Hacking Linux

Marek Kubica
Betreuer: Dipl.-Inform. Holger Kinkelin
Proseminar Network-Hacking und Abwehr WS09/10
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: kubica@in.tum.de

KURZFASSUNG

Eines der üblichen Argumente für Linux ist, dass es sicher sei. Dieses Argument war lange Zeit wahr, weil Linux ein Nischendasein fristete, aus dem es sich jedoch inzwischen gelöst hat. Damit wird auch dieses System für Angreifer interessant: in der vorliegenden Arbeit werden typische Sicherheitsprobleme in Linux-Systemen vorgestellt. Dabei werden neben den Problemen auch Lösungen aufgezeigt, die die Sicherheit eines Systemes auf ein sehr hohes Niveau bringen können. Nicht jede Sicherheitslücke ist behebbar, aber inzwischen wird auch bei Linux die Sicherheitsproblematik zunehmend ernstgenommen.

SCHLÜSSELWORTE

Linux, Kernel, Buffer Overflow, Symlink, Bruteforce, Sicherheit

1. EINLEITUNG

Linux ist ein freier Unix-Klon, der von einer großen Gemeinschaft über den ganzen Planeten verteilter Programmierer entwickelt wird. Die Kommunikation zwischen diesen Programmierern verläuft zu einem großen Teil über das Internet, man könnte direkt sagen, dass Linux ein Kind des Internets ist und somit auch das Internet und generell Netzwerke seit langem ein wichtiger Teil von Linux sind. Schon seit vielen Jahren hat sich Linux als Betriebssystem für Server etabliert, daher ist es sinnvoll, sich mit den Problemen und Risiken von Linux-Systemen auseinanderzusetzen, sowie mögliche Lösungen zu betrachten.

Um die Sicherheitsprobleme zu verstehen, ist es notwendig zurückzublicken. Auf diese Weise kann man die zu dieser Zeit getroffenen Entscheidungen und ihre heutigen Konsequenzen besser verstehen.

Unix ist Anfang der 1970er-Jahre beim US-Telekommunikationsunternehmen AT&T als vereinfachte Version eines älteren Systems namens Multics entstanden. Dieses System war damals komplett in PDP-11/20 Assembler geschrieben, wie es für Betriebssysteme üblich war. Parallel dazu ist bei AT&T auch die Programmiersprache C von Dennis Ritchie entwickelt worden. Ritchie war neben C ebenfalls an Unix beteiligt und so geschah es, dass Unix 1973 in C neu geschrieben wurde. Seit diesem Zeitpunkt ist Unix untrennbar mit der Sprache C verbunden [22].

C wurde jedoch nicht nur zum Programmieren von Betriebssystemen verwendet, sondern auch für die Programme, die auf diesen Systemen laufen. Dies ist darin bedingt, dass C auf Unix eine naheliegende Wahl war, die Unterstützung durch Editoren und Programmiertools gut, und das Unix-Umfeld traditionell viele C-Programmierer aufweist. Auch sind in C geschriebene Programme üblicherweise sehr schnell, was insbesondere aufgrund der beschränkten Hardware in den 1970er und 1980er-Jahren wichtig war. So kam es, dass C auch außerhalb der Unix-Welt Fuß fassen konnte und Ableger von C wie Objective-C und C++ entstanden sind.

Die Anfänge von Linux reichen in das Jahr 1991 zurück, als der damalige Informatikstudent Linus Torvalds mit dem in C geschriebenen Lehr-Betriebssystem MINIX unzufrieden war und entschied, selbst ein Unix-ähnliches System zu schreiben [21].

Die Popularität von C unter Systemen wie Linux hat jedoch nicht nur Vorteile: so ist die Sprache nun über 30 Jahre alt und die Anforderungen haben sich geändert. Das größte Problem an C ist der direkte Zugriff auf den Speicher, was einerseits hohe Effizient erlaubt, aber andererseits dem Programmierer eine große Verantwortung aufbürdet. Darüber hinaus ist beim Entwurf der C Standard Library der Sicherheitsaspekt nicht bedacht worden, da Computersicherheit im Jahr 1972 bei weitem nicht so ernst genommen wurde wie heutzutage [8]. Durch die geringe Vernetzung waren auch Hacking-Angriffe eher die Ausnahme, sodass diese Probleme damals nicht wichtig genug schienen.

Diese Ausarbeitung beschreibt in Abschnitt 2 die größten Sicherheitsprobleme, die heutzutage unter Linux existieren. Danach werden in Abschnitt 3 Lösungen vorgestellt, mit denen sich viele Sicherheitslücken schließen lassen.

2. TYPISCHE ANGRIFFSVEKTOREN

Es gibt viele Verfahren wie man Sicherheitsmechanismen überwinden kann. Eine vollständige Auflistung würde den Rahmen dieser Arbeit sprengen und dennoch nie vollstän-

dig und aktuell sein können. Daher werden im Folgenden nur „klassische“ Angriffsmethoden vorgestellt, mit denen eine Vielzahl von Software angegriffen werden kann.

2.1 Buffer Overflow

Um Buffer Overflows zu verstehen, ist es notwendig zu verstehen, wie Programme und Daten im Speicher liegen. In Von-Neumann-Maschinen gibt es einen Speicher, in dem sowohl Programme als auch Daten liegen [23]. Somit ist es möglich, Programmcode in den Speicher zu schreiben und danach direkt auszuführen, was für Laufzeit-Kompilierung (Just In Time, JIT) optimal ist, aber auch Sicherheitsrisiken aufwirft, wenn ein Angreifer in die Situation kommt, eigenen Programmcode in den Speicher des angegriffenen Systems zu schreiben.

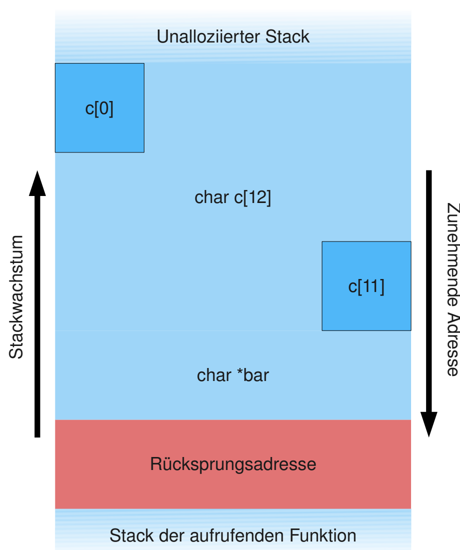

Abbildung 1: Initialer Puffer

In C ist es notwendig, die Länge von Variablen im Speicher entweder im vorhinein zu deklarieren oder dynamisch Speicher via `malloc()` anzufordern. Oft werden in diesen Speicherbereichen Strings gespeichert. Diese Variablen sind dann als „Buffer“ (Puffer) bekannt. Man kann innerhalb dieses Speicherbereiches auf sichere Art und Weise scheiben [6]. Jedoch führt die C-Runtime keinerlei Bereichsprüfung aus, so kann es bei der Übergabe von zu langen Strings passieren dass man über den reservierten Speicher hinausschreibt – üblicherweise liegen dort aber andere Daten oder Teile des Programmes die durch diese Buffer Overflows korrumpiert werden. Normalerweise stürzt ein solches Program früher oder später ab, wenn versucht wird sinnvolle Daten aus dem überschriebenen Speicherbereich zu lesen. Dies ist ein häufig anzutreffender Programmierfehler der durch die C-Standard-Library noch verschlimmert wird: sie enthält viele Funktionen, die Strings in Puffer schreiben, ohne zu prüfen ob der Puffer groß genug ist, um den String aufzunehmen. Wenn der Puffer zu kurz ist, wird einfach dahinter weitergeschrieben und die dahinterliegenden Daten überschrieben. Diese Probleme sind in C derart häufig, dass es Speicher-Debugger wie Valgrind gibt, die den Programmierern helfen, solche Fehler aufzudecken.

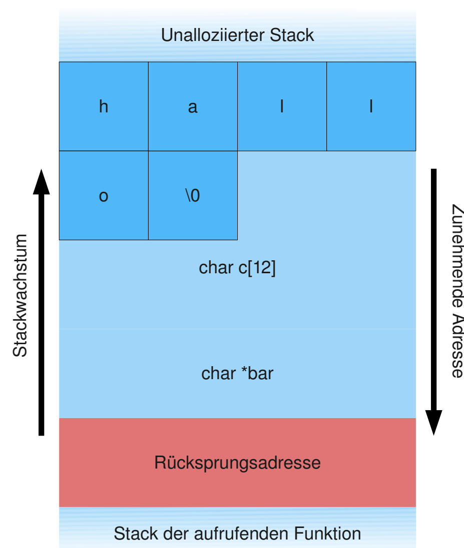

Abbildung 2: Puffer mit ’hallo’

Abbildung 1 zeigt schematisch, wie ein 12 Byte großer Puffer auf dem Stack liegt, Abbildung 2 zeigt wie der legitime String „hallo“ in den Puffer geschrieben wurde. Man kann erkennen, dass unter dem für den Puffer reservierten Bereich die Rücksprungsadresse liegt zu der das Programm nach Beendigung der aktuellen Funktion springen wird. Diesen Aufbau kann man ausnutzen, indem man, wie Abbildung 3 zeigt, einen zu langen String in den Puffer schreibt. Dabei wird die Rücksprungsadresse durch den neuen String überschrieben und der Angreifer kann das Programm dazu bringen, Code an einer beliebigen Adresse auszuführen. Im vorliegenden Beispiel versucht der Angreifer den Code der im Puffer liegt auszuführen, was aber scheitern würde da der String keinen gültigen Maschinencode enthält und daher das Programm crashen würde. Ein richtiger Angriff würde statt dem Buchstaben „A“ einen richtigen „Shellcode“ in den Puffer plazieren.

Diese „Shellcodes“ sind nicht nur einfache Eingabedaten, sondern speziell präparierte Strings die Maschinencode enthalten, der auf dem angegriffenen System ausführbar ist [13]. Wenn es zu einem Buffer Overflow kommt und ein Shellcode in den Programmspeicher kopiert wurde, kann der Shellcode ausgeführt werden. Ab diesem Zeitpunkt ist die Sicherheit des Programmes verletzt, da der Shellcode beliebigen Code ausführen kann, unter anderem, wie schon der Name impliziert, eine Shell starten. Wenn das angegriffene Programm nun zusätzlich noch mit root-Rechten läuft, ist die Integrität des gesamten Systems gefährdet.

