F Sicherheit von Programmen / Anwendungen

F.1 Überblick

- Schwachstellen und Bedrohungen bei der Ausführung von Software
- Exploits
- Return-oriented Programming
- Mobiler Code

F.2 Schwachstellen und Bedrohungen bei der Ausführung von Software

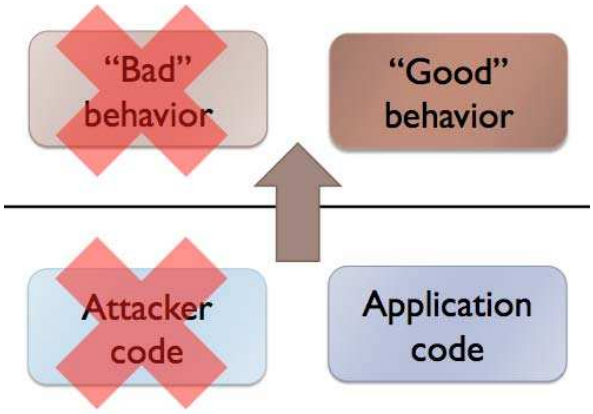
- ◆ Quelle des Codes unklar
- ◆ manipulierter Code
- ◆ fehlerhafter Code
- ◆ unbekannte Funktionalität

- Gegenmaßnahmen
  - ◆ Quelle sicherstellen, Manipulationen erkennen
    - ➤ Hashes veröffentlichen
    - ➤ Code signieren
  - ◆ Sicherheit durch Softwareentwicklungsprozess garantieren
    - ➤ Code Reviews
    - ➤ Unterstützung durch Programmiersprachen und Entwicklungswerkzeuge

F.3 Exploits

F.4 Return-oriented Programming

- *Bad code versus bad behavior*



  - ➤ Problem: diese Annahme ist falsch!

- Ansatz für ***Return-oriented Programming***:
  - ➤ ausreichend grosse Menge von Programmcode
    - ➥ erlaubt einem Angreifer beliebige Berechnungen und Abläufe
  - ➤ Voraussetzung: Integretät des Kontrollflusses ist verletzbar!

1 Buffer-Overflow-Angriffe

- Manipulation der Rücksprungadresse
  + Injektion von Programmcode auf dem Stack
  - ◆ Bösartiger Code wird auf dem Stack abgelegt und dort ausgeführt
  - ◆ Gegenmassnahme: W-xor-X
    - ➤ schreibbare Speicherseiten sind nicht ausführbar
    - ➤ Voraussetzung: MMU unterstützt "XD" oder "NX"-Bit (neuere Intel- und AMD-Prozessoren, viele RISC-Prozessoren)
- Manipulation der Rücksprungadresse
  + *Return-into-libc*
  - ◆ Bösartiges Verhalten wird durch Manipulation des Kontrollflusses bewirkt
    - ➤ Aufruf von system(), exec(), printf(), etc.
    - ➤ keine Code-Injektion erforderlich
  - ◆ nur eingeschränkte Möglichkeiten, abhängig von den in der libc vorhandenen Funktionen

  ➥ **diese Annahme ist falsch!**

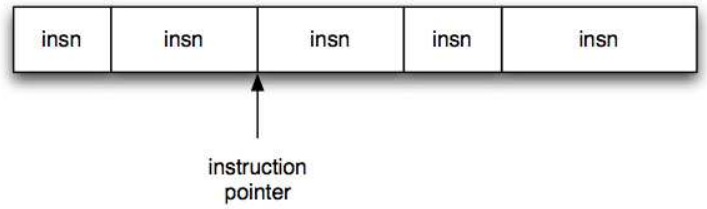
2 Verallgemeinerung von Return-into-libc

- libc ist eine sehr umfassende Codebasis
- es ist nicht notwendig, komplette Funktionen auszuführen
  - ➤ jeder Sprung in Code der libc kommt beim nächsten **`ret`**-Befehl zurück!
- wenn die libc eine Turing-vollständige Menge von Codeschnipseln (jeweils abgeschlossen mit `ret`) enthält, kann man beliebige Programme zusammenstellen!
  - **?** findet man diese Codeschnispel
  - **!** ja

