Malware-Analyse und Reverse Engineering

13: Rückblick über das Semester

29.6.2017

Prof. Dr. Michael Engel

Überblick

MARE 2017:

- Einführung und Motivation, Malware-Historie
- Übersetzen und Linken von Programmen, Binärformate
- Laufzeitverhalten von Programmen, dynamisches Linken
- Unix-Prozesse, Programmstart, Stackframes und -verhalten
- Buffer Overflows und Funktionsaufrufe
- Return Oriented Programming
- Schutzmechanismen gegen Buffer Overflows
- Statische und dynamische Codeanalyse
- Polymorphe Viren und selbstmodifizierender Code
- Systemaufrufe
- Verhaltensanalyse von Viren
- Rootkits

Einführung und Motivation, Malware-Historie

HOCHSCHULE COBURG

Kurze Geschichte der Computerviren

- Definition Reverse Engineering und Malware
- Historie von Malware
- Verbreitung
- Beispiele
- Code und Funktionsweise eines einfachen Beispielvirus
- Rechtliche Implikationen

Übersetzen und Linken von Programmen, Binärformate

Compiliervorgang:

- Separate Schritte:
  - Übersetzen: .c -> .o
  - Linken: .o -> ELF

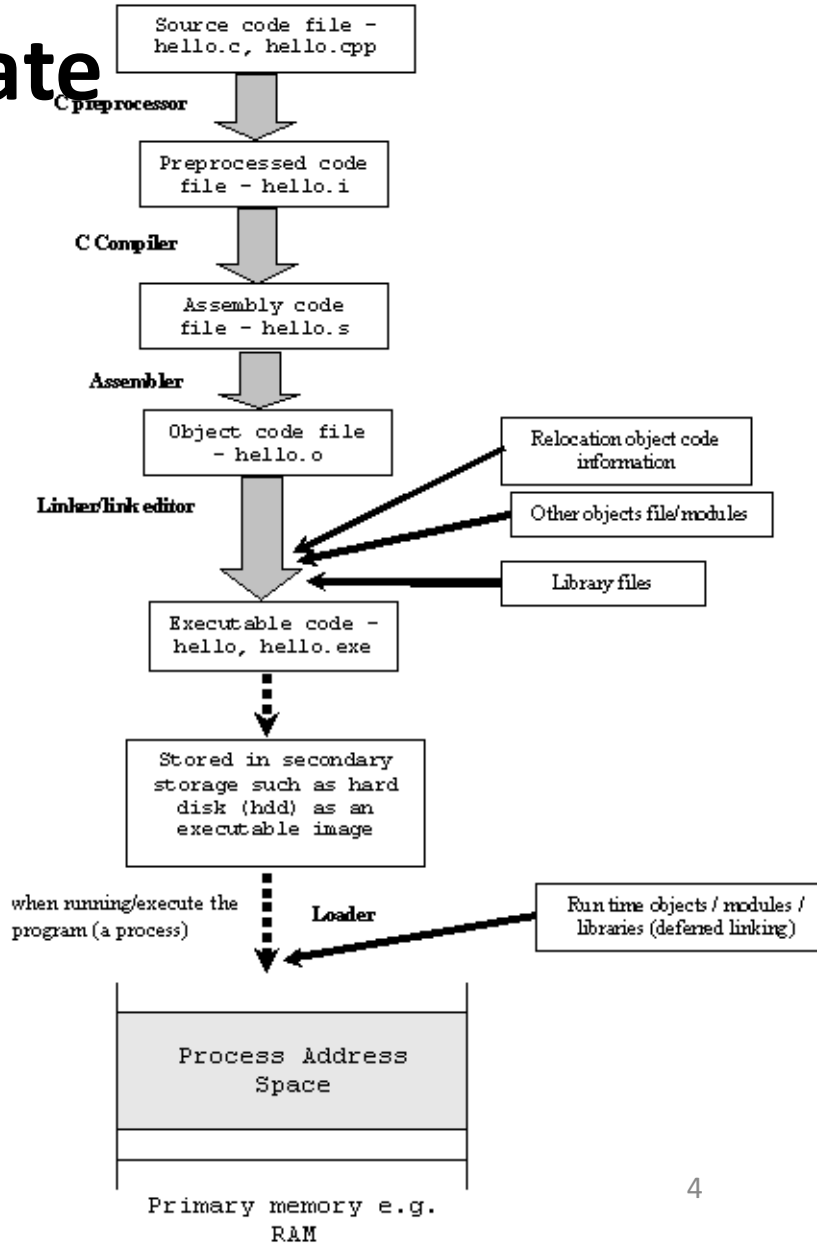
Binärformate

- ELF: Struktur, Sektionen (.text, ...)
- Werkzeuge zur Analyse (readelf, ...)
- Symbole und Adressen

Linker

- Symbole
- Relokationen

Endianness



Laufzeitverhalten von Programmen, dynamisches Linken

Virtueller Speicher in Linux

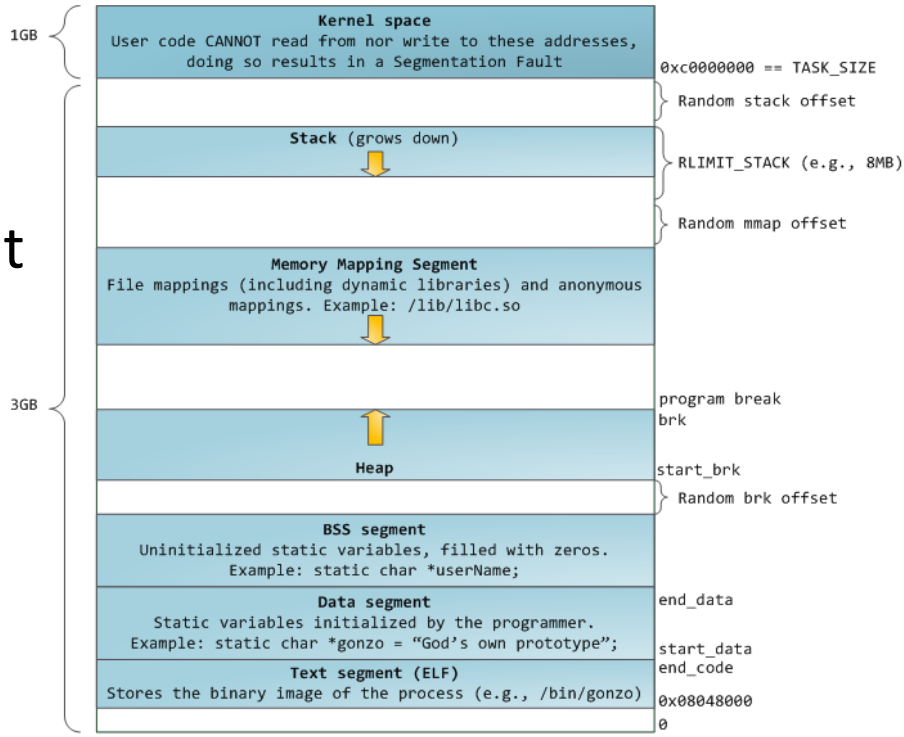
- Seitentabellen, Umsetzung virtuelle -> physikalische Adressen
- Virtuelles Speicherlayout von Prozessen

Programme und Prozesse

- Prozessstart:
  Übergang Programm -> Prozess
- Linkerskripte und Speicherlayout

