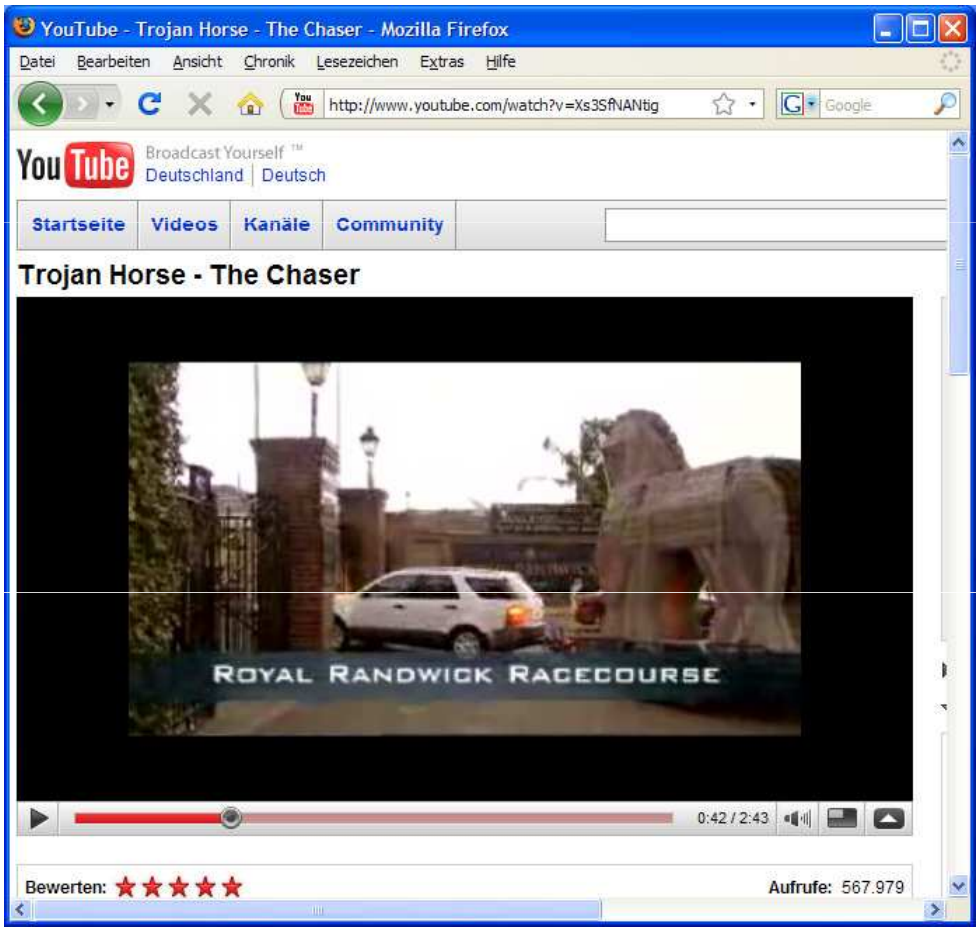
Felix Freiling

20.1.2011

Scan meines Rechners am 12.11.2009



Neues vom Trojanischen Pferd



Motivation: Privilegeskalation

- Typischer Angriffsweg:
  - Verschaffen von lokalem User-Zugriff durch brechen eines schwachen Passwortes
  - Ausnutzen einer lokalen Schwachstelle des Systems zur Erlangung von root-Rechten
  - Installation einer Hintertür und Verwischen der Spuren
- Jetzt: Ausnutzen lokaler Schwachstellen
  - buffer overflows, etc.

- Merke: Prüfe Deine Eingabedaten!

Übersicht

- *Race Conditions*
- [Buffer Overflows]
  - Heap Overflows
  - Stack Overflows
- Integer Overflows
- Format-String-Angriffe
- (SQL) Injection
- [Cross Site Scripting]

- Abschließend: kurzer Exkurs zu Return-oriented Programming

Kritische Abschnitte

- Multitasking Betriebssystem: nebenläufige Prozesse, die sich den Prozessor (und andere Ressourcen) teilen
  - Unterbrechungen (Interrupts) können laufenden Prozess jederzeit unterbrechen
    - Beispiele: Taste wird gedrückt, Festplatte hat Daten gelesen, Zeitscheibe ist abgelaufen
  - Interrupt-Handler wird ausgeführt (wie Unterprogramm-Aufruf)
- Codesequenz, die ununterbrochen ausgeführt werden muss = kritischer Abschnitt
- Beispiel: Einfügen eines neuen Prozesses in die ready-Warteschlange im Kernel

Implementierung kritischer Abschnitte

- Auf Prozessorebene: Unterbrechungssperren
  - privilegierte Maschinenbefehle
  - Unterbrechungen sollten nicht zu lange ausgestellt bleiben
    - In der Zwischenzeit auflaufende Interrupts könnten verloren gehen
- Auf Prozess-Ebene: Synchronisationskonzepte
  - busy waiting (TAS-Schleife), Sperren (locks), Semaphore, Monitore (Java `synchronized`)
  - Realisierung eines Synchronisationsprotokolls
  - Annahme: jeder beteiligte Prozess führt das Protokoll aus, bevor er in den kritischen Abschnitt eintritt
  - Intelligentes Warten: keine Blockierung anderer Prozesse (die nicht die gemeinsame Ressource brauchen)
- Kritische Abschnitte treten auch in "normalen Code" auf...

