Technische Universität Ilmenau 2. Mai 2024

Vorlesung

Schutz von Kommunikationsinfrastrukturen

Prof. Dr.-Ing. Günter Schäfer

NeXTeX

Vorlesungsmitschrift von
Adrian Schollmeyer

Inhaltsverzeichnis

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Threats and Security Goals . . . 3
- 1.3 Security Analysis of Layered Protocol Architectures . . . 4
- 1.4 Security Analysis of Communication Infrastructure . . . 4
- 1.5 Towards Systematic Threat Analysis . . . 5
- 1.6 System Security Engineering Process . . . 5
- 1.7 A High Level Model for Internet-based IT-Infrastructure . . . 6
- 1.8 Countering Attacks . . . 6
  - 1.8.1 Principle Classes of Action . . . 6
  - 1.8.2 Safeguards Against Information Security . . . 7
- 1.9 Terminology . . . 7
- 1.10 Security Services . . . 8

2 Security Aware System Design and Implementation **9**
- 2.1 Problems of Practical System Security . . . 9
- 2.2 Origin of Attacks . . . 9
- 2.3 Unix Access Control . . . 10
- 2.4 Buffer Overflows . . . 10
  - 2.4.1 Stack Smashing . . . 11
  - 2.4.2 Angriffe auf dem Heap . . . 12
  - 2.4.3 Exploits mit Buffer Overflow . . . 12
  - 2.4.4 Verhinderung von Buffer-Overflow-Angriffen . . . 13
- 2.5 Format String Attacks . . . 14
- 2.6 Race Conditions . . . 14
- 2.7 SQL Injections und Cross-Site-Scripting (XSS) . . . 15
- 2.8 Malware . . . 15
- 2.9 Exploiting Trust . . . 16
  - 2.9.1 Reflections on Trusting Trust . . . 16
  - 2.9.2 Security of Open Source Software . . . 17
- 2.10 Trating Input in Secure Programs . . . 17
- 2.11 Countermeasures . . . 18

Stichwortverzeichnis **19**

1 Introduction

1.1 Motivation

Moderne Infrastruktur ist zunehmend abhängig von funktionierenden Kommunikationsnetzen. Verfügbarkeit und Sicherheit beim Betrieb von solchen Kommunikationsnetzen und daran angeschlossenen Teilnehmern wird immer wichtiger. Dabei ist insbesondere nicht nur die Verfügbarkeit wichtig, sondern auch die Sicherheit, da Angriffe auch die Verfügbarkeit und Stabilität des Systems beeinträchtigen können.

1.2 Threats and Security Goals

Eine **Bedrohung** ist ein mögliches Ereignis oder eine Sequenz von Ereignissen, die zu einer Verletzung von mindestens einem Sicherheitsziel führen kann. Im Unterschied zu einem **Angriff**, der tatsächlich passiert, liegt eine Bedrohung bereits vor, wenn es nur möglich ist, einen Angriff zu fahren. Beim Schutz von Kommunikationsinfrastruktur sollte nicht auf Angriffe reagiert, sondern Bedrohungen im Vorfeld vermieden werden.

Sicherheitsziele können abhängig von verschiedenen Faktoren definiert werden, beispielsweise anhand der Anwendungsumgebung:

Telekommunikationsanbieter Privatsphäre der Nutzer, Zugriffsbeschränkung für administrative Funktionen, Ausfallsicherheit

Firmen-/Privatnetze Privatsphäre, Verschluss von Firmengeheimnissen, Authentizität von Nachrichten, Ausfallsicherheit

Alle Netzwerke Schutz vor Eindringlingen von außen

Technisch definiert wird sich auf fünf Ziele beschränkt:

Vertraulichkeit (von Daten, Identitäten, etc.)

Integrität (Sicherstellung, dass Daten auf dem Kommunikationsweg nicht verändert worden sein können; benötigt auch Authentizität der Nachricht)

Accountability (Nachweisbarkeit, welche Entität für welches Ereignis verantwortlich ist)

Kontrollierter Zugriff (Durchsetzung von Regeln, welche Entitäten welche Rechte im System haben)

Verfügbarkeit (Systeme sollten verfügbar sein, korrekt funktionieren und in ihrer Leistungsfähigkeit nicht beeinträchtigt werden)

Angriffe lassen sich auch in Kategorien einteilen:

Masquerade (der Angreifer kann sich als jemand anderes ausgeben)

Eavesdropping (Abhörung von übermittelten Nachrichten)

Authorization Violation (ein Angreifer macht etwas, wofür er eigentlich nicht die nötigen Berechtigungen haben sollte)

Verlust oder Veränderung von Daten

Repudiation (Abstreiten von Kommunikationsvorgängen, erfordert meist anwendungsspezifisches Wissen)

Fälschung von Informationen (Angreifer erzeugt neue Informationen im Namen einer andere Entität)

Sabotage

Verschiedene Angriffe bedrohen Sicherheitsziele in verschiedener Weise. Es ist jedoch immer möglich, dass die Verletzung eines Sicherheitsziels dazu führen kann, dass andere, gefährlichere Angriffe gefahren werden können (bspw. kann mit Eavesdropping ein Root-Passwort ausgelesen werden, wodurch Maskerade möglich ist).

1.3 Security Analysis of Layered Protocol Architectures

Die erste Frage bei der Analyse ist, an welchen Stellen ein Angreifer welche Angriffe fahren kann. Je nach Ort kann ein Angreifer dann an verschiedenen Schichten der Protokollarchitektur angreifen. Je nach Schicht sind wiederum verschieden mächtige Angriffe möglich.

Auf der Nachrichtenebene (sprich, wo einzelne Pakete der eingesetzten Protokolle verschickt werden), lässt sich analysieren, welche Angriffe auf die **PDUs** (Protocol Data Units) gefahren werden können. Für einen erfolgreichen Angriff darf es keine Nebeneffekt auf andere Verbindungen, idealerweise auch nicht auf die betroffene Verbindung geben. Ansonsten besteht das Risiko, dass der Angriff scheitert oder sogar vom Opfer erkannt wird.

1.4 Security Analysis of Communication Infrastructure

Neben den Angriffen auf die Informationsübertragung sind auch Angriffe auf die Systeme, die Teil des Kommunikationsnetzes sind, wichtig, u. A.

