# Praktikum Anwendungssicherheit Zusammenfassung

# Ein Bild, das Text, Screenshot, Schrift, Zahl enthält. Automatisch generierte Beschreibung1) Grundlagen

## C-Kompilierung

* Pre-proccesor: Auflösen von Header Files, Inklusion in Quelldatei
* Compiling: Umwandlung in Assembler Code (eventuell mit Optimierungen)
* Assembler: Übersetzung in Maschinencode, Library Funktionen (printf) werden nicht aufgelöst
* Linking: Kombination von mehreren Object Files (Quelldatei + Libraries) in eine einzige Executable
  + Dynamisches Linken: Laden von Libraries zur Laufzeit
* Disassembler: Übersetzt Maschinencode in Assembler Code, nur funktionale Gleichheit zum ursprünglichen Assembler Code -> Keine 1:1 Entsprechung von .s auf .o
* Decompiler: Übersetzt Maschinencode in Quellcode, Reverse-Engineering-Technik

## Funktionsaufruf

* Vor Funktionscode:
  + Caller übergibt Daten gemäß Calling Convention, schreibt dann Instruction Pointer (IP) auf Stack (Adresse der ersten Instruktion nach der Funktion) und führt dann Jump-Befehl zur Funktion aus
  + Callee speichert alten Base Pointer (BP) auf Stack, setzt Base Pointer auf aktuellen Stack Pointer
* Nach Funktionscode:
  + Callee stellt alten Base Pointer vom Stack wieder her (Leave), lädt Rücksprungadresse vom Stack

## Calling Conventions

* Methode, mit der Unterprogrammen Daten übergeben werden
* „Regeln“ für Caller und Callee
* Callee muss oft einzelne Register sichern, wenn benutzt
* Abhängig vom Betriebssystem (+Prozessor)
* Wechsel zwischen Conventions durch „Bridge“-Code möglich
* Auf Argumente, die in Registern übergeben werden, kann schneller zugegriffen werden

### cdecl

* Argumente in umgekehrter Reihenfolge auf dem Stack übergeben (erstes Argument steht oben auf dem Stack)
* Caller löscht Daten vom Stack

### System V AMD 64

* Argumente in 6 Registern übergeben, Fallback für mehr Argumente ist Stack

### Microsoft x64

* Argumente in 4 Registern, Fallback ist Stack
* 32 Byte red zone auf dem Stack zwischen Argumenten und IP

### Fastcall

* Callee muss Argumente vom Stack entfernen (mit „ret n“ = Return + Move SP)

# 2) Buffer Overflows

* Schreiben von mehr Daten als vorgesehen in einen Buffer
* Folge: Überschreiben von angrenzenden Speicherbereichen

## Arrays/Strings in C

* int my\_ints[16] erstellt ein Int-Array aus 16 Integers
* Kontinuierliche Speicherregion aus 16 \* 4 = 64 Bytes (-> „Low-level Konstrukt“)
* Undefined Behavior, wenn Array „out of bounds” indexiert wird
* String sind Arrays vom Typ char, die mit einem NULL-Byte enden
* Strlen(): Länge des Strings von Startadresse bis zum NULL-Byte
* Sizeof: Länge des Arrays, in dem der String gespeichert ist (mit NULL-Byte, statisch)

## (Stack) Buffer Overflow Angriffe

* Kontrollfluss Manipulation durch Überschreiben von lokalen Variablen oder Return Adressen
* Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

  Automatisch generierte BeschreibungShellcode Injection
  + Shellcode: Programm (Maschinencode) typischerweise zum Aufrufen einer Shell
  + Shellcode muss oft gewisse Restriktionen einhalten: Kein NULL-Byte, möglichst klein, nur alphanumerisch
  + Rücksprungadresse so überschreiben, dass es auf den Shellcode zeigt (bspw. an den Start des Buffer mit dem Overflow)
  + Mit ASLR ist nicht immer bekannt, wo genau der Buffer liegt. Daher: Nop-Slides in den Buffer schreiben, die nichts tun, nur die letzte Instruktion nach den Slides wird ausgeführt -> Größere Target Area