Beispiel

- Zugriff auf eine Datei

```
if (access("filename", R_OK) == 0)
{ fp = fopen("filename", "r"); }
```

- Rechteüberprüfung und Zugriff auf Datei sind nicht wechselseitig ausgeschlossen!
  - TOC: time of check
  - TOU: time of use

Angriffsszenario

- Einfaches Programm zum anlegen eines neuen Users in Unix
  - liest Kennung, Passwort, etc.
  - Prüft Berechtigungen
  - fügt neuen Eintrag an `/etc/passwd`, `/etc/shadow` an
- Programm muss SUID root sein
- Programm `race_vp.c`
  - hat zwei Parameter
  - erster Parameter: Dateiname
  - zweiter Parameter: String
  - nach diversen Checks wird String in Datei geschrieben
- Können Race Condition ausnutzen, um root Zugang zu erhalten

Angriffsplan

- Angriffsskript `racer.pl`
  - mittels `nice` wird die Wahrscheinlichkeit einer Unterbrechung erhöht
  - anschliessend wird Datei `./dummy` umgebogen auf `/etc/shadow`
- Fragen:
  - warum `touch dummy` ?
  - wieso bewirkt der gegebene String root-Zugang?
  - was macht `ln -f -s /etc/shadow .dummy` ?

Analyse

- Race Conditions sind besonders gefährlich, weil man sie kaum automatisiert entdecken kann
  - sind auch nicht an eine spezielle Programmiersprache gebunden (wie stack overflows)
- Voraussetzungen für eine Race Condition:
  - Mehrere verschiedene Threads oder Prozesse, die eine gemeinsame Ressource verwenden:
    - Gemeinsame Variablen
    - Gemeinsame Datei oder gemeinsam verwendeted Verzeichnis
    - Windows Registry
    - Datenbank
- Übliche Orte, wo Race Conditions auftreten:
  - Signal Handler
  - Verwendung temporärer Dateien
  - Funktionen mit Seiteneffekten (non-reentrant) in Programmen mit mehreren Threads

Abhilfe

- In Programmen mit mehreren Threads:
  - Korrekte Synchronisation herbeiführen
    - verwenden von Semaphoren, Locks, Monitoren (Java synchronized)
  - Bei fehlender Synchronisation:
    - Seiteneffektfreies Programmieren
    - Funktionen müssen sich selbst aufrufen können (*reentrant code*)
- In Signal Handlern:
  - Nur *reentrant code* in Signal Handlern benutzen
  - Besser: Signals in Signal Handlern verbieten
- Falls Dateien/Verzeichnisse das Problem sind:
  - Dateien erzeugen in Bereichen, wo normale Benutzer keine Schreibberechtigung haben
  - Nicht eindeutige Namen sind notwendig sondern unvorhersagbare Namen
    - Echte Zufallsmuster als Dateinamen verwenden

Exkurs: der ptrace Bug

- 2003 wurde bekannt, dass es eine Schwachstelle im aktuellen Linux-Kernel gibt. Voraussetzungen:
  - Kernel arbeitet mit nachladbaren Modulen
  - `/proc/sys/kernel/modprobe` zeigt auf eine ausführbare Datei
  - Aufrufe von `ptrace()` werden nicht geblockt
- `man ptrace`:
  - The ptrace system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.
- Problem: Beim Nachladen von Kernelmodulen (`modprobe`) startet der Kernel einen Prozess und setzt anschliessend die EUID und EGID auf 0
  - Vor dem Setzen von EUID und EGID kann man den Prozess mit ptrace manipulieren

Übersicht

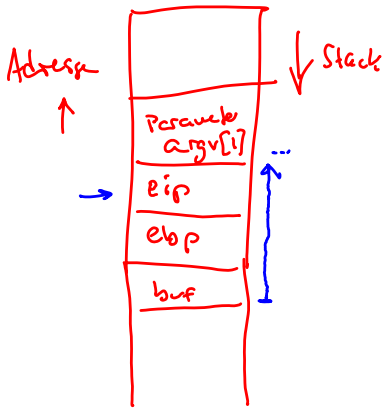
- Race Conditions
- *[Buffer Overflows]*
  - Heap Overflows
  - Stack Overflows
- Integer Overflows
- Format-String-Angriffe
- (SQL) Injection
- [Cross Site Scripting]

Weitere schlechte Beispiele...

```
#include <stdio.h>

void DontDoThis(char* input) {
  char buf[16];
  strcpy(buf, input);
  printf("%s\n", buf);
}

int main(int argc, char* argv[]) {
  DontDoThis(argv[1]);
  return 0;
}
```



Bemerkungen

- Klassische Stack Overflow Schwachstelle:
  - Eingabe wird mittels `strcpy` in einen Buffer geschrieben
  - Buffer ist lokale Variable
  - Kopieren geschieht innerhalb einer Unterprozedur
  - Eingabeargument wird nicht überprüft
- Aufgabe für Zuhause:
  - Probieren Sie die eben vorgestellte Stack Smashing Technik auf diesem Beispiel aus

Berühmtes Beispiel

- C-Funktion : `char* gets(char* str)`
  - Siehe: `man gets`
  - Liest von der Standardeingabe Zeichen und kopiert sie in den String `str`. Liefert einen Zeiger auf `str` zurück.
  - Kopiert so lange, bis `\n` (newline) oder `\0` (Null-Zeichen) gelesen wird
- Code mit Schwachstelle:

```
char buf[20];
gets(buf);
```

- Original-Schwachstelle aus `fingerd`
  - Ausgenutzt im Morris Worm (siehe später)
- Was kann hier schiefgehen?

Wie findet man Buffer Overruns?

- Aufpassen auf:
  - Input, egal ob von der Kommandozeile, vom Netzwerk oder aus einer Datei
  - Weitergabe von input (siehe oben) an innere Programm- und Datenstrukturen
  - Verwendung "unsicherer" (d.h. unbeschränkter) String-Kopieroperationen (`strcpy`, `strcat`, `sprintf`, ...)
  - Arithmetik der Berechnung von Puffergrößen oder verbleibenden Puffergrößen
- Unsichere String-Funktionen kann man systematisch finden
  - Zum Beispiel durch den Compiler: `#undef strcpy`
  - Manchmal implementieren Applikationen die C-Library intern neu
- Immer die Frage: Was kontrolliert der Angreifer?

Wie testet man auf Buffer Overruns?

- Fuzz Testing (*Fuzzing*)
  - Bomardieren der Applikation mit pseudo-zufälligen Eingabestrings zunehmender Länge
  - Falls die Applikation abstürzt, hat man es mit hoher Wahrscheinlichkeit mit einem Buffer Overrun zu tun
- Beim Fuzz Testing sollte man in der Applikationsumgebung testen (keine Debug-Ausgaben, kein *debug build* etc.)
  - Den fertigen, optimierten Code testen
- Falls String-Eingabe *und* Länge vom Benutzer kommen:
  - Eingegebene Länge verschieden von der String-Länge testen
  - Hier auch auf Integer Overflows achten

Wie schützt man sich vor Buffer Overruns?

- Unsichere C-Funktionen (`strcpy`, `strcat`, `sprintf`, etc.) durch "sichere" Versionen ersetzen (`strncpy`, `strncat`, `snprintf`, etc.)
  - Hier auf Arithmetik zur Berechnung von n achten
- Prüfen der Abbruchbedingungen in Schleifen
- C-Strings durch C++-Strings ersetzen
  - Kann viel Neuprogrammierung existierender Software erfordern
- Analysewerkzeuge benutzen:
  - Viele Compiler prüfen auf Basis von Heuristiken auf mögliche Buffer Overruns

- Beim Programmieren besser aufpassen!

Zusätzliche Abwehrmaßnahmen

- Stack-Schutz (z.B. *Stackguard*):
  - Ein spezieller Schutzstring (*canary*) wird zusammen mit der Rücksprungadresse auf den Stack gelegt
  - Vor dem Rücksprung prüft automatisch eincompilierter Code, ob der *canary* noch lebt
  - In modernen Compilern per Option zuschaltbar
- Nicht-ausführbarer Stack
  - Hardware kann den Stack als *non-executable* markieren
  - Jeder Versuch, shellcode auf dem Stack auszuführen, schlägt fehl
  - Unterstützung hängt sehr von Hardware und Betriebssystem ab

Exkurs: Der Morris-Wurm

- Im November 1988 legte ein Computerprogramm große Teile des Internet lahm
  - Wurm = Schadsoftware, die sich selbständig von Rechner zu Rechner verbreitet (unter Ausnutzung von Schwachstellen)
- Zustand der Internet-Sicherheit 1988 sehr mangelhaft
  - schwache Passwörter
  - laxe Sicherheitsrichtlinien
    - rhosts
    - bekannte Schwachstellen in finger, sendmail etc.
- Netzgemeinde konterte erfolgreich
  - MIT und Berkeley waren dauerhaft "online" und hatten nur so die Chance, den Wurm zu fassen
  - Logging-Informationen waren sehr wichtig für die Rekonstruktion

Der Morris-Wurm: Chonologie

- Am Abend des 2. November 1988:
  - 6:00 PM At about this time the Worm is launched.
  - 8:49 PM The Worm infects a VAX 8600 at the University of Utah (cs.utah.edu)
  - 9:09 PM The Worm initiates the first of its attacks to infect other computers from the infected VAX
  - 9:21 PM The load average on the system reaches 5
    - (normal load average at this time is 1. Any load average higher than 5 causes delays in data processing.)
  - 9:41 PM The load average reaches 7
  - 10:01 PM The load average reaches 16

Chronologie (Forts.)

  - 10:06 PM At this point there are so many worms infecting the system that no new processes can be started. No users can use the system anymore.
  - 10:20 PM The system administrator kills off the worms
  - 10:41 PM The system is reinfected and the load average reaches 27
  - 10:49 PM The system administrator shuts down the system. The system is subsequently restarted
  - 11:21 PM Reinfestation causes the load average to reach 37.
- [In der gleichen Art und Weise werden weltweit 6000 weitere Rechner befallen]

Überblick

- Der Wurm besteht aus
  - 99 Zeilen C Programm (bootstrap)
  - eine große Objektdatei (entweder für VAX oder für Sun-3), die aus C-Quellen generiert wurde
- Angriffsfunktionalität:
  - wie finde ich neue Rechner?
  - wie komme ich auf die Rechner drauf?
  - wie führe ich den Wurm auf dem Rechner aus?
- Verteidigung:
  - wie verstecke ich mich?
  - wie entziehe ich mich einer Analyse?

Finden neuer Rechner

- Der Wurm suchte sich neue Rechneradressen in
  - `/etc/hosts.equiv`
  - `/etc/rhosts`
  - `.forward` und `.rhosts`
  - Routinginformationen aus der Ausgabe von `netstat`
  - zufällig gewählte Adressen im lokalen Netz

Eindringen in neue Rechner

- Der Wurm konnte verschiedene Schwachstellen ausnutzen:
  - `finger`: *buffer overflow* mit `gets()`
  - *trap door* in falsch konfiguriertem `sendmail`
- Der Wurm versuchte, schwache Passwörter zu knacken:
  - eingebautes Lexikon
  - nach erfolgreichem Raten: Einloggen mit `rexec`
  - `rsh` benutzen bei Einträgen in `rhosts`

Propagierung des Wurms

- Ziel: Kommandointerpreter (shell) auf entferntem Rechner
  - herunterladen, compilieren und ausführen eines 99-zeiligen C-Programms (bootstrap)
  - der bootstrap code öffnet eine eigene Netzwerkverbindung zum vorherigen Rechner
    - holt sich die benötigten Dateien direkt herüber
    - startet eine neue Angriffsrunde