- Endsysteme
- Router
- wichtige Infrastrukturdienste (z. B. DNS, E-Mail, Webserver, etc.)

Dabei erweitert sich die Analyse deutlich und wird viel anwendungs- und systemspezifischer.

1.5 Towards Systematic Threat Analysis

Die Erstellung beliebiger Listen möglicher Angriffe ist keine besonders zielführende Methode. Dabei ist es nur schwer möglich, die Vollständigkeit der identifizierten Angriffe zu zeigen.

Ein **Bedrohungsbaum** beschreibt Bedorhungen auf verschiedenen Eben mit verschiedenen Leveln an Abstraktion. An den obersten Knoten werden die generellen Ziele definiert, während beim Absteigen in den Baum immer konkretere Angriffsszenarien und Bedrohungen notiert werden. Irgendwann erhält man Blattknoten, die sehr detaillierte Bedrohungen beschreiben und die in einer (weniger beliebige) Bedrohungsliste aufgenommen werden können.

Bei der Erstellung von Bedrohungsbäumen sollte darauf geachtet werden, auf jeder Analyse vollständig zu arbeiten, bspw. in dem binäre Entscheidungen in Knoten repräsentiert werden (sodass alle möglichen Fälle abgedeckt werden). Weiterhin können Knoten auch in verschiedenen logischen Beziehungen (und, oder) stehen, je nachdem ob ein Angreifer bspw. nur eine von mehreren oder auch mehrere Hürden auf einmal überwinden können muss, um einen Angriff durchzuführen (vergleiche: Stellen, um in ein Haus einzubrechen vs. Sicherheitsmaßnahmen wie Zäune, Wachhunde, etc.). Sind nun Angriffsszenarien bekannt, kann der Aufwand (die Kosten), Angriffe durchzuführen, abgeschätzt werden, um zu analysieren, welche Angriffe am wahrscheinlichsten sind. Für oder-verknüpfte Knoten muss dabei das Minimum genommen werden (angenommen der Angreifer kennt den Aufwand, wird er den geringsten Aufwand wählen), während bei und-verknüpften Knoten das Maximum gewählt werden muss (er muss mindestens diesen Aufwand betreiben, um den Angriff durchzuführen).

Daneben stellt sich auch die Frage, wie viele Personen zu einem Bestimmten Angriff bereit sind. Das ist vor allem eine Frage dessen, was geschützt werden soll (vgl. Staatsgeheimnisse vs. eine Privatwohnung) und wer als Angreifer infrage kommt (und welche Möglichkeiten die Angreifer dann potentiell haben). Dies lässt sich im Bedrohungsbaum nicht direkt modellieren. Stattdessen muss die resultierende Angriffsliste auf Kosten und Gewinn analysiert werden und die Frage ist, welche Angreifer dann noch zu welchen Angriffen bereit sind.

1.6 System Security Engineering Process

Dies erlaubt ein etwas systematischeres Vorgehen:

- Spezifikation der Systemarchitektur
- Identifikation von Bedrohungen, Schwachstellen und Angriffstechniken
- Abschätzung von Risiken (zusätzliche Attribute am Bedrohungsbaum)
- Priorisierung von Schwachstellen
- Identifikation und Installation von Sicherungsmaßnahmen (für Schwachstellen mit hoher Priorität)

- Neuanalyse/Iteration der o. g. Schritte

Es kann dabei prinzipiell auch möglich sein, dass der Angreifer damit rechnet, dass bestimmte Sicherungsmaßnahmen ergriffen werden. Es besteht also grundsätzlich die Gefahr, dass durch die Installation von solche Maßnahmen auch neue Bedrohungen entstehen, deren Ausnutzung potentiell schon vom Angreifer geplant sind. Es kann also auch Sicherungsmaßnahmen geben, die tatsächlich kontraproduktiv sind.

1.7 A High Level Model for Internet-based IT-Infrastructure

Die Hauptunterscheidung ist zwischen private Netzen (d. h. Zeug, was keine Dienste anbietet, sondern mit Diensten spricht; bspw. Heimnetze, Sensornetze, etc.), Support-Infrastruktur (Transportnetze wie das Internet) und ISP-Netzwerke (Bereitstellung von Diensten, Cloud-Hosts, Rechenzentren, Mobilkommunikationsnetze).

Grundsätzlich dürfen physikalische Bedrohungen (bspw. Zerstörung eines Rechenzentrums) nicht missachtet werden und physische Redundanz ist immer wichtig, um Verfügbarkeit zu garantieren. Dennoch sind solche Bedrohungen für diese Vorlesung weniger relevant und werden nicht wirklich behandelt.

Link-basierte Bedrohungen auf physischer Ebene sind zwar theoretisch möglich, lohnen sich aber meist weniger als Angriffe auf dem Data Link Layer. Die Untersuchung und Behandlung solcher Bedrohungen ist Teil der Veranstaltung „Network Security“.

Beim Network Layer werden hier auch Dienste hinzugezogen, die nicht direkt auf dem Network Layer laufen, aber für den Betrieb des Netzes an sich notwendig sind (bspw. DNS). Die Anwendungsschicht wird in diesem Bedrohungsbaum vereinfacht und ist nicht vollständig. Dieser Bereich ist auch sehr stark davon abhängig, welche Anwendungen im Netz betrieben werden.

1.8 Countering Attacks

1.8.1 Principle Classes of Action

Prävention Umfasst alle Maßnahmen, um Angreifer davon abzuhalten, erfolgreiche Angriffe durchzuführen (bspw. Verschlüsselung, Signaturen, Firewalls) Dies muss grundsätzlich passieren, *bevor* der Angriff stattfindet.

Erkennung Umfasst alle Maßnahmen, um laufende oder vergangene Angriffe zu erkennen. Dazu gehören Audit Trails, Traffic Monitoring (idealerweise on-the-fly), etc. Wird ein laufender Angriff erkannt, sollten Maßnahmen ergriffen werden, um den Angriff entweder zu unterbinden oder zunächst zu entscheiden, ob der Angriff zu echten Schäden führen oder der Angreifer ermittelt werden kann.