➥ erlaubt den Missbrauch jedes angreifbaren Programms für beliebiges Fehlverhalten

3 Grundprinzipen von Return-oriented Programming

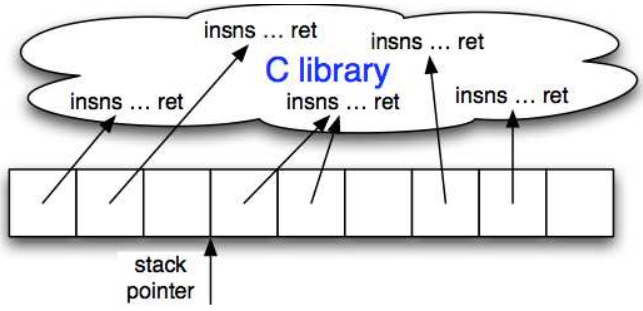
- Normaler Programmablauf



- ➤ *Instruktion pointer* (%eip) legt fest, welche INstruktion als nächstes geholt und ausgeführt wird
- ➤ nach Ausführung der Instruktion wird %eip automatisch erhöht, so dass es auf die nächste Instruktion zeigt
- ➤ Kontrollfluss kann durch Änderung von %eip gesteuert werden

3 Grundprinzipen von Return-oriented Programming (2)

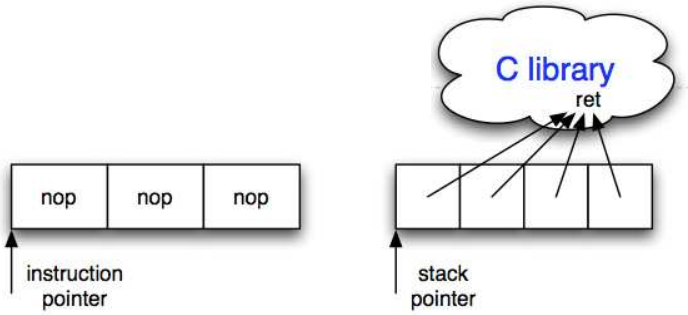
- Return-oriented Programming auf Instruktionsebene



- ➤ *Stack pointer* (%esp) legt fest, welche Instruktionsfolge als nächstes ausgeführt wird
- ➤ der Prozessor erhöht %esp nicht automatisch — aber der `ret`-Befehl am Ende einer Instruktionsfolge tut dies!
- ➤ Einstieg in solch ein Programm: einmalig manipulierte Return-Adresse auf dem regulären Stack

4 Baugruppen für RoP-Programme

★ No-op

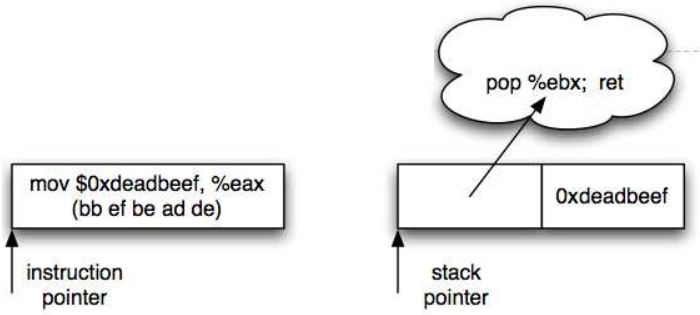


➤ tut nichts erhöht lediglich %eip

➤ Äquivalent bei RoP
– %esp zeigt auf ein `ret`-Anweisung
– %esp wird inkrementiert

4 Baugruppen für RoP-Programme (2)

★ Laden einer Konstante in einer Register (Immediate Constant)