Verteidigungsmethoden

- Der Wurm
  - änderte seinen Namen (zu `sh`)
  - änderte dauernd seine PID (mittels `fork`)
  - versuchte möglichst keine Spuren auf der Platte zu hinterlassen
    - schrieb die verwendeten Dateien in den Hauptspeicher und löschte sie sofort von der Festplatte
  - schaltete die Erzeugung von *core* Dateien ab
  - notwendige Dateien auf der Festplatte wurden auf sehr einfache Art verschlüsselt
    - XOR mit 0x81 bzw. 0x80
  - authentifizierte sich auf eine einfache Art vor der Propagierung (Austausch von "Magic Numbers")

Was der Wurm nicht tat

- der Wurm änderte oder löschte keine Dateien
- der Wurm verschickte keine geknackten Passworte
- der Wurm versuchte nicht root-Rechte zu erlangen bzw. root-Rechte speziell zu nutzen
- der Wurm griff keine Maschinen an ausser VAX und Sun 3 mit BSD Unix
- der Wurm propagierte sich nicht über Disketten
- der Wurm verursachte keine physischen Schäden

C-Pseudocode von `doit()`

```
doit()
{
   seed the random number generator with the time
   attack hosts: gateways, local nets, remote nets
   checkother();
   send message();
   for (;;)
   {
      cracksome();
      other_sleep(30);
      cracksome();
      change our process ID
      attack hosts: gateways, known hosts, remote nets, local nets
      other_sleep(120);
      if (12 hours have passed)
         reset hosts table
      if (pleasequit && nextw > 10)
        exit(0);
    }
}
```


Details

- `checkother()` testet, ob ein anderer Wurm bereits läuft
  - versucht eine TCP Verbindung zu Port 23357 aufzumachen (Wurm-Server)
    - falls da einer ist, wird ausgewürfelt, wer überlebt
    - setzen von `pleasequit`
- `send_message()` in jedem 15. Wurm sendete diese Funktion offenbar einen 1-byte UDP-Paket an `ernie.berkeley.edu`
  - Verwendung unklar, möglicherweise ein Ablenkungsmanöver
  - hatte einen bug: wollte UDP-Datagram über TCP-Socket senden

Details

- `cracksome()` versucht lokale Passwörter zu knacken
  - liest die lokale Passwortdatei
    - testet erst Null-Passwort, dann login-Namen, dann einfache Variationen
    - anschliessend Liste von 432 eingebauten Begriffen
    - anschliessend `/usr/dict/words`
  - versucht mit geknackten Passwörtern Accounts mit demselben Namen auf fremden Rechnern zu entern
- `other_sleep()` enthält die Server-Komponente
  - macht intern ein `select()` und wartet auf eine eingehende Verbindung auf TCP Port 23357
  - bei einer "authentifizierten" Verbindung vollzieht der Wurm das server-seitige Wurmprotokoll

Folgen

- Robert Morris wurde 1990 zu 3 Jahren Haft auf Bewährung verurteilt, plus 400 Stunden gemeinnützige Arbeit und $10.050 Strafe
  - er ist jetzt Professor am MIT
    - `http://www.pdos.lcs.mit.edu/~rtm/`
  - er hat (nach meinen Informationen) nie öffentlich über den Vorfall berichtet

Quellen zum Morris-Wurm

- Peter J. Denning: Computers under Attack. Intruders, Worms and Viruses. Addison-Wesley, 1990.
- Charles Schmidt und Tom Darby: The Morris Internet Worm. `http://www.snowplow.org/tom/worm/worm.html`
- Quellcode des Wurms plus viele Untersuchungsberichte und sonstige Informationen: `http://ftp.cerias.purdue.edu/pub/doc/morris_worm/`

Übersicht

- Race Conditions
- *Buffer Overflows*
  - Heap Overflows
  - Exkurs: Stackaufbau bei Intel x86
  - Exkurs: Shellcode
  - *Stack Overflows*
- *Integer Overflows*
- Format-String-Angriffe
- (SQL) Injection
- Cross Site Scripting

Integer Overflows

- Zahlen im Computer sind (fast) immer in der Größe beschränkt
  - Programmiersprachen und Compiler bieten typischerweise Zahlen mit 8 Bit, 16 Bit, 32 Bit und neuerdings auch 64 Bit an
- Wegen der beschränkten Größe kommt es im Computer immer wieder zu anderen Rechenergebnissen als auf dem Papier
- Jede Programmiersprache ist betroffen
  - Datenformate und Casts haben ihre eigenen Regeln
  - Man muss viele Regeln kennen, um hier keine Fehler zu machen
- Betrachten hier ein paar Beispiele, um ein Gefühl für das Problem zu bekommen
  - C/C++
  - Java

Datentypen in C/C++

- (`signed`/`unsigned`) `char`
  - Laut Standard "minimale Einheit, um ein Zeichen zu speichern"
  - also minimal 8 Bit, mit oder ohne Vorzeichen
  - Bereich mindestens zwischen 0 und 255 bzw. -127 und 127
  - Ohne explizites Schlüsselwort entscheidet der Compiler ob ein einfaches `char` signed oder unsigned ist
- (`signed`/`unsigned`) `short` (`int`),
  - minimal 16 Bit, mit oder ohne Vorzeichen
  - Bereich mindestens zwischen -32767 und 32767
- (`signed`/`unsigned`) `long` (`int`)
  - minimal 32 Bit, mit oder ohne Vorzeichen
  - Bereich mindestens zwischen -2147483647 und 2147483647
- (`signed`/`unsigned`) `int`
  - Laut Standard "mindestens ein `short`"
  - Garantierter Wertebereich -32767 bis 32767 bzw. 0 und 65535 (unsigned)
  - Auf vielen Compilern ist ein `int` ein `long`
- Neuerdings in C standardisiert: `long long` (64 Bit)

Regeln des Typecast

```
const long MAX_LEN = 0x7fff;
short len = strlen(input);
if (len < MAX_LEN) // do something
```

- Zuweisung len = strlen(input) enthält impliziten Cast
  - len ist short, strlen ist size_t (normalerweise größter unsigned int, den die Maschine hergibt)
  - Hier: Downcast
    - Zugewiesener Wert wird degradiert zu short
    - Passiert durch "Abschneiden" von Bytes
    - Normalerweise Compiler-Warnung
    - Informationsverlust möglich
- Vergleich `len < MAX_LEN` enthält ebenfalls einen Cast
  - Beide Seiten werden zum jeweils "höchsten" Typ befördert (up-casting, promotion)
    - short len wird zu long
    - Upcast ist wohldefiniert sowohl für signed als auch unsigned
      - Bei signed wird das Vorzeichen mitgezogen

Regeln des Typecast (Forts.)

- Konvertierung von signed nach unsigned
  - Bitmuster wird beibehalten
  - Interpretation kann sich ändern
  - Beispiel:
    - `unsigned char c = (signed char) 0xff`
    - `0xff` bedeutet -1 als `signed char`
    - nach dem Cast hat es die Bedeutung 255
- Konvertierung von unsigned nach signed analog
  - Bitmuster wird beibehalten
  - Interpretation kann sich je nach Vorzeichen ändern
- Wenn sowohl Vorzeichen, also auch Bit-Größe gecastet werden muss:
  - erst Bitgröße upcasten
  - dann Vorzeichen uminterpretieren

- In der Header-Datei limits.h werden maximale Werte für die verschiedenen Datentypen definiert
  - Zum Beispiel: `INT_MAX`, `INT_MIN` etc.

Arithmetische Operationen

- Addition/Subtraktion
  - Problem des "wrap around"
    - unsigned char 255 + 1 = 0
    - signed char 127 + 1 = -128
- Multiplikation
  - Problem der zu großen Ergebnisse
  - Falls a*b > INT_MAX ist das Ergebnis falsch
  - Test der Bedingung im nächst größeren Datentyp
  - Alternative: Test ob b > INT_MAX/a
- Division
  - Division durch Null immer verboten
  - Unerwartet aber bei signed char: -128 / -1
    - Erwartet eigentlich: -(-128) = 128
    - Wrap around führt wieder zu -128

Vergleiche

- Oft gemachter Fehler: Test auf maximale Größe mit einem signed int

```
if (x < MAX_BUF) // do something
```

- Falls `x` signed ist, kann ein Angreifer `x` zum Überlauf bringen
  - `x` wird negativ, Test gelingt
- In solchen Fällen:
  - besser unsigned int benutzen
  - oder erst auf positiv testen und dann auf kleiner als geforderter Maximalwert

Java

- Java hat ähnliche Probleme mit Integer Overflows wie C
- Zitat aus der Java Language Specification:
  - The built-in integer operators do not indicate overflow or underflow in any way. The only numeric operators that can throw an exception (§ 11) are the integer divide operator / (§ 15.17.2) and the integer remainder operator % (§ 15.17.3), which throw an ArithmeticException if the right-hand operand is zero.
- Quelle:
  `http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html#9151`
- Vorteile in Java gegenüber C:
  - Alle Integers sind vorzeichenbehaftet
  - Einziger unsigned Typ ist char (16 Bit vorzeichenlos)

Beispiel

```
class Test {
  public static void main(String[] args) {
    int i = 1000000;
    System.out.println(i * i);
    long l = i;
    System.out.println(l * l);
    System.out.println(20296 / (l - i));
  }
}
```

- Gibt aus:

```
-727379968
1000000000000
```

- gefolgt von einer `ArithmeticException`

Vermeiden von Integer Overflows

- Aufpassen beim Programmieren:
  - Jede Art von Indexkalkulation
  - Jede Art von Berechnung einer Puffergröße
- C/C++ Programmierer sollten besonders aufpassen
- Aber auch Java-Programmierer können leicht fehlerhafte Logik einbauen

- Beim Testen:
  - Eingabestrings kritischer Größe testen
  - 127, 128, 255, 256 Bytes
  - 32K, 64K Bytes sowie jeweils ein paar Bytes mehr oder weniger

- Avoid "clever" code - make your checks for integer problems straightforward and easy to understand. (Howard et al., p. 40)

Übersicht

- Race Conditions
- Buffer Overflows
  - Heap Overflows
  - Exkurs: Stackaufbau bei Intel x86
  - Exkurs: Shellcode
  - Stack Overflows
- Integer Overflows
- *Format-String-Angriffe*
- (SQL) Injection
- Cross Site Scripting

Exkurs: `printf` mit Formatstrings

- Signatur von `printf`:

```
int printf(const char* format, ...);
```

- Der Formatstring `format` bestimmt wie die weiteren übergebenen Parameter behandelt werden
- Beispiele:
  - `printf("Zahl: %d", 35);`
  - gibt die Zahl 35 in dezimaler Darstellung aus
  - `printf("Zahl: %x", 35);`
  - Ausgabe der Zahl 35 in hexadezimaler Darstellung
  - `printf("%s %n", buf, &num_bytes);`
  - Gibt den String `buf` aus und schreibt in `num_bytes` die Anzahl der ausgegebenen Zeichen

- Der Formatstring kann beliebig komplex sein
  - Beispiel: `printf("%.20d", 35);`
  - gibt die Zahl 35 mit 20-stelliger Genauigkeit aus

Problemfälle

- Was ist, wenn man keine Zahl angibt?
  - `printf` nimmt implizit an, dass die auszugebenden Argumente auf dem Stack liegen
- Beispiele:
- `printf("%x");`
  - gibt das oberste 32-Bit-Wort auf dem Stack in hexadezimaler Schreibweise aus
- `printf("%x%x%x%x");`
  - gibt die obersten vier Worte von Stack aus
- `printf("%.20x");`
  - gibt das oberste Wort mit 20 Stellen Genauigkeit aus
- `printf("AAA%n");`
  - Schreibt in die Speicherstelle, auf die das oberste Stack-Element zeigt, die Anzahl der bisher ausgegebenen Bytes

Was kann passieren?