2.2 Symlink-Attacken

Ein weiteres, relativ subtiles Problem sind Symlink-Attacken. Symlinks sind Zeiger auf andere Dateien. Diese Symlinks können ganz normal geöffnet und editiert werden – jegliche Änderung wird an der Datei vorgenommen, auf die

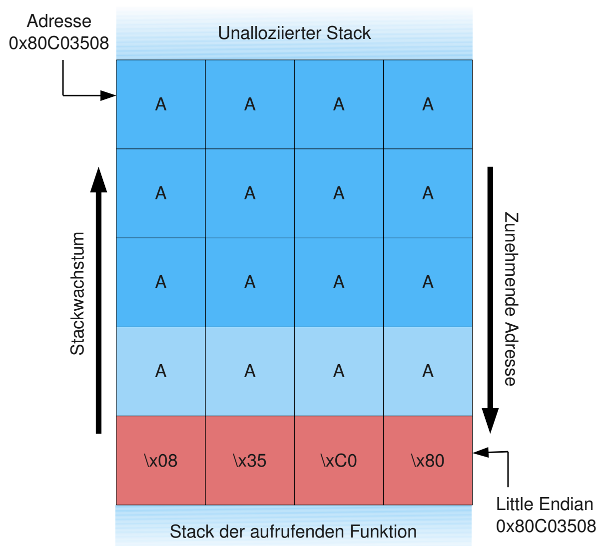

Abbildung 3: Overflow: Puffer überschreibt Rücksprungsadresse

verlinkt wurde.

Viele Applikationen nutzen temporäre Dateien die in der Regel in `/tmp` angelegt werden. Jedoch ist es bei unsicher geschriebenen Programmen möglich, dass ein Angreifer einen Symlink anlegt, wo das Programm eine temporäre Datei erwartet. Wenn das Programm diesen Symlink ohne weitere Prüfung einfach verwendet und sie öffnet, öffnet sie in Wahrheit die Datei auf die der Angreifer den Symlink hat zeigen lassen. Wenn das Programm in diese Datei schreibt, schreibt es ebenfalls in das neue, vom Angreifer festgelegte Ziel.

Dies ist besonders kritisch, wenn das angegriffene Programm mit root-Rechten läuft, so dass der Angreifer das Programm dazu überreden kann, in beliebige Dateien zu schreiben. Analog ist es dadurch auch möglich, das Programm beliebige Dateien lesen zu lassen. Das ermöglicht etwa das ausspionieren von Dateien wie `/etc/shadow`, um über Bruteforce (siehe Abschnitt 2.3) diese Passwörter zu knacken.

Erschwerend kommt hinzu, dass wenn das Programm die Zugriffsrechte der temporären Datei ändert, sich diese Änderungen nicht auf den Symlink auswirken, sondern auf die Datei auf die der Link zeigt. Somit ist es unter Bedingungen möglich, über eine Symlink-Attacke ein root-Programm dazu zu bringen, Dateien für beliebige Nutzer les- oder schreibbar zu machen, wodurch der Angreifer sich erweiterte Rechte auf dem System erschleichen kann.

2.3 Bruteforce

Eine der trivialsten Arten sich Zugriff auf ein System zu verschaffen ist es, Passwörter zu erraten. Diese Methode ist sehr simpel, jedoch gibt es dennoch oftmals zu viele Passwörter als dass ein Mensch die Geduld hätte, sie alle auszuprobieren. Daher gibt es Software, die Passwörter automatisiert nach bestimmten Regeln ausprobieren können. Die einfachste Methode ist es, beliebige Zeichenketten zu generieren und diese als Passwort auszuprobieren. Gegebenfalls kann das sehr zeitaufwändig sein. Eine oftmals effizientere Methode nutzt aus, dass Menschen dazu tendieren einfache Passwörter zu verwenden – Passwörter wie „test“, „1234“ oder „password“ sind einige sehr populäre Passwörter. Daher ist es möglich, Wörterbuch-Dateien zu verwenden, in denen sehr viele Wörter stehen und diese von der Software automatisch ausprobieren zu lassen.

Es gibt zwei Möglichkeiten, Passwörter nach Bruteforce-Methode zu knacken:

1. direkte Anmeldeversuche: Diese Methode ist zeitaufwändig, weil sichere Software zwischen gescheiterten Loginversuchen zur Sicherheit eine bestimmte Zeit wartet und keine weiteren Versuche zulässt. Ein weiterer Nachteil ist, dass ein Administrator diesen Einbruchsversuch bemerken kann.

2. Passwortdatei knacken: Passworte werden in der Regel nicht im Klartext gespeichert sondern in gehashter Form, somit lässt sich aus dem Hash das ursprüngliche Passwort nicht bestimmen. Sofern jedoch der Hashingalgorithmus bekannt ist, ist es möglich sehr schnell viele mögliche Passwortkandidaten zu hashen und zu prüfen, ob der gesuchte Hashwert generiert wurde. Für Hashingverfahren wie MD5 existierten sogar sogenannte Rainbow-Tables, bei denen die Hashwerte schon vorberechnet wurden, so dass diese Methode sehr schnell zum Ziel führt. Daher wurde zusätzlich zum Hash ein sogenannter „Salt“ eingeführt, der vor dem Hashen an das Passwort angehängt wird.

   Diese Methode ist wesentlich schneller als die direkte Methode, erfordert aber den Zugriff auf so eine Passwortdatei, was bei gut abgesicherten Systemen möglichst vermieden werden sollte.