## Ein Bild, das Text, Schrift, Screenshot, Reihe enthält. Automatisch generierte BeschreibungG1) Stack Canaries

* Compileroption, die standardmäßig an ist
* Random 4-Byte Integer Wert auf dem Stack (bspw. am Start eine Stack Frames)
* Bei Programmstart erstellt und in globaler Variable gespeichert, identisch für alle Funktionen (auch nach fork)
* Funktion prüft beim Return, ob der Canary-Wert überschrieben wurde -> dann Terminierung des Programms
* Alternative: Terminator Canary, bei dem der Canary aus NULL-Terminators besteht (LF) -> verhindert Angriffe mit strcpy(), aber nicht bei gets()
* Lokale Variablen vor dem Canary können trotzdem überschrieben werden (moderne Compiler versuchen aber, Buffer so nah wie möglich am Canary zu platzieren)

## Stack Canary Bypass

* Canary muss mit dem korrekten Wert überschrieben werden
* Information leak: Auslesen des Canary Wertes
* Brute Force: Versuche bei 32 Bit
* Byte-by-Byte Brute Force
  + Voraussetzung: Buffer Overflow mit kontrollierbarer Länge & Programm mit fork() -> Kein Absturz des kompletten Programms bei falschem Wert, nur Absturz vom geforkten Prozess
  + Für jedes Bytes nacheinander alle Werte durchprobieren, prüfen ob Programm noch läuft
  + 8\*256 = 2048 Versuche

## G2) NX-Bit

* Statusbit für Speicherseiten, markiert Seiten als Non-Executable
* Prozessor wirft Exception wenn Code so einer Seite ausgeführt werden soll
* Vom Betriebssystem für bspw. Stack/Heap gesetzt
* Dazu: Code Pages sind nicht schreibbar
* Einige eingebettete Systemen unterstützen das NX-Bit nicht

## Ein Bild, das Text, Diagramm, Screenshot, Zahl enthält. Automatisch generierte BeschreibungBypass: Ret2Libc

* Überschreibe Return-Adresse mit Adresse von existierenden Funktionen (aus Binary, oder aus Libraries -> libc)
* Oft Aufrufen vom system() Syscall mit bin/sh als Parameter -> Shell
* „bin/sh“ steht als String in libc!
* Umgeht NX-Bit, da kein neuer Code eingeschleust wird

## Verallgemeinerung: ROP

* Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

  Automatisch generierte BeschreibungAuch überschreiben der Return-Adresse durch existierende Funktionen (nicht zwingend aus libc)
* ROP-Chain: Sequenz von Adressen von Gadgets, Adressen von Funktionen und Parametern -> Beliebige Funktionen können hintereinander ausgeführt werden
* Je nach Calling Convention müssen Argumente auf Stack/Register übergeben werden
* Überschreiben von Registern durch ROP-Gadgets: CPU-Instruktionen, die im Programm vorhanden sind
* Beispiel: „pop rdi; ret“ überschreibt rdi und springt zur nächsten Instruktion
* Argumente auf dem Stack werden durch pop ret, oder pop pop ret etc. gelöscht
* Voraussetzung: Buffer Overflow + Libc Version kennen

## G3) ASLR

* Feature auf OS-Level
* Gegenmaßnahme für ROP und Ret2Libc
* Normalerweise werden Libraries immer an dieselbe Stelle geladen
* ASLR Randomisiert Ladeadresse von libc und anderen Libraries
* Randomisiert auch Stack und Heap
* Adressen von Funktionen in den Libraries können nicht mehr hardcoded werden
* Ladeadresse von Executable nur randomisiert, wenn sie PIE-kompiliert ist

## ASLR Bypass

* Information Leak, um Ladeadresse von libc zu bestimmen
* Leake Adresse bekannter Funktion, berechne mit Offset (dafür muss libc Version bekannt sein) die Ladeadresse von libc
* Leak von Heap, Stack oder GOT

# 3) Race Conditions

* Bug, erzeugt durch gleichzeitiges Zugreifen mehrerer Prozesse auf geteilte Resource
* Behebbar durch Synchronisation
* Automatische Erkennung schwer, manuelle Analyse erforderlich

## Ein Bild, das Text, Schrift, Screenshot, weiß enthält. Automatisch generierte BeschreibungTime-of-check-Time-of-Use (TOCTOU) Bugs

* Zeitraum zwischen einer Überprüfung (Time-of-check) und Verwendung des Prüfungsergebnisses (Time-of-use)
* Bsp: Angreifer verändert File zu einem symbolischen Link, auf den er eigentlich nicht zugreifen darf, nach dem Access Check -> Genaue zeitliche Abstimmung erforderlich
* Kann auch bei Single-Threading auftreten