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  if (argc > 1)
    printf(argv[1]);

  return 0;
}
```

- Übergabe von `"Hello World"` ergibt
  `Hello World`
- Übergabe von `"%x %x"` ergibt (zum Beispiel)
  `12ffc0 4011e5`

Format-String-Angriff

- Plan: verwende %x zur Analyse des Stacks und %n zur Veränderung des Speichers
- Beispiel:
  - gebe geeignete Anzahl von Zeichen aus
  - lege Zeiger auf die Rücksprungadresse auf den Stack
  - überschreibe Rücksprungadresse mit gewünschter Zahl mittels %n

Beispiele

- Falsch:

```
printf(user_input);
```

- Richtig:

```
printf("%s", user_input);
```

- Oft gemachter Fehler bei Debug-Ausgaben:
  - Mittels `sprintf` eine Nachricht in einem Buffer `err_msg` vorbereiten
  - Dann:

```
fprintf(STDOUT, err_msg);
```

- Problem: Angreifer kann die Format-String-Zeichen `%x` etc. escapen
  - `fprintf` hat dann dieselben Probleme wie Beispiel von oben

Internationalisierung

- Häufig werden Meldungstexte einer Applikation in einer externen Datei abgelegt
  - Je nach verwendeter Sprache wird andere Datei verwendet
- Falls Angreifer Zugriff auf diese Datei hat, ist ebenfalls die Gefahr von Format String Angriffen gegeben

Vermeiden von Format String Schwachstellen

- Alle Aufrufe der `printf`-Familie sind verdächtig
  - Falls der Format-String nicht explizit angegeben wurde, Herkunft des Strings prüfen
- Beispiel:

```
fprintf(STDOUT, msg_format, arg1, arg2);
```

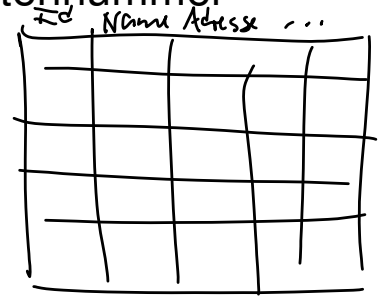
- Wo kommt `msg_format` her? Wer hat darüber Kontrolle?
- Viele Source Code Scanner erkennen mittlerweile Format-String-Probleme (RATS, flawfinder, pscan)
- Beim Testen: Streuen Sie `%x` oder `%n` in Ihre Eingabestrings ein
- Am besten alle `printf`-Aufrufe vermeiden
  - In C++ besser die Stream-Operatoren verwenden

Übersicht

- Race Conditions
- [Buffer Overflows]
  - Heap Overflows
  - Stack Overflows
- Integer Overflows
- *Format-String-Angriffe*
- *(SQL) Injection*
- [Cross Site Scripting]

Exkurs: relationale Datenbanken und SQL

- SQL ist eine deklarative Sprache zur Manipulation relationaler Datenbanken
  - Structured Query Language
  - Entstanden in den 1970er Jahren
  - Universell standardisiert und von nahezu allen relationalen Datenbanken unterstützt
- Vorsicht: starke Vereinfachung!
- Eine relationale Datenbank besteht aus einer Menge von Tabellen
- Beispiel: eine Tabelle mit Einträgen Id, Name, Adresse, Kreditkartennummer



Einfache SQL-Befehle

- Mit SQL kann man Datenbanktabellen verwalten
- Beispiel: Erstellen von Tabellen

```
CREATE TABLE customers (
   id int;
   name varchar (30),
   adress varchar (40),
   ccnum varchar (16),
   expiry varchar (4),
   password varchar (16);
   last_updated date);
```

- Löschen von Tabellen:

```
DROP TABLE customers
```

Einfache SQL-Befehle (Forts.)

- Einfügen von Werten:

```
INSERT INTO customers VALUES
  (354, 'John Doe', 'A5, 6',
  '1234567812345678', '0606', 'pa$sw0rd'
  'Nov-24-2006');
```

- Aktualisieren von Werten:

```
UPDATE customers
   SET name = 'John Doe jun.',
   address = 'A5, 7' WHERE id = 354;
```

- Löschen von Werten:

```
DELETE FROM customers
   WHERE id = 354;
```


Der `select`-Befehl

- Zur Abfrage und Anzeige von Einträgen in der Datenbank gibt es den Befehl select
- Syntax:

```
SELECT [ALL | DISTINCT] <select list>
   FROM <table reference list>
   WHERE <search condition list>
   [ORDER BY <column designator> [ASC | DESC]
          [, <column designator> [ASC | DESC]]
    ...]
```

- Beispiel: Alle Kunden, die in "A5, 6" oder "A5, 7" wohnen:

```
SELECT id, name, address
   FROM customers
   WHERE address = 'A5, 6' OR address = 'A5, 7';
```

Kommandozeile und Kommentare

- SQL-Befehle werden entweder auf der Kommandozeile oder in einem Skript gestartet
  - Auf der Kommandozeile: SQL-Interpreter
- Normalerweise erst Verbindung zur Datenbank aufbauen
  - Login mittels Name und Passwort

- Auf der Kommandozeile bzw. in Skripten kann man Kommentare angeben
  - Kommentarzeichen `--` (doppeltes Minuszeichen)
  - Alles, was auf der Kommandozeile im Befehl folgt, wird ignoriert
- Beispiel:

```
SELECT * FROM customers -- get all at once
```


SQL-Injection-Angriffe

- Webanwendungen besitzen typischerweise:
  - eine Datenbank
  - die Möglichkeit, dynamische Webseiten zu erstellen
- Oft:
  - mySQL-Datenbank und PHP
- Aber auch alle anderen Sprachen, die auf eine Datenbank zugreifen können, haben dasselbe Problem:
  - Perl, Python, Java, ASP, C, C++, ...
- Problem:
  - SQL-Abfragen werden durch Verkettung von Strings zusammengebaut
  - Anfragen enthalten User Input
- Ein Angreifer kann durch geschickte Wahl der Eingabe die Semantik der Abfrage ändern

Unterschiedliche Semantiken

- Was machen die folgenden SQL-Befehle?