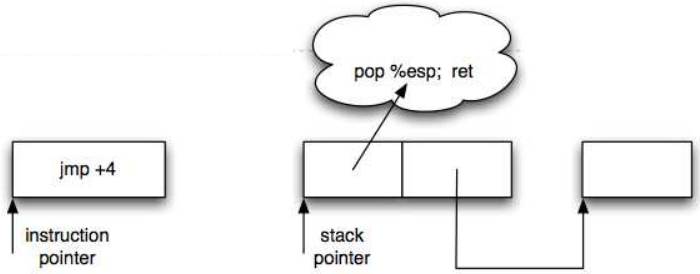


➤ Instruktion können direkt Konstanten enthalten

➤ Äquivalent bei RoP
– zu ladende Konstante wird auf dem Stackabgelegt
– `pop`-Anweisung lädt vom Stack in das Zielregister
– %esp wird darurch automatisch inkrementiert und zeigt dann auf die nächste Befehlsfolge

4 Baugruppen für RoP-Programme (3)

★ Sprünge

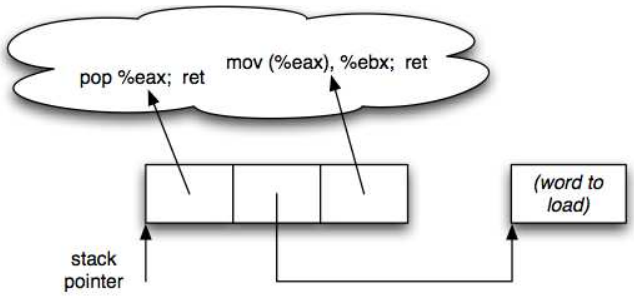


➤ Normale Programmierung
– %eip wird auf neuen Wert gesetzt (ggf. abhängig von einer Bedingung)

➤ Äquivalent bei RoP
– %esp wird auf neuen Wert gesetzt (ggf. abhängig von einer Bedingung)

4 Baugruppen für RoP-Programme (4)

★ Laden eines Werts aus dem Speicher in ein Register



◆ In diesem Fall wird mehr als ein Code-Schnipsel benötigt, um eine Baugruppe zu realisieren

➤ erster Schritt
– Speicheradresse des zu ladenden Werts nach %eax laden

➤ zweiter Schritt
– Inhalt von (%eax) in %ebx laden

■ Schreiben in Speicherzellen erfolgt analog

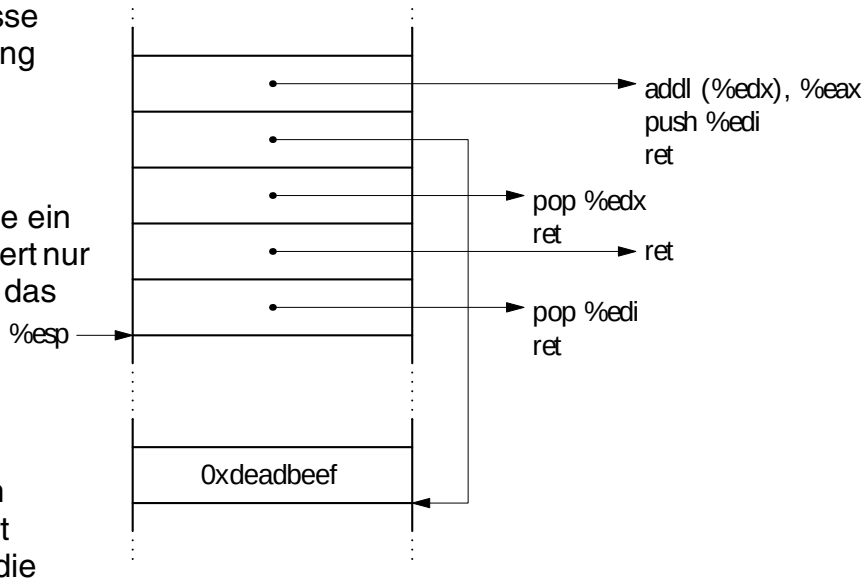
4 Baugruppen für RoP-Programme (5)

- ★ Arithmetik
  - ◆ ALU-Modell
    - ➤ ein Operand ist %eax, der andere eine Speicherposition
    - ➤ das Ergebnis wird entweder in %eax oder die Speicherzelle geschrieben
- ★ Addition
  - ◆ einfachste gefunden Code-Sequenz wäre
    `addl (%edx), %eax; push %edi; ret`
    - ➤ erste Operation arbeitet wir gewünscht, Ergebnis steht in %eax
    - ➤ zweiter Teil (`push-ret`-Folge) ist problematisch
  - ◆ Probleme:
    - ➤ push schreibt einen Wert auf den Stack und zerstört damit die Sprungadresse von ret
    - ➤ Baugruppe wird dadurch insgesamt zerstört (für die nächste Nutzung, z. B. in einer Schleife)

4 Baugruppen für RoP-Programme (6)

- ★ Addition (Fortsetzung)
  - ◆ Lösung des ersten Teilproblems: %edi muss sinnvolle Zieladresse enthalten

- ➤ vor `addl` die Adresse einer `ret`-Anweisung nach %edi laden
- ➤ Adresse der `ret`-Anweisung wirkt wie ein `nop` und inkrementiert nur %esp (neutralisiert das `push`)
- ➤ Endefffekt: `push` legt einen `nop`-Befehl auf den Stack (aber zerstört dabei Verweis auf die `addl`-Befehlssequenz!)



4 Baugruppen für RoP-Programme (7)

★ Addition (Fortsetzung)

◆ Lösung des zweiten Teilproblems: Baugruppe wird so erweitert, dass sie zuerst dynamisch die Adresse der addl-Sequenz an ihr Ende schreibt

(1) Sprung auf `addl`-Sequenz in %ecx merken

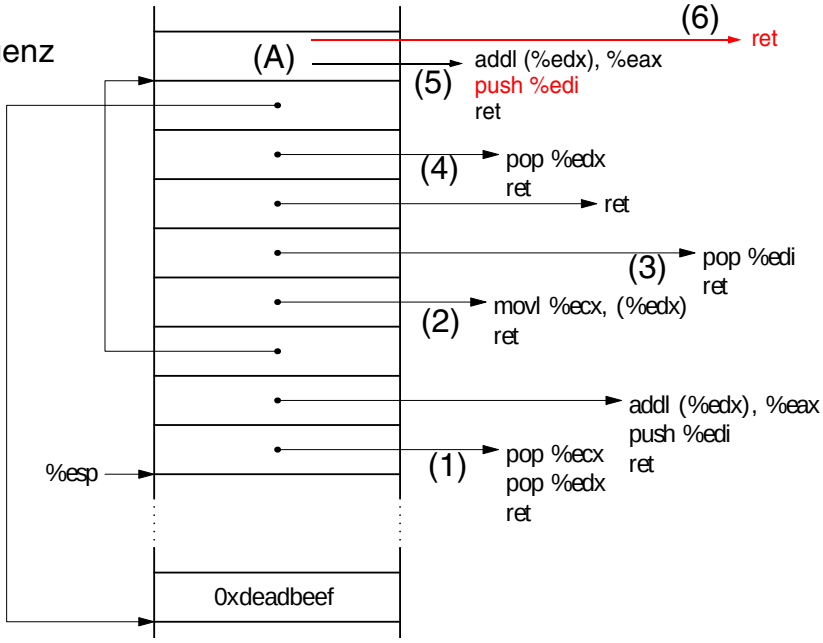
(2) %ecx nach (A) schreiben

(3) nop-Befehl → %edi (wird am Ende der `addl`-Sequenz (A) überschreiben)

(4) %edx-Operand laden

(5) addl-Sequenz anspringen, am Ende wird (A) überschrieben

(6) nop-Befehl



4 Baugruppen für RoP-Programme (8)

★ Addition (Fortsetzung)

◆ die verwendeten Codeschnipsel sind in der libc so auffindbar

➤ pop %ecx; pop %edx; ret

➤ addl(%edx), %eax; push %edi; ret

➤ movl %ecx, (%edx); ret

➤ pop %edi; ret

➤ pop %edx; ret
(einfach einen Befehl weiter hinten in die erste Sequenz springen)

★ Subtraktion

➤ neg %eax; ret in Kombination mit Additions-Baugruppe

★ XOR, And, Or, Not, Shifts und Rotates

★ Multiplikation

◆ keine passende Sequenz gefunden, muss aus Addition und logischen Operationen aufgebaut werden

4 Baugruppen für RoP-Programme (9)

★ Bedingte Sprünge

◆ relativ aufwändig, Realisierung in drei Schritten

➤ Operation ausführen, die die gewünschten Flags in %eflags setzt

➤ Flags in general-purpose Register transferieren und das gewünschte Flag mit logischen Operationen isolieren

➤ das gewünschte Flag nun nutzen, um %esp um die gewünschte Sprungweite zu modifizieren

4 Baugruppen für RoP-Programme (9)

★ Systemaufrufe

◆ viele Systemaufrufe werden über einfache Wrapper-Funktionen in der lic angesprungen

➤ Argumente vom Stack in Register transferieren und Systemaufrufnummer in %eax ablegen

➤ trap in den Kern (indirekt über linux-gate.so.1)

➤ Fehlerabfrage und Transfer des Ergebniswerts nach %eax

◆ syscall-Baugruppe

➤ Argumente und Systemaufrufnummer selbst vorbereiten

➤ eine Wrapper-Funktion hinter diesen Schritten anspringen

5 Return-oriented Programming – Resumee

- Werkzeugunterstützung
  - ➤ Suche von geeigneten Code-Schnipseln
  - ➤ Compiler zur generierung von RoP-Code aus C-ähnlichem Quellcode
- Konzept ist weitgehend Prozessorunabhängig
  - ➤ Anwendbarkeit für verschiedene RISC- und CISC-Prozessoren wurde gezeigt
- Funktionsmanipulation wurden erfolgreich demonstriert
  - ➤ Z80-basierte Wahlcomputer wurden umprogrammiert, um Ergebnisse zu fälschen obwohl Sicherheitsmechanismen die Manipulation des Programmcodes unterbinden
- Gegenmassnahmen
  - ◆ Adress-Space-Randomization (aber Kernel-Code ist meistens in den Adressraum der Anwendungen fest eingeblendet und eine ideale Quelle)
  - ◆ Trennung von Aufrufstack und Funktionsdaten

F.5 Mobiler Code

1 Überblick

- zunehmende Verbreitung verteilter Softwaresysteme
  - ➤ Agentensysteme
  - ➤ Applets und Servlets, ActiveX-Controls
  - ➤ Aktive Netzwerke
  - ➤ Plug-ins
  - ➤ aktive Inhalte von Web-Seiten und E-Mails
- Mobiler Code
  Software, die auf einem entfernten, potentiell nicht vertrauenswürdigen Rechner generiert wurde und die auf einem Gastrechner ausgeführt wird
  - ➤ grundlegender Unterschied zu Client-Server-Modell:
    Software ist auf den Knoten resident, es werden nur Daten (Aufrufparameter) transportiert
- Plattformabhängig (z. B. ActiveX) oder -unabhängig (z. B. Java)

2 Bedrohungen

- Unsicherheit über die tatsächliche Quelle des Codes
  - ➤ wurde der Code bereits an der Quelle verfälscht?
- Übertragung des Codes über ein Transportmedium
  - ➤ was passiert unterwegs mit dem Code?
- Angriffe auf den mobilen Code auf dem Gastrechner
  - ➤ sind die ausführenden Subjekte berechtigt?
  - ➤ sind ggf. lokale, sensible Daten des Codes (z. B. Kommunikationsschlüssel) gegen unberechtigte Zugriffe geschützt?
  - ➤ kann der Code vor oder bei der Ausführung verfälscht werden?
  - ➤ was tut die Ausführungsumgebung tatsächlich mit dem Code?
  - ➤ was erwartet der Absender des Codes von der Ausführung?
  - ➤ bewegt sich der Code evtl. danach weiter oder zurück zum Absender (Agentensystem) - und in welchem Zustand ist er dabei?

2 Bedrohungen (2)

- Angriffe des mobilen Codes auf den Gastrechner
  - ◆ was soll der Code tun, was tut er tatsächlich?
    - ➤ auf welche Daten kann er zugreifen?
    - ➤ welche Daten überträgt er wohin?
    - ➤ welche Ressourcen kann er belegen? (Denial-of-Service-Aktivitäten)
    - ➤ Maskierungsangriffe
      - – Vortäuschen einer falschen Identität
      - – Vortäuschen einer falschen Funktionalität
- ! Sicherheit über die Quelle des Codes bringt nicht unbedingt Sicherheit über die Funktionalität
  - ➤ auch signierter Code kann bedrohliche Funktionalität beinhalten!

3 Beispiel: ActiveX-Controls

- Erweiterung von Microsofts OLE-Technologie
  - ➤ ab Mitte der 1990er Jahre
- Wiederverwendbare Softwarekomponenten in Binärform
  - ➤ Ausführung in spezieller Ausführungsumgebung z. B. Browser
  - ➤ Code wird über Netzwerk aus *Codebase* geladen Angaben z. B. im Inhalt einer Webseite
  - ➤ Ausführung des Codes mit umfangreichen Rechten
    - – Zugriff auf Systemschnittstelle
    - – Zugriff auf Dateisystem
    - – Zugriff auf Netzwerk

➥ Quelle massiver Sicherheitsprobleme

4 Schutz des mobilen Codes

- Verschlüsselungstechniken, Signieren
  - ➤ Sicherstellen der Quelle
  - ➤ Schutz bei Übertragung
  - ➤ Schutz auf dem Gastsystem bis zur Ausführung
  - ! kein Schutz in der Ausführungsumgebung auf dem Gastsystem
- Ausführung nur in vertrauenswürdigen Umgebungen
  - ◆ Basis: Trusted-Computing-Platform Technologie
    - ➤ Konfiguration des Gastsystems kann zuerst über Remote-Attestation-Protokoll abgeprüft werden
    - ➤ spezielle Ausführungsumgebung wird vorab auf Gastsystem geschickt (→ virtuelle Maschine)
    - ➤ mobiler Code wird nur in der speziellen Ausführungsumgebung ausgeführt
  - ◆ Problem:
    - ➤ Qualität von Attestierungsaussagen

5 Schutz des Gastrechners

- Beschränkung der Zugriffsrechte des mobilen Codes
  - ➤ Kontrolle des Zugriffs auf die Systemschnittstelle
  - ➤ Kontrolle des Zugriffs auf Speicherbereiche (→ Sandboxing)
  - ➤ Ausführung in einer abgeschotteten virtuellen Maschine
- Eindeutige Identifikation des Codes
  - ➤ *Authenticode*-Techniken
    - – Hash über Code wird mit privatem Schlüssel des Code-Herstellers signiert → Zertifikat des Herstellers
    - – Echtheit des Hersteller-Zertifikats wird durch Signatur einer Zertifizierungsstelle nachgewiesen (z. B. VeriSign)
  - ➤ Problem: auch zertifizierter Code kann Fehler, Viren, etc. enthalten und damit Bedrohung darstellen
    - – falsches Sicherheitsgefühl beim Anwender!

5 Schutz des Gastrechners (2)

- Proof-Carrying-Code
  - ➤ Gastrechner legt Sicherheitsstrategie fest und veröffentlicht sie Beispiele:
    - – Zugriff nur auf bestimmte Speicherbereiche
    - – keine Pufferüberläufe, weil Feldgrenzen immer geprüft werden
    - – ausschließlich Typ-konforme Zugriffe
  - ➤ Mobiler Code wird vom Erzeuger mit Beweis ausgestattet, dass eine bestimmte Sicherheitsstrategie eingehalten wird
  - ➤ beim Laden des Codes überprüft der Gastrechner den Beweis
    - – spart aufwändige Kontrollen zur Laufzeit

◆ Problem: Erstellung der Beweis sehr aufwändig
bislang nur wenig Unterstützung durch Entwicklungsumgebungen