# 4) Global Offset Table

* Binaries sind oft dynamisch gelinkt -> Nicht der komplette genutzte Code ist in Binary, es gibt Referenzen auf genutzte Funktionen, werden aus der Systembibliothek geladen
* Binary ist so deutlich kleiner und flexibler (Austauschbarkeit der Systemfunktion)
* Referenz kann nicht hardcoded sein, da sonst bspw. libc immer an selbe Adresse geladen werden würde

### Ein Bild, das Text, Screenshot, Schrift, Diagramm enthält. Automatisch generierte BeschreibungLazy Binding

* Dynamic Linker befüllt Global Offset Table: Adresse von genutzten Funktionen zur Laufzeit
* Compiler schreibt für jede Library Funktion Dummy-Code in PLT Section der Binary
* Dieser springt in die GOT, die initial wieder auf den Dummy-Code verweist
* Dummy springt zum Dynamic Linker und löst Adresse der Funktion auf, diese wird in GOT eingetragen
* Folge: GOT ist beschreibbar

### Angriffe

* Arbitrary Read: Durch Lesen von Einträgen in der GOT, Adressen von Funktionen bestimmen -> nützlich für ROP-Gadgets + Umgehen von ASLR
* Arbitrary Write: Überschreiben des GOT-Eintrags einer Funktion -> Kontrollfluss Manipulation

### G4) Relocation Read-only (RELRO)

* Compileroption, verhindert GOT-Angriffe
* Linker beschreibt GOT bei Programmstart, danach wird GOT read-only
* Nachteil: Auflösen aller Adressen genutzter Funktionen braucht Zeit

### G5) Position Independent Executables (PIE)

* Compile-time Feature
* Generiert Executables, die an beliebige Adressen geladen werden kann
* Adressen von Funktionen in der Binary können nicht mehr hardcoded werden
* GOT ist nicht mehr immer an derselben Stelle (Offset ist aber immer gleich)
* Genutzt vor allem für Shared Libraries, damit sie immer genutzt werden können
* Anmerkung: Leak von einer Adresse, von der man den fixen Offset zur Binary Base kennt, umgeht PIE

# Ein Bild, das Schrift, Text, Reihe, weiß enthält. Automatisch generierte Beschreibung5) Integer Bugs

* Ein Bild, das Text, Schrift, Screenshot, Typografie enthält.

  Automatisch generierte BeschreibungInteger haben feste Länge in c:
* Wenn das Ergebnis einer Operation nicht darstellbar (zu groß, zu klein) ist, resultiert dies in einem Overflow
* Bsp.: Overflow bei der Berechnung der Größe für einen Buffer -> Heap Overflow Exploit

## Integer Vergleiche

* Datentypen kleiner int werden zu int gecastet (char, short)
* Bei Datentypen größer gleich int wird kleinerer Datentyp zum größeren (signed) Datentyp gecastet
* Bei signed/unsigned mismatch wird zu unsigned gecastet
* Folge: Negative ints können größer als unsigned ints sein
* Beispiel: Prüfung, ob Abheben von Geld erlaubt. Kontostand unsigned, Betrag ist signed -> Unerlaubtes abheben von negativem Kontostand möglich

# 6) Use-after-free

* Malloc allokiert fixen Speicherbereich auf dem Heap, returned Pointer
* Free() gibt Speicherbereich wieder frei. MMU wird benachrichtigt, dass Speicherbereich nicht mehr benötigt wird und re-allokiert werden darf
* Problem: Daten bleiben erhalten, Pointer in den Speicherbereich auch (Dangling Pointer)
* Information Leak, wenn derselbe Speicherbereich neu benutzt wird
* Ausnutzbar, wenn Daten Pointer enthalten
* Gegenmaßnahme: Pointer Nullen

# 7) Symbolic execution, angr

* Symbolischer Ausdruck: Ausdruck der einen beliebigen Wert repräsentiert
* Symbolische Ausführung: Operationen werden auf symbolischen Ausdrücken angewandt
* Symbolische Analyse: Symbolische Ausführung -> Änderungen auf Variablen tracken (Constraints +Formeln aufstellen) -> Automatische Lösung der Constraints + Formeln
* Branching: Aktuelle Constraints kopieren, für Branch Taken/Not Taken Constraint hinzufügen, Symbolische Ausführung auf beiden Branches weiterführen
* Anzahl Branches können explodieren (bei Loops)

## Angr

* Framework für Symbolic Execution
* Jede Variable ist ein Bitvektor, ob symbolisch oder konkret
* SimProcedures: Re-implementierte Library Funktionen mit weniger Komplexität -> erleichtert Symbolische Ausführung

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung