Prozessinjektion

Debugging mit Ptrace

Seminarausarbeitung

ausgearbeitet von

Lennart Hein, Maurice Happe, Kaywan Katibeh

3012079, 2961658, 2960091

vorgelegt an der

Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Informatik IV

Arbeitsgruppe für IT-Sicherheit

im Studiengang

Informatik (B.Sc.)

Erstprüfer:	Prof. Dr. Michael Meier Universität Bonn
Zweitprüfer:	Dr. Matthias Frank Universität Bonn
Betreuer:	Dr. Felix Jonathan Boes Universität Bonn

Bonn, 10. Mai 2019

Kurzfassung

Prozessinjektion bezeichnet die Technik, mit der ein Code in ein laufendes Programm eingefügt werden kann. Ein Debugger benutzt diese Methode um andere laufende Programme zu manipulieren. Das Ziel der Projektgruppe ist der Bau eines Debuggers für Linux mit dem Fokus auf leichter Verständlichkeit des Codes. Die Hauptfunktionen des Debuggers sind folgende: Schrittweise Abarbeitung des Codes, Haltepunkte setzen und Speicheradressen manipulieren. All diese Funktionen werden mit Hilfe des Systemaufrufs `sys_ptrace` in C implementiert. Des Weiteren wird die Windows Funktion CreateRemoteThread, die bereits unter dem Namen „Jugaad“ nach Linux 32-Bit portiert wurde, für 64-Bit kompatibel gemacht. Jugaad ermöglicht das Erstellen eines neuen Threads in einen bestehenden Prozess.

Inhaltsverzeichnis

1 Einleitung 1

2 Bau eines Debuggers 2
- 2.1 Benötigtes Vorwissen . . . 2
  - 2.1.1 Systemaufrufe in Linux . . . 2
  - 2.1.2 Linker und Loader . . . 2
  - 2.1.3 Global Offset Table und Procedure Linkage Table . . . 3
  - 2.1.4 Address Space Layout Randomisation (ASLR) . . . 4
  - 2.1.5 POSIX Signale . . . 5
  - 2.1.6 ptrace . . . 6
- 2.2 Debugger Kernaufgaben . . . 7
  - 2.2.1 Speicher auslesen und verändern . . . 7
  - 2.2.2 Code injizieren . . . 8
  - 2.2.3 Breakpoint . . . 9
  - 2.2.4 Schritt für Schritt - Singlestep . . . 11
  - 2.2.5 Signale senden . . . 12
  - 2.2.6 Stackframe anzeigen . . . 12
- 2.3 Debugger Implementation und API . . . 13
  - 2.3.1 API . . . 13
  - 2.3.2 Debugger-Server . . . 15
  - 2.3.3 Front End . . . 15
  - 2.3.4 Minimum Working Example . . . 16

3 Jugaad - CreateRemoteThread für Linux 18
- 3.1 CreateRemoteThread . . . 18
- 3.2 Speicher reservieren mit mmap . . . 19
- 3.3 Prozesse erschaffen mit clone . . . 20
- 3.4 Jugaad . . . 20
  - 3.4.1 CreateRemoteThread Umsetzung in Linux . . . 21
  - 3.4.2 Shellcodes . . . 22
- 3.5 Jugaad 64 Bit . . . 24
  - 3.5.1 Shellcodes in 64 Bit . . . 24
  - 3.5.2 Probleme mit Jugaad . . . 27
  - 3.5.3 jugaad2.0 . . . 27

4 Stand und Ausblick 29
4.1 Stand des Debuggers 29
4.1.1 Disassemblieren 29
4.1.2 Auflösen von Symbolen 29
4.1.3 Parsen von Opcodes 29
4.1.4 Step in und Step over 29

Literaturverzeichnis 31

Abbildungsverzeichnis 33

1 Einleitung

Maurice Happe,
Kaywan Katibeh,
Lennart Hein

Im Rahmen der Projektgruppe Prozessinjektion ist der Bau eines Debuggers für 64 Bit Linux Systeme gefordert. Ein Debugger soll dem Nutzer die Möglichkeit geben beliebig Haltepunkte zu setzen, um den Code stückweise auszuführen. Ein Debugger muss dem Nutzer jegliche Adressen und Register ausgeben und diese auch verändern können. All diese Funktionen werden durch den Systemaufruf `sys_ptrace` ermöglicht. C ist eine maschinennahe Sprache und somit optimal für das Erstellen eines Debuggers geeignet. Der Fokus der Projektgruppe liegt auf gut lesbarem und dokumentierten Code. Im Vergleich mit Vorbildern eines Debuggers wie dem gdb, ist dieser Debugger nicht über eine Konsole bedienbar, sondern über eine GUI. Die Oberfläche ist mit NodeJS realisiert und über jeden Browser verfügbar.

2 Bau eines Debuggers

In diesem Kapitel werden die wichtigsten Methoden zur Implementation der Funktionen eines Debuggers vorgestellt. Das Hauptwerkzeug hierfür ist die C-Funktion `ptrace`, die den Systemaufruf `sys_ptrace` verwendet.

2.1 Benötigtes Vorwissen

In diesem Abschnitt wird das wichtigste Vorwissen zum Bau eines Debuggers beschrieben. Das Verständnis von Systemaufrufen (2.1.1), der Funktionsweise von Loadern (2.1.2) und ASLR (2.1.4), POSIX-Signalen (2.1.5) sowie des Systemaufrufs `sys_ptrace` (2.1.6) werden folgend beschrieben.

2.1.1 Systemaufrufe in Linux

Maurice Happe

Systemaufrufe sind die Schnittstelle zwischen Kernel-Mode (privilegierten Modus) [Man13] und User-Mode (eingeschränkter Zugriff). Sie ermöglichen es Programmen, die nur User-Mode-Rechte haben, Systembefehle auszuführen. Das erfolgt, indem das User-Mode-Programm die Kontrolle an den Kernel übergibt. Der 64-Bit Linux Kernel folgt hierbei der „System V AMD64 ABI“ Konvention. Der Wert in `RAX` bestimmt den genauen Systemaufruf [Cha12]. Sechs weitere Register (`RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9` sind für optionale Parameter reserviert [Fre18e]. So benötigt der Systembefehl `sys_close` einen weiteren Parameter, um dem Kernel mitzuteilen, welches Objekt geschlossen werden soll. Der Kernel überprüft, ob das ausgeführte Programm die Rechte zu dem Systemaufruf besitzt und gibt bei Misserfolg eine Fehlermeldung zurück. Bei Erfolg legt der Kernel einen Rückgabewert in `RAX` ab und gibt die Kontrolle wieder an das Programm im User-Mode zurück. Mit `sys_ptrace` lassen sich die Werte in `RAX` und den Parameter-Register auch vor Beginn des Systemaufrufs noch ändern. Bei Verlassen des Systemaufrufes lässt sich der Rückgabewert in `RAX` ändern. Genauere Details zu `sys_ptrace` werden in 2.1.6 erklärt.

2.1.2 Linker und Loader

Kaywan Katibeh

Nachdem ein Programm kompiliert wird, folgt die Load-Time, in der der Linker und Loader ausgeführt werden. Die Aufgabe des Linkers ist es abstrakte Namen mit den zugehörigen konkreten Adresswerten zu verbinden. Wenn zum Beispiel eine Variable X verwendet wird, so verbindet der Linker den Adresswert, in dem sich X befindet, mit der Variabel X. Nach dem selben Prinzip werden, Funktionsaufrufe mit der aufgerufenen Funktion verbunden [Cha08, Kap 4]. Wenn die

Funktion printf aufgerufen wird, dann wird sie mit der Startadresse der Funktion printf in der libc assoziiert. Zudem fügt der Linker fragmentierte Code Abschnitte zu einem zusammen. [MM12]

Der Loader lädt das Programm und verwendete Bibliotheken in den Arbeitsspeicher, sodass sich die Code Abschnitte nicht überlappen [Lev99]. Im folgenden werden 3 verschiedene Arten von Loadern vorgestellt[Dou06]:

Beim Absolute Loading werden nur absolute Adressen geladen. Das bedeutet, dass zum Beispiel bei einem JUMP X der Adresswert von X mit der absoluten Adresse referenziert wird. Der Nachteil dieser naiven Art ist, dass vorrausgesetzt wird, dass das Programm unverändert an der selben Stelle im Speicher liegt. Die Ladezeit ist dadurch aber minimal kürzer. Die Adresswerte werden während der Kompilierung geladen. [Dou06]

Relocatable Loading wertet die Adresswerte relativ zu dem Programm aus. Ein JUMP X in Adresszeile N, dass auf die Zeile $N + 100$ referenziert wird mit JUMP 100 abgekürzt. Somit kann das Programm unabhängig im Speicher liegen. Dennoch darf intern im Code nichts verändert werden, weil die berechneten Offset Werte nicht mehr korrekt sein könnten. Die Adresswerte werden während der Ladezeit geladen. [Dou06]

Das Dynamic Runtime Loading bindet die Adresswerte erst bei der Ausführung. Bei einem JUMP X wird der relative Offset und die Startadresse des Programms im Speicher addiert und somit die phsyische Adresse berechnet. Der Vorteil beim Dynamic Runtime Loading ist, dass das Programm noch vor Ausführung intern verändert werden kann und, dass es unabhängig im Speicher liegt. Ein Nachteil hierbei ist, dass die Ladezeit länger ist. [Dou06] Code der mit Dynamic Runtime Loading geladen wird nennt man Position Independent Code (PIC) 2.2.3

2.1.3 Global Offset Table und Procedure Linkage Table

Maurice Happe

Die Global Offset Table (GOT) ist eine Liste mit den Startadressen von globalen Variabeln und Funktionen. Wenn ein Programm zum Beispiel die Funktion printf verwendet, so muss die Startadresse von printf in libc bekannt sein. Wenn die Startadresse nicht im Programm vorliegt, wird nach Anfrage von der GOT die absolute Adresse zurückgegeben. Die GOT liegt im Data-Segment des Codes, da die GOT je vorhandene globale Variable und Funktion nur eine Anfrage bearbeiten muss. Im Text-Segment müsste je Referenz auf eine globale Variable oder Funktion eine Anfrage bearbeitet werden. [MM12] Die GOT ist insofern sehr wichtig, da die Adresswerte für jeden Prozess mit ASLR (2.1.4) randomisiert werden.