```
SELECT name, address, ccnum
   FROM customers
   WHERE id = 345;

SELECT name, address, ccnum
   FROM customers
   WHERE id = 345 OR 1=1;

SELECT name, address, ccnum
   FROM customers
   WHERE id = 345; DROP TABLE customers;
```


Unterschiedliche Semantiken (Forts.)

- Was passiert bei folgenden Befehlen?

```
INSERT INTO customers (name)
  VALUES 'Sverre H. Huseby')

INSERT INTO customers (name)
  VALUES ('James O'Connor')

SELECT * FROM customers
  WHERE name = 'Sverre H. Huseby' AND
  password = 'pa$sw0rd'

SELECT * FROM customers
  WHERE name = 'Sverre H. Huseby' -- AND
  password = 'pa$sw0rd'
```

Angreifbarer Java-Code

- Ausschnitt aus einem Java-Programm zur Manipulation der Datenbank:

```
...
userName = request.getParameter("user");
password = request.getParameter("pass");
query = "SELECT * FROM customers "
  + "WHERE name='" + userName + "' "
  + "AND password='" + password + "'";
...
```

- Was kann man jetzt alles anstellen? Was muss man dafür eingeben?

Deaktivierung Passwortauthentifikation

- Eingabe von

```
Felix Freiling' --
```

- Java-Code:

```
query = "SELECT * FROM customers "
  + "WHERE name='" + userName + "' "
  + "AND password='" + password + "'";
```

- erzeugt dadurch:

```
SELECT *  FROM customers
  WHERE name='Felix Freiling' -- ' AND password='';
```

- Deaktivierung der Passwortauthentifikation!

Deaktivierung (Forts.)

- Eingabe von

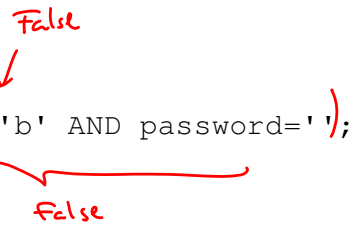
```
Felix Freiling' OR 'a'='b
```

- Java-Code:

```
query = "SELECT * FROM customers "
  + "WHERE name='" + userName + "' "
  + "AND password='" + password + "'";
```

- erzeugt dadurch:

```
SELECT *  FROM customers
  WHERE name='Felix Freiling' OR 'a'='b' AND password='';
```



- AND bindet stärker als OR!

Löschen von Daten

- Eingabe von

```
'; DELETE FROM customers --
```

- Java-Code:

```
query = "SELECT * FROM customers "
  + "WHERE name='" + userName + "' "
  + "AND password='" + password + "'";
```

- erzeugt dadurch:

```
SELECT *  FROM customers
  WHERE name=''; DELETE FROM customers -- AND password='';
```

- Default: alle Einträge in Datenbank löschen

Einfügen von Daten

- Eingabe von

```
'; INSERT INTO customers VALUE ... --
```

- Java-Code:

```
query = "SELECT * FROM customers "
  + "WHERE name='" + userName + "' "
  + "AND password='" + password + "'";
```

- erzeugt dadurch:

```
SELECT *  FROM customers
  WHERE name=''; INSERT INTO customers VALUE ... -- AND
    password='';
```

- Einfügen eines neuen Eintrags!

Herausfinden der Tabellenstrukur

- Für einen Angriff ist der Aufbau der Tabellen wichtig
  - Name, Attribute, Wertebereiche
- Verschiedene Wege, das herauszufinden:
  - Open Source Software (kein Problem)
  - Fehlermeldungen des Datenbankinterpreters
- Beispiel:
  - Falsche Attributnamen verwenden
  - Oft geben die Datenbanken die komplette Tabellenstruktur der Datenbank aus

Problematischer Java-Code

```
import java.*;
import java.sql.*;
...
public static boolean doQuery(String Id) {
  Connection con = null;
  try {
    Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver";
    con = DriverManager.getConnection("jdbc:microsoft:sqlserver:" +
        "//localhost:1433", "sa", "$3cre+");
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("SELECT ccnum FROM cust WHERE id = "
        + Id);
    while (rs.next()) {
      // do something with a line of results
    }
    rs.close(); st.close();
  }
  catch ...
  finally {
    try {
      con.close();
    } catch (SQLException e) {}
  }
  return true;
}
```


Schlechter PHP-Code

```
<?php
  $db = mysql_connect("localhost", "root", "$$sshhh...");
  mysql_select_db("Shipping", $db);
  $id = $HTTP_GET_VARS["id"];
  $qry = "SELECT ccnum FROM customers WHERE id=%$id%";
  $result = mysql_query($qry, $db);
  if ($result) {
    echo mysql_result($result, 0, "ccnum");
  } else {
    echo "No result! " . mysql_error();
  }
?>
```

Schlechter SQL-Code

- Auch in SQL kann man SQL Injection machen:

```
CREATE PROCEDURE dbo.doQuery(@query nchar(128))
AS
  exec(@query)
RETURN
```

- Weiteres Beispiel:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
  DECLARE @query nchar(256)
  SELECT @query =
    'select * from cust where id = ''' + @id + '''
  EXEC @query
RETURN
```


Auch String-Ersetzen ist schlecht

- Statt Konkatentation wird String-Ersetzen verwendet
  - Hierbei können dieselben Fehler auftreten
- Beispiel in C#:

```
...
try {
  SqlConnection sql = new SqlConnection(
      @"data source=localhost;" +
      "user id=sa;password=pa$sw0rd;");
  sql.Open();
  string sqlstring="SELECT ccnum" +
    " FROM customers WHERE id=%ID%";
  String sqlstring2 = sqlstring.Replace('%ID%',id);
  SqlCommand cmd = new SqlCommand(sqlstring2, sql);
  ccnum = (string)cmd.ExecuteScalar();
}  catch ...
...
```

Wie findet man solche Schwachstellen?

- Gefahr überall vorhanden, wo
  - Benutzerinput entgegengenommen wird
  - dieser Input nicht auf Validität gefprüft wird
  - Input zur Abfrage einer Datenbank verwendet wird
  - String-Konkatenation oder String-Ersetzen zur Konstruktion der Abfrage genutzt wird
- Alle Datenbankabfragen müssen im Code Review geprüft werden
- Beim Testen vorgehen wie beim Fuzzing:
  - Große Menge an zufällig generierten SQL-ähnlichen Befehlen
  - mit wahllos eingestreuter Interpunktion
- Es gibt auch Tools auf dem Markt, die SQL-Injection erkennen sollen
  - Eigene Applikation mit SQL-Injection-Schwachstellen schreiben und die Tools zuerst darauf testen

Wie kann man SQL-Injection vermeiden?

- Input immer Überprüfen
  - Eingabe niemals vertrauen
  - Am besten: mit regulären Ausdrücken so viel Struktur im Input prüfen, wie möglich
- Niemals String-Konkatenation oder String-Ersetzen zum Aufbau von SQL-Statements verwenden
  - Parametrisierte SQL-Statements (manchmal auch als *prepared statements* bezeichnet) verwenden
  - Hierbei werden SQL-Befehle mit Platzhaltern vorgefertigt (prepared)
  - Mit speziellen *bind*-Befehlen kann man (geprüfte) Eingaben an die Platzhalter binden

Weitere Informationen

- Howard, LeBlanc, Viega: 19 Deadly Sins of Software Security, Kapitel 4
- Huseby: Innocent Code, Kapitel 2
- Steve Friedl's "SQL Injection Attacks by Example", http://www.unixwiz.net/techtips/sql-injection.html

- Anmerkung:
  - In den meisten Code-Beispielen waren Login und Passwort der Datenbank im Quelltext fest einprogrammiert
  - Das ist auch gefährlich
  - Besser: kritische Daten von ausserhalb holen
    - in eigene Zugriffsfuktionen kapseln
    - Dateien ausserhalb des normalen Zugriffs von Benutzern ablegen

Von SQL-Injection zu Code-Injection

- SQL-Injection ist nur der Anfang
- Oft rufen Systeme mit Benutzereingaben Kommandointerpreter auf
  - Beispiel: UNIX shell
- Berühmtes Beispiel: alter IRIX login
  - Man konnte sich einloggen oder alternativ Dokumentation ausdrucken und dafür den richtigen Drucker angeben
- Quellcode tat in etwa dies:

```
char buf[1024];
snprint(buf, "system lpr -P %s",
    user_input, sizeof(buf)-1);
system(buf); // execute buf in shell
```

Eingabe: XYZ; xterm&

Worauf muss man achten?

- Benutzereingaben überprüfen
- Vorsicht bei Übergabe von Benutzereingaben an das System
- Kritische Befehle:
  - C/C++ (Posix): system(), popen(), execlp(), execvp()
  - C/C++ (Win32): ShellExecute() und Konsorten
  - Perl: system, exec, Hochkommata `...`, open mit pipe und noch ganz viel mehr
  - Java: Class.forName(String name), Class.newInstance(), Runtime.exec()
- Beim Testen:
  - einfach mal Semikolon eingeben und sich selbst eine Mail schicken lassen ("; mail a@b.c Test")
  - vorher und/oder nachher beliebige Hochkommata angeben

Vermeiden von Code-Injection

- Niemals einen Kommandointerpreter aus der Programmiersprache heraus aufrufen
- Wenn das nicht geht, dann:
  - Benutzereingabe überprüfen
  - Falls Eingabe nicht okay:
    - Befehl nicht ausführen!
    - Befehl nicht auf den Bildschirm zurückausgeben (siehe Cross Site Scripting)
    - Dem Angreifer nicht zu viel Informationen geben
      - Vage bleiben, zum Beispiel "Invalid Character"
    - Den Fehler in Logfile abspeichern
      - Aufpassen, dass der Angriff nicht eigentlich dem Logfile galt
  - Prüfen der Eingabe durch eine Positivliste
- Zusätzliche Abwehrmaßnahmen:
  - Taint Mode in Perl
  - Eigene Wrapper-Funktionen für Systemaufrufe schreiben, die Tests kapseln

Übersicht

- Race Conditions
- [Buffer Overflows]
  - Heap Overflows
  - Stack Overflows
- Integer Overflows
- *Format-String-Angriffe*
- *(SQL) Injection*
- *[Cross Site Scripting]*

Zusammenfassung

- Race Conditions
- Buffer Overflows
  - Heap Overflows
  - Stack Overflows
- Integer Overflows
- *Format-String-Angriffe*
- *(SQL) Injection*
- Cross Site Scripting

- *Ausblick:*
  - Sicherheitsprobleme im Netzwerk
  - Sicheres System-Setup und -Management