Unter Linux wird beim Login-Vorgang die Datei `/etc/passwd` geprüft, um festzustellen, ob der User der eingeloggt werden soll, überhaupt auf dem System existiert. Diese Datei ist eine durch Doppelpunkte separierte Textdatei, in der neben dem Usernamen auch weitere Informationen über den User stehen. Früher war dies auch der Ort, an dem die Passwortinformationen gespeichert wurden. Da diese Datei jedoch für alle Systemuser lesbar ist, steht das Passwort inzwischen nicht mehr in dieser Datei. Der neue Ort des Passwortes ist `/etc/shadow`, wo es in gehashter Form steht und nicht für beliebige Nutzer lesbar ist.

Dadurch entfällt die Möglichkeit, das Passwort auf diese Weise zu knacken (sofern man nicht sowieso schon root-Rechte erlangt hat), jedoch ist es immer noch für unprivilegierte Nutzer des Systems möglich, die Benutzernamen der anderen Nutzer herauszufinden – das bedeutet, dass diese nicht mehr erraten oder auf anderem Wege herausgefunden werden müssen. Erschwerend kommt hinzu, dass unter Unices der Superuser nahezu immer „root“ heißt, somit sind direkte Angriffe auf den root-Account am einfachsten.

Der eigentliche Knackvorgang ist ab diesem Zeitpunkt mechanisch und trivial: die Software probiert so lange, bis sie

Erfolg hatte und ein Passwort erraten hat oder der Suchraum erschöpft ist.

2.4 Input validation

Beim Schreiben von Software gehen Programmierer meistens davon aus, dass die Nutzer die Software so verwenden wie sie selbst. Dadurch entwickeln sie eine art Art Betriebsblindheit, die dazu führt, dass das Programm bei einer unüblichen Benutzereingabe fehlerhaftes Verhalten an den Tag legt. In der Entwicklung wurden einfach bestimmte Fälle übersehen und in diesen ist das Programmverhalten daher unbekannt.

Durch falsche, unvorhergesehene Eingaben können die Nutzer ein Programm aus dem Tritt bringen, so dass es entweder falsche Ergebnisse liefert oder abstürzt. In bestimmten Fällen ist es also möglich, das Programm zu einer Aktion zu zwingen, die die Programmierer nicht vorgesehen haben. Eine solche Lücke könnte im Login-Programm sein, dass es den Benutzer bei bestimmten Eingaben ohne Passwort anmelden lässt [13] oder einfach abstürzt, so dass sich keine anderen, legitimen Benutzer mehr anmelden können.

Ein Beispiel für fehlerhafte Prüfung von Eingaben ist der „Ping of death“: üblicherweise sind ICMP PING-Pakete nur wenige dutzend Byte klein, jedoch ist es möglich Pakete größer als $2^{16} - 1$ Bytes zu schicken, indem man sie fragmentiert. Auf dem Zielsystem werden diese Pakete dann von den Netzwerktreibern wieder in ein ganzes Paket zusammengesetzt und zur Verarbeitung an ICMP-Implementierung im Kernel weitergeleitet, die dann crasht und das System dadurch aufhängt. Interessanterweise war dieser Bug nicht Linux-spezifisch, auch andere Systeme wie Windows, Mac OS sowie einige Netzwerkdrucker waren betroffen. Dies bezeugt, dass fehlerhafte Eingabebehandlung ein generelles Problem ist.

Es gibt auch Tools mit denen man sich bei dem Finden und Ausnutzen solcher Bugs helfen lassen kann; sie werden *Fuzzer* genannt. Diese Tools versuchen Eingabedaten zu generieren, die das zu testende Programm aus dem Tritt bringen. Dabei gibt es verschiedene Vorgehensweisen, von intelligenten Methoden wie dem Generieren von fast-korrekten Eingabedaten bis hin zur Verwendung von Zufallsdaten [19].

Ein modernes Beispiel für fehlerhafte Eingabeprüfung sind XSS-Lücken sowie SQL-Injections in Webapplikationen [9]. Dabei wird jedoch in der Regel nicht der Server angegriffen, sondern nur die Webapplikation selbst.

3. GEGENMAßNAHMEN

Eine hunderprozentige Sicherheit wird es natürlich nie geben, jedoch reichen oft schon ziemlich triviale Maßnahmen, um die Sicherheit des Systems zu erhöhen. Dadurch kann man etwa das System für automatisiert ablaufende Exploits, die das Internet nach gefährdeten System durchsuchen uninteressant machen, was schon viele Angriffe durch Massenscans von Computern ausschließt.

3.1 Aktuelle Software

Wenn Sicherheitslücken in Programmen bekannt werden, werden – gerade bei wichtiger Software wie Apache oder dem Linux-Kernel – von den Entwicklern Patches oder aktualisierte Releases der Software bereitgestellt, die das Problem beheben. Diese Verbesserungen müssen dann von den Administratoren möglichst zeitnah auf ihren Systemen installiert werden, um sich vor Angreifern wehren zu können, denn diese haben ebenfalls die Sicherheitslücken im Blick.

Dadurch dass nahezu alle Linux-Distributionen zur Software-Verwaltung ein ein Paket-Management (wie APT, RPM und ähnliches) bieten, sind Administratoren von Linux-Systemen in einer außergewöhnlich komfortablen Lage. So stellen die meisten namhaften Anbieter der Distributionen Paketquellen bereit, die gepatchte und aktualisierten Pakete enthalten. Das geht teilweise so weit, dass selbst noch alte Softwarepakete, die nicht mehr von ihren Entwicklern gepflegt werden, durch die Distributoren mit aktuellen Patches versorgt werden [3]. Jedoch gibt es von Distribution zu Distribution Unterschiede, wie schnell die Updates veröffentlicht werden [4].