Eine Variante der Erkennung von potentiellen Angreifern ist der Betrieb eines **Honeypots**. Diese Systeme sind am Netz erreichbar mit dem Ziel, dass Angreifer diese angreifen. Auf solchen Systemen sollten keine echten Dienste laufen, sodass der Zugriff auf das System direkt als Angriff erkannt werden kann. Wichtig ist

hierbei jedoch auch wieder, dass der Honeypot selbst kein Einfallstor in echte Kommunikationsinfrastruktur darstellt.

Reaktion Umfasst alle Maßnahmen, die in Reaktion auf vergangene oder laufende Angriffe unternommen wurden.

1.8.2 Safeguards Against Information Security

- Physical Security (Beschränkung von physischem Zugang zu Servern u. Ä.)
- Personnel Security (Überprüfung von Personal, welches Zugriff zu Servern u. Ä. hat)
- Administrative Security (Ausbildung von Personal, Einbringung von Fremdsoftware, Prozeduren und Workflows zur Untersuchung von Sicherheitsvorfällen, Review von Audit Trails etc.)
- Emanations Security (Abstrahlung von Geräten, etc.)
- Media Security (Kontrolle darüber, wie sensible Informationen (sicher) reproduziert und zerstört werden können, Sicherhung der Speichermedien, Virenscans, etc.)
- Lifecycle Controls (Kontrolle des Softwareentwicklungsprozesses, Programmierstandards, etc.)
- Computer/System Security (Schutz der verarbeiteten Informationen und verarbeitenden Geräte)
- Kommunikationssicherheit (Schutz der Informationen auf dem Transportweg, Schutz der Kommunikationsinfrastruktur)

1.9 Terminology

Security Service Abstrakter Dienst, welcher eine bestimmte Sicherheitseigenschaft garantieren soll. Kann mithilfe von Kryptographie, Protokollen, aber auch konventionellen Mitteln (z. B. Ablegen des Datenträgers in einem Tresor) garantiert werden. Meist werden mehrere Dienste miteinander kombiniert.

Cryptographic Algorithm

cryptographic algorithm

Cryptographic Protocol

cryptographic protocol Regeln, wer wann welche Berechnungen durchführen und welche Nachrichten verschiken muss, um bestimmte Sicherheitsziele zu erreichen (z. B. für Authentisierung, Schlüsselaustausch, etc.).

1.10 Security Services

Authentisierung

authentication

Integrität

integrity

Vertraulichkeit

confidentiality

Zugriffskontrolle

access control

Nichtabstreitbarkeit

non-repudiability Es soll sichergestellt werden, dass kein Kommunikationspartner später abstreiten kann, an einer Kommunikation teilgenommen zu haben. Dies lässt sich eigentlich nur durch Zertifizierung durch eine unabhängige dritte Stelle sicherstellen.

2 Security Aware System Design and Implementation

2.1 Problems of Practical System Security

Es ist unmöglich, die Sicherheit eines moderat komplexen Systems zu zeigen. Allgemein ist es schwierig, die Abwesenheit einer Eigenschaft (hier: Bedrohung) zu zeigen. Daher ist es wichtig, möglichst einfache Systeme einzusetzen, wenn Sicherheit garantiert werden soll. Zusätzlich müssen die Quellen beherrscht werden, von denen die Systeme kommen. Meist ist dabei Software das Problem, die durch schlechtes Softwaredesign oder schlechte Implementierung Sicherheitsprobleme haben. Systemadministratoren müssen dann häufig Sicherheitspatches einpflegen, was zu zusätzlicher Last (und potentieller Nachlässigkeit) bei der Wartung führt, wodurch wieder Fehler gemacht werden können.

Weitere Probleme sind der Einsatz von Low-Level-Programmiersprachen, die einfache Angriffe ermöglichen (bspw. Buffer Overflow), schlechte Standardkonfigurationen von Software, Erweiterbarkeit von Software durch Updates und dynamisch ladbare Erweiterungen (e. g. Plugins) und der Mangel an Diversität in beliebten Rechenumgebungen (bspw. Windows auf PCs, Linux auf Servern, Cisco IOS für Router, etc.). Durch die immer steigende Geschwindigkeit bei der Softwareentwicklung kommt es zu immer kürzeren Entwicklungszyklen mit schlechter Spezifikation von Anforderugen, sofern diese überhaupt existieren. Gerade letztes führt dazu, dass Sicherheitsaspekte nicht oder nur unzureichend beachtet werden.

2.2 Origin of Attacks

Zunächst werden Angriffe nach ihrem Ursprung unterschieden. Entfernte Angriffe werden typischerweise unter Ausnutzung geklauter/schwacher Passwörter oder offener Schwachstellen in der Software gefahren, um Zugriff auf ein System zu erhalten. Lokale Angriffe arbeiten mit einem bestehenden Zugang und haben zum Ziel, die eigenen Berechtigungen auszuweiten. Unter Linux ist dies beispielsweise ein interessantes Ziel, da normale Service-Benutzer (also eigens für den Betrieb eines Dienstes eingerichtete Benutzerkonten) meist stark eingeschränkte Berechtigungen haben. Echte Benutzeraccounts hingegen sind noch interesssanter als Ziel, da diese zwar nicht durchgehend aktiv in Benutzung sind, aber dafür über Tools wie sudo Root-Zugriff haben.

Am interessantestn sind von den beiden Arten die Remote-Angriffe, da der Angreifer in aller Regel keinen direkten Zugang zum System hat und dies in so einem Fall also immer eine Voraussetzung für einen lokalen Angriff ist.

2.3 Unix Access Control

Im Server-Bereich sind unixoide Systeme wie Linux sehr prävalent. Auch Router-Betriebssysteme haben häufig Linux als Grundlage. Dies hat auch zur Folge, dass die meisten lokalen Exploits von der Ausnutzung unixoider Zugriffskontrollen abhängen. Die Zugriffskontrolle unter Unix basiert auf Benutzern, die in Gruppen sein können, sowie Berechtigungen auf Dateien, die für Besitzer der Datei, Gruppe der Datei und beliebige andere Benutzer festgelegt werden können. Hinzu kommt der root-Benutzer (UID 0), der immer alle Berechtigungen hat und Berechtigungen nach Belieben ändern kann. Bei der Rechteprüfung wird für einen Prozess die effektive Benutzer-ID, effektive Gruppen-ID und die Berechtigungseinstellungen der Datei miteinander verglichen, um zu entscheiden, ob eine bestimmte Aktion (Lesen, Schreiben, Ausführen) zulässig ist. Zusätzlich können Dateien mit dem *setuid*-Bit versehen, mit dem bei Ausführung der Datei der Prozess mit dem Besitzer der Datei als effektive User-ID ausführt wird. Ähnlich kann mit *setgid* auch die effektive Gruppen-ID verändert werden.

Die Benutzung potentiell gefährlicher Operationen wie setuid oder setgid sollten nur sparsam eingesetzt werden. Grundsätzlich ist es keine gute Idee, mit mehr Rechten zu arbeiten als nötig, da nicht nur die Gefahr besteht, dass (versehentlich) ausgeführter Schadcode zu Sicherheitsproblemem führt, sondern schon allein menschliche Fehler (z.B. Tippfehler) können zu Schäden am System führen. Prozesse, die mit hohen Rechten (bspw. durch setuid) laufen, kann eine Ausnutzung von Schwachstellen/Programmierfehlern benutzt werden, um (Voll-)Zugriff auf das System zu erhalten. Solche Zugriffe können möglicherweise erlauben, zusätzliche Zugänge z. B. durch Manipulation von `/etc/passwd` zu erhalten.

Ein Workaround, um hohe Berechtigungen für notwendige Operationen (bspw. Binden von Ports < 1024) zu machen, ohne dauerhaft mit hohen Privilegien zu laufen, besteht darin, alles, was hohe Berechtigungen benötigt, zum Start des Dienstes auszuführen (das ist meist ausrechend) und danach die Berechtigungen auf die eines weniger privilegierten Benutzers zu reduzieren, sodass spätere Angriffe weniger Schaden anrichten können.

2.4 Buffer Overflows

Diese Angriffsart betrifft vor allem low-level Programmiersprachen wie C oder C++, die manuelle Speicherverwaltung benutzen. Speicher kann dort auf dem Stack oder auf dem Heap allokiert werden. Auf dem Stack werden dabei die lokalen Variablen, Rückgabewerte, Sprungadressen und Funktionsargumente gespeichert. Dieser Bereich wird vom Programm automatisch verwaltet. Auf dem Heap hingegen wird Speicher grundsätzlich manuell, z. B. durch `malloc` und `free` angefordert und freigegeben.

Buffer-Overflow-Angriffe beruhen darauf, über Speichergrenzen hinweg zu schreiben. In Sprachen wie C und C++ wird dabei ausgenutzt, dass es keine automatisch Prüfung von Speichergrenzen bspw. von Arrays gibt. So können Programmierfehler dazu führen, dass in Speicherbereiche geschrieben wird, die gar nicht zu dem Objekt gehören, was eigentlich geschrieben werden sollte. Dies kann von Angreifern ausgenutzt

werden, um bspw. Sprungadressen zu überschreiben und damit andere Funktionen aufzurufen. Ein übliches Problem dabei ist bspw., wenn in einen Puffer Daten geschrieben werden sollen, die länger als der Puffer selbst sind. Wurde dabei dann kein ordentliches Bounds-Checking implementiert, kann in Speicherbereiche hinter dem Puffer geschrieben werden.

Die Konsequenzen eines Buffer Overflows sind nicht fest definiert. Je nachdem, welcher Speicher sich dahinter befindet, verhält sich das Programm komisch (Daten wurden falsch geschrieben und jetzt befindet sich das Programm in einem falschen Zustand), stürzt ab (z. B. bei Zugriff auf einen nicht allokierten Speicherbereich) oder verhält sich wie zuvor (z. B. wenn in Padding-Bereiche geschrieben wurde). Es hängt also immer davon ab, wie viele Daten außerhalb der Grenzen geschrieben wurden und welche Speicherbereiche sich dort befinden.

Ein Angreifer kann je nach Möglichkeiten Buffer Overflows für verschiedene Zwecke nutzen:

- Manipulation von Rücksprungadressen, um anderen (z. B. eingeschleusten) Code auszuführen
- Eskalation von Privilegien, indem bspw. Programme mit setuid ausgenutzt werden
- etc.

Besonders interessant sind dabei immer Angriffe auf dem Stack, da nur hier Rücksprungadressen manipuliert werden können. Beliebt dabei sind viele Funktionen der C-Standardbibliothek wie bspw. `gets`, `strcpy`, `scanf`, die kein Bounds-Checking unterstützen. Aber auch Funktionen mit Bounds-Checking (z. B. `strncpy`) können ausgenutzt werden, wenn dort Strings ohne Nullbyte kopiert werden, ohne nach der Operation ein Nullbyte ans Ende zu setzen.

2.4.1 Stack Smashing

Ein wichtiges Problem bei Buffer Overflows ist das Speicherlayout, vor allem das des Stacks. Hier legt der Compiler für jede Funktion bestimmte Werte auf den Stack, u. A. Funktionsparameter, Rücksprungadresse und der sogenannte **Stack Frame**, der den Beginn des Speicherbereichs markiert, ab dem die Variablen gespeichert werden, die auf dem Stack gespeichert sind. Ein übliches Layout eines Stack Frames:

- Funktionsargumente…
- Rücksprungadresse
- Alter wert ebp (Mitte des Stack Frames, ab wo die Variablen beginnen)
- Stack-Variablen (Variablen in der Funktion, die nicht auf dem Heap allokiert sind)

Der Pointer ebp wird dabei auf die Anfangsadresse der „Mitte“ des Stack Frames gesetzt, also da, wo die erste Variable der Funktion steht. Ab hier bis zum Ende des Stacks liegen

alle lokalen Variablen, die die Funktion benutzt. Problem ist jedoch, dass der Stack von oben nach unten aufgebaut wird. Die Rücksprungadresse wird also in einer höheren Adresse auf dem Stack gespeichert als die Variablen auf dem Stack. Dennoch werden bspw. Arrays so auf dem Stack gespeichert, dass das erste Element (Offset 0) an der niedrigsten Adresse des Puffers (also ganz unten im Stack) liegt, während das letzte Element an der höchsten Adresse des Puffers (also ganz oben im Stack, z. B. kurz vor der Rücksprungadresse) liegt. Wenn man also als Stack-Variable einen Puffer stehen hat und zu weit schreibt, kommt man schnell in die Situation, bei der die Rücksprungadresse überschrieben werden kann. Das gezielte Überschreiben der Rücksprungadresse durch einen Buffer Overflow wird auch als **Stack Smashing** bezeichnet.

2.4.2 Angriffe auf dem Heap

Das Speicherlayout auf dem Heap sind stark davon abhängig, wann wie viel Speicher allokiert wurde und wann welche anderen Prozesse speicher allokiert haben. Hier ist es sehr schwer vorherzusagen, welche Speicherbereiche wo stehen. Durch dynamisches Linken ist es auch schwer vorhersehbar, welche Version einer Bibliothek vom Programm benutzt wird, was das Vorhersagen des Speicherlayouts weiter erschwert.

Dennoch ist es möglich, potentiell kritische Variablen zu finden und auszunutzen. Zudem muss dann aber auch noch für einen Buffer-Overflow-Angriff eine Möglichkeit gefunden werden, einen Puffer so zu überschreiben, dass solch eine Variable manipuliert werden kann.

Allgemein sind Stack Overflows einfacher durchzuführen als Heap Overflows.

2.4.3 Exploits mit Buffer Overflow

Ein mögliches Ziel ist das Erlangen einer Shell, damit der Angreifer beliebig Befehle auf dem System ausführen kann. Üblicherweise wird der boshafte Code (z. B. Start einer Shell) vom Angreifer kompiliert und dann versucht, diesen Code irgendwo in den Speicher zu schreiben, um ihn Anschließend durch Stack Smashing ausführen zu können. Beim Schreiben von solchem Exploit-Code muss häufig darauf geachtet werden, dass der Code keine Null-Bytes enthält, da diese bei den Funktionen, die Strings verarbeiten, häufig als Ende des Strings angesehen werden. Stattdessen werden dabei dann Tricks wie XOR angewendet, die Nullbytes erzeugen können, ohne dass Nullbytes in den Programmcode geschrieben werden müssen. Dies erfordert jedoch manchmal auch Anpassung des kompilierten Schadcodes.

Manchmal ist es auch schwierig, zu genau der richten Adresse zu springen. Eine Technik ist dabei, viele NOPs an vor den Schadcode zu schreiben, sodass es nur irgendwie möglich gemacht werden muss, an eine der NOP-Instruktionen zu springen. Dies wird als **NOP-Slide** bezeichnet. Solche NOP Slides können bspw. durch Heap Spraying (Allokation von ganz viel Speicher gefüllt mit NOPs) erstellt werden.

2.4.4 Verhinderung von Buffer-Overflow-Angriffen

Es ist quasi unmöglich, vollkommen fehlerfreien und nicht angreifbaren Code zu schreiben. Programmierer dazu zu motivieren, besonders vorsichtig beim Programmieren zu sein, ist nur bedingt möglich. Stattdessen werden Methoden diskutiert, Fehler technisch zu verhindern oder besser zu erkennen. Dazu gibt es verschiedene Ansätze.

Der **Stackguard** Ansatz fügt eine gewisse Menge zufälliger Daten (**canary**) an das Ende von Stack-allozierten Daten. Später wird dann geprüft, ob dieser Wert immer noch dort steht. Wenn nicht, dann scheint ein Buffer Overflow aufgetreten zu sein. Dieser Ansatz ist allerdings nur bedingt nützlich, da Variablen im Stack weiter überschrieben werden können (aber nicht mehr gesprungen werden kann). Heap Overflows sind damit allerdings weiter möglich. Außerdem hat das Problem leichte Performanceeinbußen.

Memory Integrity Checking mit Tools wie Valgrind [7] oder LLVM Address Sanitizer [1] kann während der Entwicklung Speicherprobleme aufzeigen. Allerdings sind diese sehr langsam und werden daher in Produktiv-Builds nicht eingesetzt. Daher können Angreifer Buffer Overflows, die nicht bei der Entwicklung erkannt wurden, weiterhin ausnutzen. Außerdem können solche Tools auch nur beschränkt gegen Probleme schützen, die in Drittbibliotheken auftreten.

Eine weitere Möglichkeit besteht darin, die Speicherseiten in ausführbare und nicht ausführbare Seiten zu unterscheiden. Damit ist es möglich, zu verhindern, dass Code auf Seiten ausgeführt wird. Wird dann der Stack als nicht ausführbar markiert, dann kann dort eingeschleuster Programmcode nicht ausgeführt werden. Stattdessen würde das Programm abstürzen. Leider unterstützen nicht alle Programme (bspw. solche mit JIT Compilern) dies nicht. Es könnte allerdings weiter möglich sein, geschickt an ausführbare Stellen zu springen, die die gewünschten Operationen ausführen. Man ist zwar weiterhin stark in den Operationen eingeschränkt, die man ausführen kann, aber diese können möglicherweise gut kombiniert werden, um Schadcode zu erstellen. Diese Art der Programmierung heißt **Return-Oriented Programming** (ROP).

Eine Gegenmaßnahme zu ROP ist **Address Space Layout Randomization** (ASLR). Ziel hierbei ist, es dem Angreifer schwer zu machen, die richtigen Sprungadressen zu erraten. Dazu werden Bibliotheken beim Laden an zufällige, aber aufeinanderfolgende Adressen geladen. Dadurch können die Adressen für ROP nicht mehr geraten werden. Dies erfordert jedoch Unterstützung vom Betriebssystem und Programm. Wirklich effektiv ist dies auch nur, wenn Heap und Stack als nicht ausführbar markiert sind, da sonst Heap Spraying benutzt werden kann. Außerdem erfordert dies einen großen Adressraum (also z. B. reichen 32 Bit nicht), um genug Entropie haben (damit die Trefferwahrscheinlichkeit ausreichend klein ist). Diese Maßnahme führt zu leichten Performanceeinbußen, da Sprünge immer indirekt über eine Sprungtabelle laufen.

Dies sind alles nur Gegenmaßnahmen, die das Symptom, jedoch nicht die Ursache bekämpfen. Das grundsätzliche Problem, dass Buffer Overflows auftreten, sollte besser von Anfang an verhindert werden können, bspw. durch Wahl einer entsprechenden Programmiersprache, die dies verhindert.

2.5 Format String Attacks

Format-Strings werden in diversen Programmiersprachen (z. B. C) benutzt, um einen String auszugeben oder zu erzeugen, der aus einer Vorlage (dem **Format-String**) und darin an geeigneten Stellen eingesetzten variablen Werten besteht. Im Format-String werden Platzhalter für Variablen gesetzt (z. B. `%s` im String `Hallo, ich bin %s!` kann später durch einen beliebigen anderen String ersetzt werden). Ein Beispiel dafür ist die Funktion `printf` [3] aus der C-Standardbibliothek. Diese arbeitet so, dass manche C-Funktionen beliebig viele Parameter übernehmen kann. Diese sucht sich aus dem Stack an den richten Positionen die Parameter der Funktion raus. Allerdings gibt es keine Sprachkonstrukte, die sicherstellen, dass an den richtigen Stellen gesucht und die richtigen Datentypen benutzt werden.

Gefährlich werden solche Funktionen, wenn ein vom Benutzer angegebener String über eine solche Funktion ausgegeben werden soll. Hierbei können Benutzer Format-Strings in den auszugebenden String einfügen, wodurch die Funktion versucht, weitere Daten aus dem Stack auszugeben. Dies ermöglicht das Auslesen des genauen Stack-Inhalts durch aneinanderreihen von Format-Platzhaltern. Weitere Möglichkeiten ergeben sich dadurch, dass es Platzhalter gibt, um auch Variablen mit Informationen über den Format-String (bspw. die Länge des erzeugten Strings) zu befüllen. So ist es möglich, durch manipulierte Format-Strings auch gezielt Werte in den Speicher zu schreiben. Damit lässt sich beispielsweise auch die Return-Adresse manipulieren. Natürlich muss dafür auch ein Format-String beliebig lang gemacht werden können. Dazu können Format-Optionen genutzt werden, um bspw. die Stelligkeit oder Präzision einer Zahl festzulegen. Allerdings erfordert das Zählen der Stringlänge auch, dass der gesamte String erzeugt wird. Damit ist man auch durch den verfügbaren Speicher begrenzt. Dennoch gibt es auch dafür geschicktere Angriffe, die jeweils nur Teilworte schreiben und so mit weniger Speicherverbrauch beliebige Sprungadressen festlegen können.

Format-Strings sind allgemein viel gefährlicher als Buffer-Overflow-Attacken. Es ist möglich, präzise bestimmte Speicherbereiche zu überschreiben (und so bspw. einen Canary nicht zu modifizieren) und auch Daten gezielt auszulesen. Außerdem fuktionieren sie selbst mit Bounds-Checks. Dennoch lassen sie sich durch Verwendung anderer Konstrukte (z. B. C++ `std::cout`) gänzlich vermieden werden.

2.6 Race Conditions

Allgemein treten **Race Conditions** auf, wenn eine Annahme vor Ausführung von Code geprüft wird, es aber zwischen der Prüfung und der Ausführung möglich ist, die geprüfte Bedingung zu verändern. Dadurch ist es möglich, dass der Code anschließend auf falschen Annahmen ausgeführt wird und sich dann möglicherweise anders verhält. Diese Art von Angriffen erfordern, dass mehrere Threads auf dem System verfügbar sind (was jedoch auf allen modernen Systemen der Fall ist).

Leider ist diese Sorte von Bedrohung nur schwer erkennbar und schwer zu beheben. Übliche Quellen von Race Conditions sind Datesystemzugriffe oder Rechtemanagement.

Allerdings erfordert bspw. die Ausnutzung von Dateisystem Race Conditions auch lokalen Zugriff, bspw. um Dateien zu modifizieren. Diese Art von Angriff lässt sich lösen, indem soweit möglich mit Dateideskriptoren statt Dateinamen gearbeitet wird. Dies verhindert, dass sich die Datei ändert, mit der gearbeitet wird.

2.7 SQL Injections und Cross-Site-Scripting (XSS)

Diese Sorte von Angriffen wird meist in Sprachen wie Java, PHP, Perl, Python gefahren. Diese machen Angriffe wie Buffer Overflows und Format String Angriffe bereits per Design extrem schwer.

Allerdings gibt es hier andere Angriffe wie **SQL Injection**. Diese treten auf, wenn dynamsich SQL-Queries aus Strings erzeugt werden, schlimmstenfalls durch Nutzereingaben. Dabei ist es bei schlechtem Design möglich, dass ein Angreifer die eigentliche SQL-Query um Steuerzeichen (z. B. Anführungszeichen für Stringende, Semikolon für Ende der Query) zu erweitern, die es ihm ermöglichen, im Anschluss eine beliebige (oder abgewandelte) SQL-Query auszuführen.

Diese Art von Angriffen ist eine ganze Familie von Angriffen, die sich nicht nur auf SQL beschränkt, sondern auch andere Systeme mit beinhaltet, z. B. LDAP Injections, CRLF Injections oder auch **Cross-Site-Scripting** (XSS), bei der eine Möglichkeit ausgenutzt wird, aus der Ferne irgendwelchen Programmcode im Browser des Opfers auszuführen, sodass dieser Code mit den Rechten des Benutzers ausgeführt wird (z. B. mit Rechten eines Administrators).

2.8 Malware

Eines der größten Probleme heutzutage ist boshafte Software (**Malware**), die vom Opfer aufgrund von Vertrauen heruntergeladen und ausgeführt wird, aber im Hintergrund Schadcode ausführt, um bspw. Nutzer auszuspionieren oder Daten zu zerstören (z. B. Ransomware). In so einem Fall wird also nicht irgendeine Schwachstelle in laufender Software ausgenutzt, sondern Menschen werden manipuliert, Schadcode auszuführen. Bekannte Quellen für Schadcode sind bspw. fremde USB-Sticks, Code aus E-Mails, gehackten Websites oder auch fehlerhaften Betriebssystem-Updates. Ein ganz wichtiges Einfallstor für Malware sind E-Mails. Angreifer können hier Schadcode in HTML-Nachrichten einbetten, bösartige Links einfügen oder Makros in Word-Dokumenten verstecken, die beim Anschauen ausgeführt werden und Schadcode nachladen.

Malware wird in mehrere Klassen unterschieden:

Backdoors sind (teilweise unerwünschte) Features wie Vendor Logins, die den Zugriff auf das System erlauben. Diese breiten sich nicht aus.

Trojanische Pferde werden genutzt, um Computer fernzusteuern und wird häufig in normale Software eingebettet, um sich zu verstecken.

Rootkits sind spezielle trojanische Pferde, die sich tief im Betriebssystem verstecken. Diese können teilweise auch das ursprüngliche Betriebssystem als VM emulieren (**Blue Pill**) oder sich in Mikrocontrollern verstecken.

Viren verbreiten sich durch Kopieren auf Wechselmedien (z. B. USB-Sticks).

Übliche Gegenmaßnahmen sind meist schwierig. Gegen bekannte Malware sind Virenscanner und Netzwerküberwachung im Einsatz. Regelmäßige Updates sind zudem wichtig, um Schwachstellen, die das Einschleusen von Malware ermöglichen können, verhindern. Bei der Installation von Software sollte nach Möglichkeit auf signierte Software gesetzt werden, die von vertrauenswürdigen Quellen stammt (was nur schwer möglich ist). Software sollte zudem regelmäßig durch Audits und System Call Monitoring auf mögliche Backdoors geprüft werden.

Besonders wichtig ist vor allem, Benutzer darauf zu trainieren, vorsichtig zu agieren. Dabei sollte das Prinzip der geringstmöglichen Privilegien durchgesetzt werden. Dies bedeutet, dass Software niemals mehr Berechtigungen bekommt, als tatsächlich für die Arbeit notwendig ist.

Allerdings ist auch das im zweifelsfall nicht hilfreich.

2.9 Exploiting Trust

2.9.1 Reflections on Trusting Trust

Vertrauen kann immer wieder missbraucht werden. Vertrauensannahmen werden teilweise durch Programmierer implizit getroffen und nicht näher dokumentiert oder definiert. Beispielsweise wird implizit angenommen, dass Compiler den Code korrekt kompilieren. Bspw. könnte jedoch eine Backdoor im Compiler prüfen, ob gerade ein bestimmtes Programm kompiliert wird, und genau dann in den Programmcode eine Backdoor reinkompilieren, die im Quellcode des ursprünglichen Programms nicht existiert. So war es bspw. Ken Thomposn möglich, das `login`-Programm auf Unix-Systemen so zu modifizieren, dass (nur) er sich auf beliebigen Systemen als beliebiger Nutzer mit einem besonderen Super-Passwort einloggen konnte.

Durch Einbau dieser Backdoor mit einem selbst reproduzierendem Programm (*Quine*) ist es möglich, dass der Compiler immer wieder die Backdoor einkompiliert, selbst wenn der Bug aus dem Quellcode des Compilers entfernt wird. Somit wäre der Compiler (und damit die betroffenen zu kompilierenden Programme) selbst dann betroffen, wenn die Backdoor erkannt und entfernt wird.

Vertrauen ist genau genommen sogar notwendig für die Programme, mit denen andere Programme auf Schwachstellen analysiert werden, die Hardware, auf denen (Mikro-)Code läuft und selbst der Software, die die Chips designt. Denn theoretisch wäre es nämlich sogar möglich, dass ein Angreifer die Software mit einer Backdoor versieht, die den Chip designt, auf dem der Code läuft, der angegriffen werden soll. Wirklich vertrauen kann man Code (und Hardware) nur, wenn man den kompletten Produktionsprozess selbst durchgeführt hat.

2.9.2 Security of Open Source Software

Eine beliebte Annahme ist, dass Open Source Software ja sicher sei, weil jeder den Quellcode lesen und Schwachstellen finden könnte. Ein Beispiel für Sicherheitsprobleme ist CVE-2008-0166 [4], bei der eine Funktion, die angeblich uninitialisierte Daten zum PRNG hinzufügt, auskommentiert wurde. Dies war aber diejenige Funktion, die dem PRNG Entropie zugeführt hat. Durch Entfernung dieser Funktion wurde der PRNG vorhersehbar, was gravierende Auswirkungen auf die Vertraulichkeit erzeugter Schlüssel u. Ä. hatte, da nur noch 2^{18} unterschiedliche Schlüssel erzeugt werden konnten.

Diese Schwachstelle betraf alle SSL- und SSH-Schlüssel, die mit OpenSSL auf Debian-basierten Systemen erzeugt wurden. Allerdings waren selbst sichere Schlüssel betroffen, die für den Nachrichtenaustausch mit einem kompromittierten System genutzt wurden, wenn dort DSA-Schlüssel eingesetzt wurden. Dies ist ein Problem, da sich bei DSA der Schlüssel rekonstruieren lässt, wenn zum Ciphertext auch die (jetzt leicht vorhersagbare) Nonce bekannt ist.

Wenn die generierten Schlüssel kompromittiert sind, ist es auch möglich, Passwörter auszulesen und sich auf fremden Systemen einzuloggen. Damit hätte ein Angreifer also bspw. auch die Debian Paketserver angreifen und dort Malware einschleusen können. Durch die o. g. gezeigte Methode, Schadcode über Compiler einzuschleusen, wäre es so möglich gewesen, Schwachstellen für lange Zeit für Debian-Systeme zu verteilen.

Diese Schwachstelle wurde nicht durch Analyse des Quellcodes gefunden, sondern durch den Umstand, dass eine Person „zufällig“ auf zwei Systemen gleiche Schlüssel erzeugt hat. Auch bei der Lücke bei xz-Utils [2, 5] wurde das Problem zufällig gefunden, weil jemand Verzögerungen bei OpenSSH gefunden hat.

Es lässt sich also festhalten, dass der Mythos, dass Open Source Software regelmäßig auf Schwachstellen studiert wird, nicht wahr ist.

2.10 Trating Input in Secure Programs

Eine Grundannahme bei Eingaben ist, dass möglichst alles nicht vertrauenswürdig ist. Vertrauen sollte immer nur dann gegeben werden, wenn es absolut notwendig ist, z. B.

- Benutzereingaben sollten immer validiert werden.
- Ein Cloud-Anbieter, bei dem Dienste gehostet werden, muss vertraut werden, dass die Eingaben, die man bspw. per WebKVM macht, nicht mitgelesen werden.
- Zugriffe von Diensten, die auf eigenen Servern laufen, sollten nie mehr Berechtigungen auf dem Server haben als notwendig.
- Bei sämtlichen Eingaben/Übertragungen von/über Netzwerke(n) muss angenommen werden, dass ein Angreifer boshafte Daten eingeschleust oder die übermittelten Daten mitgelesen hat.

Wichtig ist auch, eigene Programme so weit wie möglich einzuschränken. Dies beginnt bei der korrekten Konfiguration von Signal-Handling (explizit default-Verhalten einstellen, damit Einstellungen vom Eltern-Programm rückgängig gemacht werden) und geht

bis hin zu freiwilligen Ressourcen- (rlimit) und Zugriffsbeschränkungen (z. B. Landlock LSM).

2.11 Countermeasures

Verschiedene Gegenmaßnahmen für Bedrohungen in Systemen sind möglich. Diese sind unterschiedlich gut. Die weit verbreitete Maßnahme ist „*Penetrate-and-Patch*“, bei der eine Bedrohung erst bekämpft wird, nachdem sie bereits ausgerollt wurde. Im günstigsten Fall wurde eine Bedrohung von einem Sicherheitsforscher gefunden und mit einer Frist gemeldet, zu der die Bedrohung veröffentlicht wird. Patches für solche Bedrohungen werden dann häufig unter Zeitdruck geschrieben und behandeln häufig nur Symptome. Diese Patches werden dann teilweise aber auch nur langsam oder gar nicht ausgerollt, weil Systeme manuell gewartet, mit neuer Firmware bespielt oder zertifiziert werden müssen. In dieser Zeit ist es also möglich, die Bedrohung aktiv auszunutzen. Dieser Ansatz ist der teuerste und unsicherste von allen.

Stattdessen sollte (unabhängig vom Softwareentwicklungsprozess) Sicherheit bereits im Softwareentwicklungsprozess berücksichtigt werden. Idealerweise ist eine Person dafür zuständig, die Sicherheitsaspekte der Software zu prüfen. Solches Personal benötigt aber nicht nur ein Verständnis von Sicherheit, sondern auch ein tiefes Verständnis des Softwareentwicklungsprozesses und muss als Ansprechpartner für Entwickler und Softwarearchitekten zur Verfügung stehen. Solche Personen sind nur schwer zu finden und kostet viel Geld.

Eine weitere wichtige Maßnahme ist die Erstellung von *Sicherheitsanforderungen*, die genau spezifizieren, welche Sicherheitsmaßnahmen umgesetzt werden, welche Daten und Vorgänge schützenswert sind und wie sie geschützt werden sollen. Spezifiziert werden muss dabei, was das System können muss und was es nicht tun darf, warum das System sich in einer bestimmten Art und Weise verhalten soll und wie Informationen geschützt werden sollen. Dabei sollte insbesondere auch berücksichtigt werden, wie schützenswert Daten sind und wie lange sie geschützt sein müssen.

Es ist ein gängiges Mantra, dass Security tief im Softwareentwicklungsprozess berücksichtigt werden muss. Dies wird aber realistisch fast nur für Software gemacht, die zertifiziert werden muss. Sicherheitspersonal sollte schon beim Product Design mitwirken und sich auf Implikationen von Designentscheidungen für die Sicherheit beschäftigen, u. A. wie Daten zwischen Komponenten fließen, wie Berechtigungen durchgesetzt werden und welche Komponenten wie anderen vertrauen.

Stichwortverzeichnis

accountability, 3
Address Space Layout Randomization, 13
Angriff, 3
ASLR, 13
authorization violation, 4

Backdoor, 15
Bedrohung, 3
Bedrohungsbaum, 5
Blue Pill, 16
Buffer Overflow, 10

canary, 13
Cross-Site-Scripting, 15

eavesdropping, 4
Erkennung, 6

Fälschung, 4
Format-String, 14

honeypot, 6

Integrität, 3

kontrollierter Zugriff, 3

Malware, 15
masquerade, 4
Memory Integrity Checking, 13

NOP-slide, 12

PDU, 4
Penetrate-and-Patch, 18
Prävention, 6

Quine, 16

Race Condition, 14
Reaktion, 7
repudiation, 4
Return-Oriented Programming, 13
Rootkit, 15
ROP, 13

Sabotage, 4
security service, 7
setgid, 10
setuid, 10
Sicherheitsanforderung, 18
Sicherheitsziel, 3
SQL Injection, 15
Stack Frame, 11
Stack Smashing, 12
Stackguard, 13

threat, 3
threat tree, 5
trojanisches Pferd, 15

Verfügbarkeit, 3
Vertraulichkeit, 3
Virus, 16

XSS, 15

Literatur

[1] *AddressSanitizer — Clang 19.0.0git documentation*. URL: https://clang.llvm.org/docs/AddressSanitizer.html (besucht am 25. 04. 2024).

[2] Lasse Collin. *XZ Utils backdoor*. 2024. URL: https://tukaani.org/xz-backdoor/ (besucht am 02. 05. 2024).

[3] *Formatted Output (The GNU C Library)*. URL: https://www.gnu.org/software/libc/manual/html_node/Formatted-Output.html (besucht am 25. 04. 2024).

[4] *NVD - CVE-2008-0166*. 2008. URL: https://nvd.nist.gov/vuln/detail/CVE-2008-0166 (besucht am 02. 05. 2024).

[5] *NVD - CVE-2024-3094*. 2024. URL: https://nvd.nist.gov/vuln/detail/CVE-2024-3094 (besucht am 02. 05. 2024).

[6] Günter Schäfer. "Schutz von Kommunikationsinfrastrukturen". 2024.

[7] *Valgrind Home*. URL: https://valgrind.org/ (besucht am 25. 04. 2024).