Speicherlayout von Programmen zur Laufzeit

- Dynamisches Linken
- Shared Libraries



Unix-Prozesse, Programmstart, Stackframes und -verhalten

Prozessmodell und Prozesserzeugung in Unix

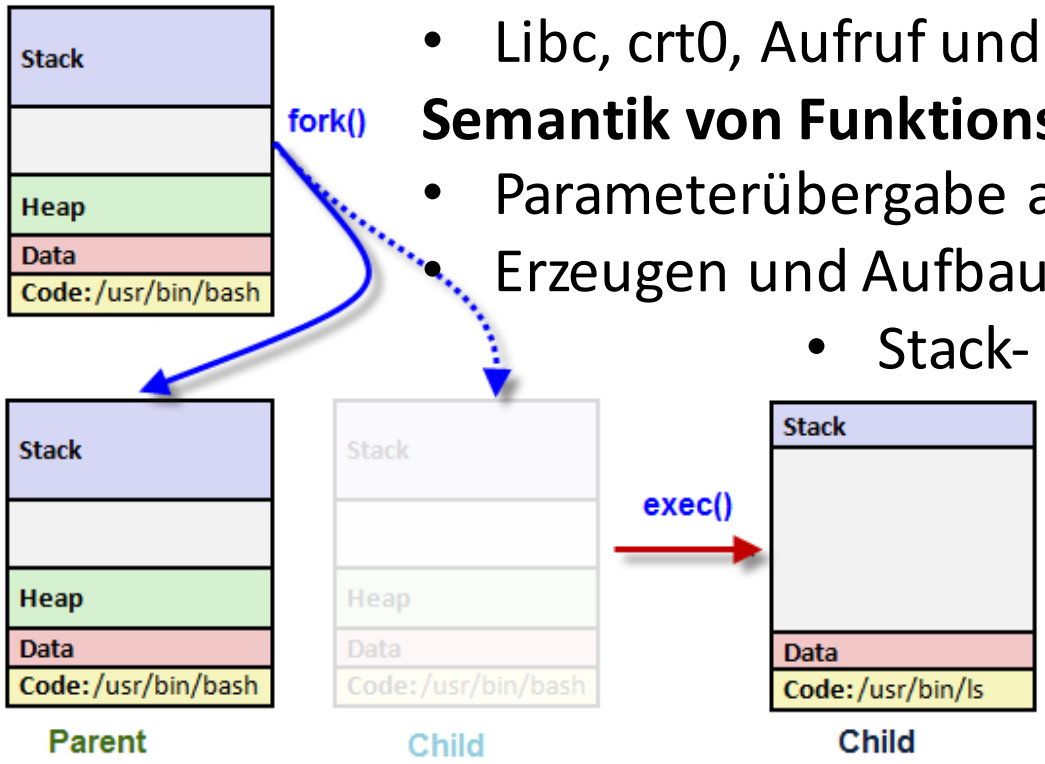
- fork und exec, Eltern-/Kindprozesse

Programmstart

- Libc, crt0, Aufruf und Parameter von main

Semantik von Funktionsaufrufen

- Parameterübergabe auf dem Stack
- Erzeugen und Aufbau von stack frames
  - Stack- und Framepointer



Buffer Overflows und Funktionsaufrufe

Funktionsaufrufe auf x86 (32 Bit) und x64 (64 Bit) x86-Linux

- Parameterübergabe: Stack vs. Register
- Funktionsaufruf in Assembler
- Funktion des Base (Frame) Pointers

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Stack-Probleme bei C-Funktionen

- Lokale und globale Variable
- Lokale Arrays, unsichere libc-Funktionen
- Buffer Overflow und Stackverhalten

Angriffsmethoden

- Ausführbarer Stack: Malware-Code im Buffer “mitliefern”
- Nicht ausführbarer Stack: Return to libc ausnutzen

Return Oriented Programming

Ausführbarer vs. nicht ausführbarer Stack-Speicherbereich

- Data Execution Protection (DEP = W^X)
- Details zu return to libc

Problem bei x64: Parameterübergabe an Funktionen

- Register statt Stack: Code zum Setzen von Registern notwendig
- Nicht auf Stack => aber Codefragmente im .text-Segment
- Anspringen kurzer „Gadgets“ mit Return (0xc3) am Ende:
  Return oriented programming (ROP)

Ausnutzen von ROP

- Finden von Gadgets im Speicher
- Verketten von Gadgets für erweiterte Funktionalität
- Gadgets mit weiterer Funktionalität (nicht nur Laden von Registern)

Schutzmechanismen gegen Buffer Overflows

Erkennen von Buffer Overflows

- Stack canaries
- Stack smashing protector
- Sicherheit von canaries

Schutz vor Buffer Overflows

- Shadow stacks

Schutz vor ROP

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP)
- Details zu virtuellem Speicher bei x86 und Adressumsetzung

Statische Codeanalyse

- Modellierung von Strings und Constraint-Analyse

Statische und dynamische Codeanalyse (1)

HOCHSCHULE COBURG

Mehr statische Codeanalysen

- Analyse von Codesignaturen (Finden von Bytefolgen in Binaries)
- Probleme der signaturbasierten Erkennung
  - false positives und false negatives
  - Größe der Signaturdatenbank und Zeitaufwand für Scans

Umgehen der Signaturanalyse

- Mutation (Obfuskation) von Viren:
  - Einfügen von NOPs
  - Codeumsortierung und Einfügen von Sprüngen
  - Ersetzen von Instr. durch semantisch identische Folgen

E800	0000	00(90)*	5B(90)*
8D4B	42(90)*	51(90)*	50(90)*
50(90)*	0F01	4C24	FE(90)*
5B(90)*	83C3	1C(90)*	FA(90)*
8B2B			

Statische und dynamische Codeanalyse (2)

Polymorphe Viren

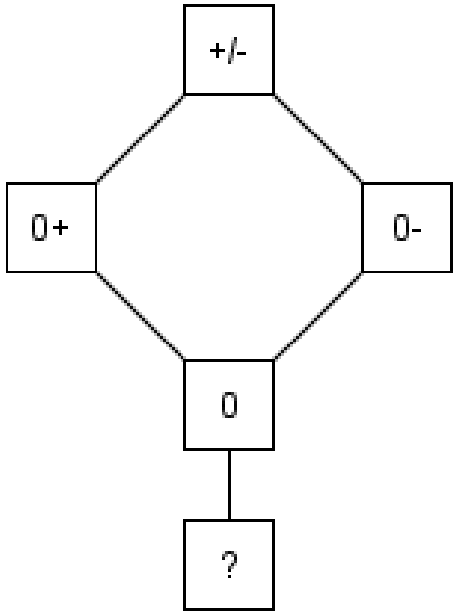
- Kombination verschiedener Obfuskationstechniken

Komplexere statische Analyse: Abstrakte Interpretation

- Abstraktion von Programmzuständen
- Abstraktion von Programmsemantik
- Anwendung: Sprunganalyse

Dynamische Analysen

- Kontrollflussverfolgung und Funktionsaufruf-graphen



Polymorphe Viren und selbstmodifizierender Code

HOCHSCHULE COBURG

Reverse Engineering und das Vermeiden davon

- Erschweren des Disassemblierens: Sprünge mitten in Opcodes
- Arten von Disassemblern
  - Linear Sweep vs. Recursive Traversal
- Definition und Semantik von Basisblöcken

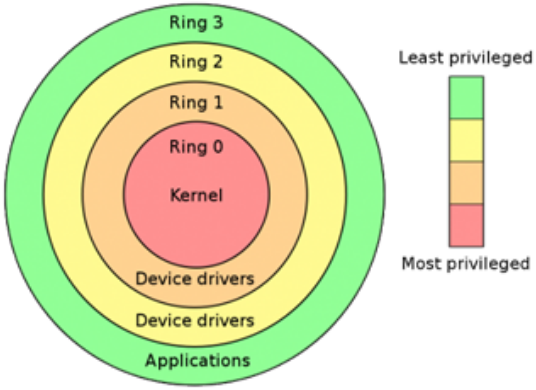
Mehr zu polymorphen Viren

- Selbstentschlüsselende Viren
- Problem der Bestimmung des aktuellen PC
- Obfuskation durch Instruktionsersetzung
- Selbstmodifizierender Code

Systemaufrufe

Systemaufrufe in Linux

- Unterschiede zu und Gemeinsamkeiten mit Funktionsaufrufen
- Schutzringe und Privilegienübergang
- Überblick über Funktionalität von Systemaufrufen
  - Systemaufruftabelle und -nummern
  - Zusammenhang libc <-> Systemaufruf
- Systemaufrufe in Assembler
  - SYSCALL-Befehl
  - Parameterübergabe und –rückgabe
- Verfolgen von Systemaufrufen
  - strace und ptrace



Verhaltensanalyse von Viren

Beobachten des Laufzeitverhaltens von Programmen

- Schon gesehen: Kontrollflussverfolgung
- Analogie zum Immunsystem
- Analyse von Folgen von Systemaufrufen
  - Systemaufruf-Traces mit lookahead
  - Einschränkungen des lookahead-Ansatzes
    - Nur Syscall-Nummern (keine Parameter) analysiert, kein Timing, keine Analyse von ausgeführten Instruktionen
- Angriffe auf lookahead-Verfahren
  - Einfügen sinn- und nutzloser Systemaufrufe
  - Verwenden von Systemaufruf-Folgen, die kürzer als der lookahead sind

Rootkits

Permanentes Kompromittieren eines angegriffenen Systems

- Installation und Verbergen von Malware-Komponenten
- Unterscheidung Kernel-/Usermode-Rootkit
  - Verwenden bereits bekannte Methoden (z.B. LD_PRELOAD)
  - Kernelmodule als Grundlage von Kernelmode-Rootkits
- Erkennung der Veränderung von Binaries
  - Checksumming mit Hashfunktionen
  - Probleme mit Hashfunktionen: Kollisionen in md5 und sha1
- Speicherbasierte Rootkits
  - Nach Reboot verschwunden, ohne Spuren zu hinterlassen
- VM-basierte Rootkits
  - Grundlagen HW-Virtualisierung: Ring -1
  - Beispiel “Blue Pill” und Erkennungsmöglichkeiten

Fazit: Was sollte das alles?

Kernkompetenzen aus der Vorlesung

- Wissen über den Übersetzungsvorgang und das Laufzeitverhalten von Software
  - Prozesse, Funktions- und Systemaufrufe, Stackverhalten
- Typische Sicherheitslücken und Möglichkeiten, sie auszunutzen
  - Buffer Overflows, return to libc, ROP, ...
  - Typische C-Programmierfehler
- Hardware- und Software-basierte Schutzmechanismen
  - DEP, ASLR, statische und dynamische Codeanalyse, ...
- ...und wie diese wieder umgangen werden können... usw.
  - Polymorphe Viren, verschlüsselter/selbstmodif. Code, ...
- Definition, Strukturen und Funktion von Rootkits

Fazit: Was sollte das alles?

Kernkompetenzen aus den praktischen Übungen

- Bessere Einblicke in Linux auf Systemebene
  - Compiler, Linker, Shared Libraries, Stackaufbau, ABIs ...
- Umgang mit Analysetools (ELF-Dump, Disassembler, Debugger)
- Analyse von Programmen auf Maschinenebene
- Implementierung typischer Angriffe: Buffer overflow, ROP
- Analyse und Umgehen von Schutzmechanismen: Canaries
- Implementieren dynamischer Analysen: Kontrollflussverfolgung
- Verwenden von Systemaufrufen, Parameterübergabe in ASM
- Tracing von Systemaufrufen als weitere dynamische Analyse
- Verbergen von Programmen als Rootkit-Teilfunktionalität
- Hausaufgabe: „Knobeln“ mit selbstmodifizierendem Code

Das war „schon“ alles...