Jedoch sind nicht alle Paket-Manager genauso sicher, so sollte man darauf achten, dass die Distribution, die man nutzt, signierte Pakete anbietet. Ansonsten wäre es möglich, dass Angreifer über einen Man-in-the-Middle-Angriff kompromitierte Pakete unterschieben, die vom Paketmanagement dann installiert würden. Leider sind signierte Pakete bisher immer noch nicht selbstverständlich. Obschon die Gefahr seit langem existiert, hat etwa Debian erst im Jahr 2005 ein signiertes Archiv eingeführt. Immerhin ist das von Debian dokumentierte System relativ aufwändig und durchdacht, um möglichen Schaden gering zu halten [2]. Es ist ratsam, sich zu informieren, wie der jeweilige Distributor dieses Problem handhabt.

Dennoch, durch die zentrale Softwareverwaltung sind Sicherheitsupdates auf Linux-Systemen genauso simpel wie Paketinstallationen; also mit wenigen Befehlen machbar.

3.2 Kompartimentierung

Man sollte sich immer bewusst sein, dass die Sicherheitsvorkehrungen eines Programmes scheitern können. Daher ist es wichtig, dass selbst wenn ein Angreifer die Kontrolle über ein Programm erlangt es ihm nicht möglich sein sollte, das gesamte System zu übernehmen. Manchmal ist die einfachste Lösung, das Programm einfach auf einem dediziertes System zu betreiben.

Eine traditionelle Methode zur Kompartimentierung unter Unix ist die „change root“ (chroot)-Funktionalität. Dabei wird einem Programm ein anderes Wurzelverzeichnis (`/`) vorgespiegelt, üblicherweise in einem abgetrenntem Bereich des Dateisystems. Da Programme nicht überhalb des Wurzelverzeichnisses zugreifen können, kann das in dem „chroot“ laufende Programm nicht auf Daten von außerhalb zugreifen [1]. Diese Methode wird vom Linux-System direkt unterstützt, bietet jedoch keine absolute Sicherheit, da es bestimmte Angriffe gibt, mit denen das Programm aus dem „chroot“ entkommen kann. Andere Unices bieten mit Jails (FreeBSD) [7] und Zones (Solaris) [18] ähnliche, aber ausgefeiltere Ansätze. So können FreeBSD-Jails auf ein Netzwerkinterface eingeschränkt werden, sodass sie nur über dieses erreichbar sind. Solaris und OpenSolaris ermöglichen darüber hinaus auch Ressourcenmanagement, um den Speicher oder die CPU-Zeit, die einer Zone zugeteilt wird, zu be-

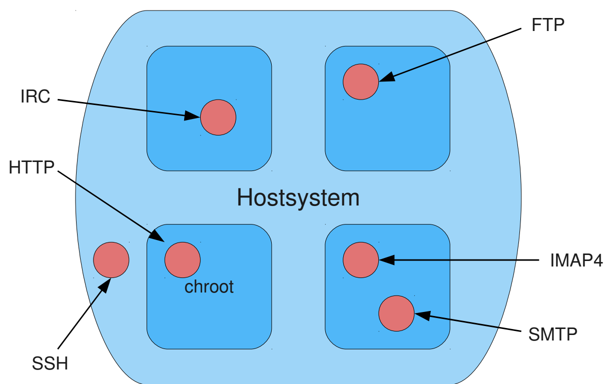

Abbildung 4: Beispielsetup mit `chroot`

schränken und somit das Gesamtsystem vor einer Überlastung zu schützen [17].

Neben dieser auf vielen Unix-Systemen verfügbaren Methode bietet Linux auch noch eine Reihe weiterer Virtualisierungstechnologien. Dabei wird dem Prozess nicht vorgespielt dass, das root-Verzeichnis woanders liegt, sondern es wird ein gesamtes System simuliert.

Sehr populär ist in dieser Hinsicht speziell unter Webhostern OpenVZ bzw. sein kommerzieller Ableger Virtuozzo. Dabei bekommt jeder Kunde einen eigenen virtuellen Computer, der sich wie ein normales System verhält. So läuft dieses virtuelle System auf einem größeren Hostsystem des Webhosters, auf dem neben dem einen System, das der Kunde bekommen hat auch viele weitere virtuelle Systeme unabhängig laufen können. Durch die Kompartimentierung wird sichergestellt, dass sich Kunden nicht gegenseitig beeinflussen können. Wenn es zu einer Kompromittierung von einem virtuellen System kommt sind die anderen virtuellen Systeme davon nicht betroffen.

Jedoch ist auch OpenVZ auf seine Weise angreifbar. Allen virtuellen Systemen ist der Kernel gemeinsam. Daher wird ein erfolgreicher Angriff auf den Kernel eines virtuellen Systemes alle anderen Systeme in Mitleidenschaft ziehen. Ansätze wie Xen und KVM, die das gesamte System virtualisieren schließen auch diese Lücke, allerdings sollte man sich immer bewusst sein, dass das sehr komplexe Lösungen sind und daher auch hier Sicherheitsupdates (Abschnitt 3.1) von zentraler Bedeutung sind.

Abbildung 4 zeigt ein Beispielsetup für Kompartimentierung. Dabei laufen mehrere Dienste auf einem gemeinsamen Hostsystem, auf dem ein SSH-Daemon läuft, um Logins zu erlauben. Die einzelnen Dienstleistungen die der Server bereitstellen soll sind in eigenen „chroots“ aufgeteilt: der HTTP-Server, der FTP-Server und der IRC-Server (Internet Relay Chat) laufen jeweils in separaten „chroots“, die Mail-Software die Mails versendet (SMTP) als auch Mails für User bereitstellt (IMAP4) ist in einem gemeinsamen „chroot“ untergebracht. Dadurch soll sichergestellt werden, dass ein Einbruch etwa in den Webserver nicht dazu führt, dass die auf dem Hostsystem gespeicherten Daten, wie etwa die Mails, kompromittiert werden und der Angriff somit nicht weiter eskalieren kann.

3.3 Absicherung des Stacks

Der Standardcompiler für C und C++ unter Linux ist der GNU C Compiler (GCC). Die Entwickler von GCC haben die Gefahr von Buffer-Overflows erkannt und in ihren Compiler eine Erweiterung namens „ProPolice“ eingebaut. Dabei werden spezielle Marker zwischen die Stackframes eingesetzt, sogenannte „canaries“. Wenn es zu einem Buffer Overflow kommt, dann werden diese Canaries als erste überschrieben werden. Diese Änderung kann nun vom Programm erkannt werden und das Programm daraufhin terminiert werden.

Dies funktioniert jedoch nur für Daten, die auf dem Stack liegen, d.h. von der automatischen Allokation von C erstellt werden. Dynamisch erstellte Datenstrukturen auf dem Heap wie etwa `malloc()` sie ermöglicht, sind immer noch durch Buffer Overflows gefährdet.

Zu erwähnen ist jedoch, dass für den Einsatz von ProPolice die Programme neu kompiliert werden müssen, da diese Sicherheitsmaßnahme standardmäßig deaktiviert ist. Einige Distributionen kompilieren ihre Binaries allerdings bereits mit Stack-Schutz, Ubuntu geht da mit positivem Beispiel vorran [20].

3.4 Absicherung des Programmspeichers

Das Problem mit „Shellcodes“ ist, dass Code in den Adressraum eines Prozesses geschrieben wird, der dann ausgeführt werden kann. Als Schutz dagegen bieten bestimmte Prozessoren Unterstützung für ein sogenanntes NX-Bit, womit Speicherregionen als nicht ausführbar markiert werden können [10]. Fast alle gängigen aktuellen PC-Prozessoren unterstützen NX, jedoch mit Ausnahme von 32-Bit x86-Prozessoren, die keine hardwareseitige Unterstützung beinhalten.

Damit das NX-Bit verwendet werden kann, ist es nötig, dass das Betriebssystem das NX-Bit unterstützt. Linux kann mit dem NX-Bit auf x86_64-Prozessoren sowohl im 32- als auch im 64-Bit Modus umgehen. Für ältere 32-Bit Prozessoren existieren Linux-Patches wie Exec Shield und PaX, die das NX-Bit emulieren können.

Durch das NX-Bit werden Speicherbereiche als „nicht ausführbar“ markiert. Somit können Shellcodes, die im Zuge von Buffer Overflows in den Programmspeicher geschrieben werden nicht mehr direkt angesprungen und ausgeführt werden. Bevor Code ausgeführt wird, wird geprüft, ob der Speicherbereich überhaupt ausführbar ist, ansonsen wird das Programm vom System terminiert.

Daneben gibt es im Linux-Ökosystem noch eine Reihe weiterer Tools, die Entwicklern helfen, ihre Speicherverwaltung abzusichern. Ein für C sehr nützliches Werkzeug ist Valgrind, ein Speicher-Debugger. Mithilfe von Valgrind kann ein Entwickler sichergehen, dass er keine Speicher-Korruption in seinen Programm hat. Dazu wird das zu debuggende Programm von Valgrind geladen und zur Laufzeit in eine Zwischenform, die *Intermediate Representation* (IR), umgewandelt. Danach wird der generierte IR-Code um weiteren IR-Code erweitert; im Falle des Speicherdebuggings ist dafür

das Tool Memcheck zuständig. Es fängt die Zugriffe auf den Hauptspeicher ab und prüft andererorts ob der Speicherbereich gültig, d.h. initialisiert ist. Zudem ersetzt es den Speicherallokator von C durch einen eigenen, der prüfen kann ob Zugriffe außerhalb des reservierten Speichers erfolgen. Schließlich wird der angepasste Code wieder in Maschinencode umgewandelt und kann vom System ausgeführt werden [14]. Die Transformationen die Valgrind ausführt, sind so aufwändig, dass das zu debuggende Programm nur noch mit einem Bruchteil seiner ursprünglichen Geschwindigkeit abläuft, daher ist Valgrind wirklich nur als für Programmierer, nicht aber für Administratoren nützlich. Ein weiteres Werkzeug ist Mudflap, welches Pointerzugriffe prüft [5], jedoch während des Kompilationsvorganges eingebunden werden muss. Es gibt also durchaus Möglichkeiten, bestehende C-Programme abzusichern, jedoch erfordert es wesentlich mehr Aufwand ein unsicheres Programm auf diese Weise abzusichern als ein mit Sicherheit im Hinterkopf geschriebenes Programm einzusetzen, wo bereits die Entwickler sich diese Mühe gemacht haben.

3.5 Limitierung der Programmrechte

Je weniger Privilegien ein Programm auf dem System besitzt, desto weniger Schaden kann es auf dem System anrichten. Dies gilt sowohl für simple Fehler im Programmcode durch die etwa Daten versehentlich gelöscht werden, als auch für Sicherheitslücken, die gezielt von Angreifern ausgenutzt werden um Schaden zu verursachen.

Daher gilt seit jeher unter Linux die Devise, so wenig Programme wie möglich mit root-Rechten auszuführen, da dadurch das System durch fehlerhafte Software nicht kompromittiert werden kann. Einige Distributionen wie Ubuntu haben beispielsweise den root-Login komplett deaktiviert. Stattdessen werden Tools wie `sudo` verwendet um Programme, die root-Rechte benötigen, zu starten.

Wenn man ein Programm als unprivilegierter User startet, hat man schon einiges an Sicherheit gewonnen, aber natürlich kann ein Programm immer noch Schaden anrichten. Es kann etwa die `/etc/passwd` ausspionieren oder auch einfach nur das Home-Verzeichnis des Users löschen, was zwar die Sicherheit des Gesamtsystemes nicht beeinträchtigt, aber dennoch für den User ein ggf. großer Schaden sein kann. Die meisten Programme benötigen jedoch nur eine Teilmenge der ihnen vom System gewährten Rechte und an dieser Stelle kann man ansetzen.

Eine Lösung könnte sein, für jedes Programm einen eigenen Benutzer im System anzulegen, jedoch ist dies Methode sehr langwierig und umständlich, daher für den Regelbetrieb ungeeignet. Selbst dann ist diese Methode noch zu grobkörnig – es ist etwa nicht möglich den Zugriff auf Sockets zu limitieren. Ein PDF-Reader benötigt üblicherweise keine Sockets, um seine Aufgabe zu erfüllen, daher könnte man auch diese Funktion deaktivieren.

Genau zu diesem Zweck sind SELinux von der NSA und AppArmor von Novell entwickelt worden – sie ermöglichen exaktere Kontrolle über die Berechtigungen von Programmen [12]. Es werden dabei den abzusichernden Programmen Sicherheitsprofile zugeordnet, die bestimmen, was ein Programm darf und wozu es nicht in der Lage sein soll.

3.6 Modularisierung von root

Linux kennt zwei Arten von Nutzern: normale Nutzer und den Superuser root. Letzterem sind viele Privilegien im System vorbehalten, wie etwa das Lesen von allen auf dem System vorhandenen Dateien oder das Schicken von Signalen an fremde Prozesse. Viele Serverdienste, die auf Ports < 1024 laufen, wie HTTP (80), SMTP (25) oder FTP (20, 21), benötigen ebenfalls Superuser-Privilegien – obwohl die Programme selbst eigentlich nicht mit besonderen Rechten laufen müssten.

Ein Beispiel ist das Unix-Programm `ping`, welches ICMP PING-Pakete schickt. Dieses Programm benötigt unter Linux Zugriff auf Raw-Sockets, hat daher das SUID-Bit gesetzt. Dieses Bit bewirkt, dass das Programm mit den Rechten des Besitzers der Datei besitzt (Owner) ausgeführt wird. Im Falle von `ping` ist dies root, somit wird es mit root-Rechten ausgeführt, selbst wenn ein normaler User es startet. Das öffnet ein Tor für Angreifer, die über Fehler des `ping`-Programmes das System attakieren können.

Hier kommen nun *Linux-Capabilities* ins Spiel: sie statten gewöhnliche User mit gewissen Privilegien aus, die root vorbehalten waren. So ist es möglich Usern Flags wie `CAP_CHOWN` (beliebiges Ändern des Dateibesitzers), `CAP_KILL` (Senden von Signalen auch an fremde Prozesse) und etliche weitere mitzugeben [11].

Am Beispiel von `ping` würde das bedeuten, dass es reicht wenn der User die Capability `CAP_NET_RAW` besitzt. Dies versetzt ihn in die Lage direkt auf Raw-Sockets zuzugreifen. Somit ist es möglich, das SUID-Bit von `ping` zu entfernen und eine potenzielle Lücke zu schließen [16].

Im Unterschied zu Abschnitt 3.2 sind Capabilities ausgelegt, Prozessen zusätzliche Rechte *zu geben*, statt die Rechte von Prozessen zu *limitieren*.

3.7 Wechsel der Software

Software wird immer mit bestimmten Zielen und bestimmten Vorraussetzungen geschrieben. So war bei der Implementierung zahlreicher älterer Softwarepakete der Sicherheitsaspekt nicht so wichtig, da das Internet wesentlich kleiner war und mehr gegenseitiges Vertrauen aufgebracht wurde.

So ist etwa der erste Mail-Server, Sendmail, für seine vielen Sicherheitslücken und Probleme bekannt, die teilweise aus Unachtsamkeit der Programmierer folgten und teilweise an der falschen Konfiguration durch Administratoren. Die Konfiguration von Sendmail ist für ihre Flexibilität bekannt, jedoch auch für die daraus bedingten Komplexität der Einrichtung. Daher ist inzwischen die Popularität von Sendmail zugunsten simplerer Mail-Server zurückgegangen. Glücklicherweise existieren Alternativen, die genau diese Probleme aufgegriffen haben. Die bekannteste Sendmail-Alternative ist Postfix, die speziell auf Sicherheit getrimmt ist [15]. So ist Postfix kein einzelnes Programm mehr, sondern eine Ansammlung von untereinander kommunizierenden Programmen, die mit minimalen Rechten laufen und sogar in chroots funktionieren können (siehe Abschnitt 3.2). Damit wird ein möglicher Schaden am System möglichst gering gehalten.

Eine andere Möglichkeit wäre der Ersatz von problemati-

schen Protokollen durch andere, brauchbarere Alternativen. Das *File Transfer Protocol* (FTP) und dessen Serversoftware waren oftmals von Sicherheitsproblemen betroffen – man könnte dieses Protokoll ohne weiteres durch SSH ersetzen, welches mit SFTP eine sichere Alternative zu FTP bietet.

Letztlich muss man auch überlegen, welche Dienste ein Rechner anbieten soll und welche nicht unbedingt notwendig sind. Je weniger Services laufen, desto weniger Angriffsfläche bietet das gesamte System.

4. ZUSAMMENFASSUNG

Die vorherigen Abschnitte haben gezeigt, dass es zwar durchaus möglich ist ein sicheres Linux-System zu betreiben, dies aber dennoch einigen Aufwand seitens des Administrator erfordert. Dennoch bietet das Linux-Umfeld eine erstaunliche Menge an Tools und Sicherheitslösungen an, mit denen man das System auf nahezu jede erdenkliche Weise absichern kann. Linux bringt mit jedem Release weitere Absicherungsmöglichkeiten mit (wie etwa Tomoyo und Smack), ebenfalls gibt es zu den aktuellen Kernel-Releases auch regelmäßig Sicherheitsupdates. Natürlich konnte nicht jedes einzelne Sicherheitsfeature in Linux im Paper angesprochen werden, da sich die Situation schnell ändert – Angreifer finden neue Lücken, Entwickler entwickeln Schutzmechanismen gegen die neuen Angriffe.

Allerdings reichen oftmals schon einfache Maßnahmen, wie aktuelle Software und sichere Passwörter um die Gefahr zu verringern. Ein System, das auf den ersten Blick sicher scheint, ist für einen potentiellen Angreifer bei einem Massenscan weniger interessant als ein System bei dem der Angreifer mögliche Sicherheitslücken leicht erkennen kann.

5. LITERATUR

[1] A. Chuvakin. Using Chroot Securely. `http://www.linuxsecurity.com/content/view/117632/49/`. [Online, accessed 17-December-2009].

[2] Debian. All about secure apt. `http://wiki.debian.org/SecureApt`. [Online, accessed 16-December-2009].

[3] Debian. Debian security FAQ. `http://www.debian.org/security/faq#oldversion`. [Online, accessed 16-December-2009].

[4] E. X. DeJesus. Linux patch problems: Your distro may vary. `http://searchsecurity.techtarget.com/news/article/0,289142,sid14_gci1202417,00.html`. [Online, accessed 16-December-2009].

[5] F. C. Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developers Summit 2003*. GCC Developers, May 2003.

[6] P.-H. Kemp. Malloc(3) revisited. `http://phk.freebsd.dk/pubs/malloc.pdf`. [Online, accessed 16-December-2009].

[7] P.-H. Kemp and R. N. M. Watson. Jails: Confining the omnipotent root. `http://phk.freebsd.dk/pubs/sane2000-jail.pdf`. [Online, accessed 17-December-2009].

[8] R. Kettlewell. C Language Gotchas. `http://www.greenend.org.uk/rjk/2001/02/cfu.html`. [Online, accessed 16-December-2009].

[9] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 30th International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.

[10] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://www.suse.de/~krahmer/no-nx.pdf`, September 2005. [Online, accessed 17-December-2009].

[11] Linux Man Page Project. capabilities - overview of linux capabilities. `http://linux.die.net/man/7/capabilities`. [Online, accessed 29-January-2010].

[12] P. A. Loscocco and S. D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[13] S. McClure. *Hacking Exposed*, chapter Hacking Unix. McGraw-Hill Professional, 6th edition, March 2009.

[14] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[15] Postfix. Postfix Overview — Security. `http://www.postfix.org/security.html`. [Online, accessed 16-December-2009].

[16] J. Rauch. Introduction to Linux Capabilities and ACLs. `http://www.securityfocus.com/infocus/1400`. [Online, accessed 16-December-2009].

[17] Sun Microsystems. System Administration Guide: Solaris Containers - Resource Management and Solaris Zones. `http://dlc.sun.com/pdf/817-1592/817-1592.pdf`. [Online, accessed 8-February-2010].

[18] J. P. van Hoogeveen. Working with Solaris Containers and the Solaris Service Manager. `http://www.sun.com/blueprints/0506/819-4328.pdf`. [Online, accessed 17-December-2009].

[19] I. van Sprundel. Fuzzing: Breaking software in an automated fashion. In *22nd Chaos Communication Congress*. Chaos Computer Club, December 2005.

[20] Wikipedia. Buffer overflow protection — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Buffer_overflow_protection&oldid=333698084`.

[21] Wikipedia. Linux — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Linux&oldid=332101608`.

[22] Wikipedia. Unix — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Unix&oldid=331924360`.

[23] Wikipedia. Von Neumann architecture — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Von_Neumann_architecture&oldid=330479010`.