Bei großen Programmen, können die GOT-Anfragen viel Zeit in Anspruch nehmen. Daher gibt es die Methode Lazy Binding, die eine Anfrage nur dann durchführt, wenn im aktuellen Programmfluss die jeweilige globale Variable oder Funktion gebraucht wird. So kann zum Beispiel auf die Einbindung von Error-Handling-Bibliotheken verzichtet werden.

Die Umsetzung des Lazy Bindings erfolgt durch die Procedure Linkage Table (PLT). Die PLT liegt im Data-Segment und wird statt der GOT als erstes aufgerufen [Ben11]. Wenn bei der Ausführung des Codes eine Startadresse benötigt wird, so wird zuerst die PLT angefragt und wenn die PLT die Startadresse nicht gespeichert hat, dann fragt die PLT die GOT nach der Startadresse. Diesen Prozess nennt man auch „trampoline“, da die Anfragenbehandlung von der PLT zur GOT übergeht

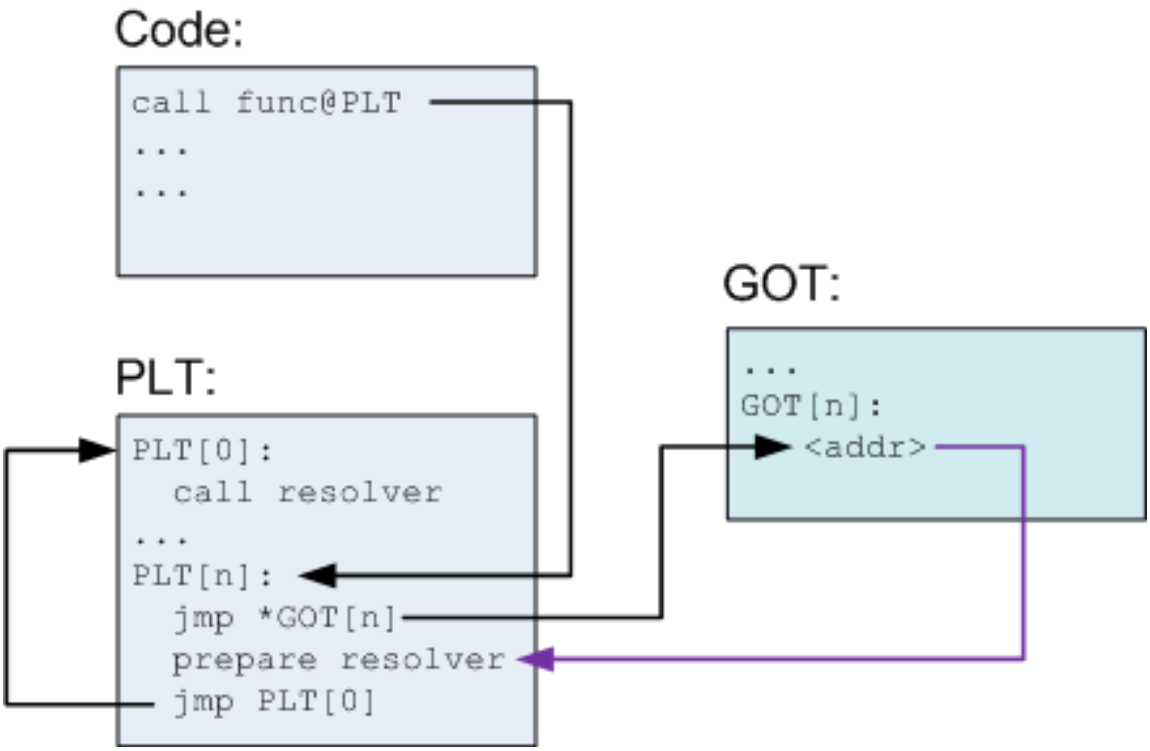

Abbildung 1: *Kommunikation zwischen Code, PLT und GOT [Ben11]*

und daraufhin wieder zurück auf die PLT (siehe Abbildung 1). Wenn dieselbe Startadresse erneut abgefragt werden soll, so liegt sie bereits in der PLT und hat eine kürze Behandlungszeit der Anfrage. [MM12]

Im Beispiel eines Hello World Programms (siehe Abbildung 2) wird die Funktion `printf` aufgerufen (in Zeile 400534). Statt des Adresswertes von der Funktion `printf`, wird die PLT referenziert. In der aufgerufenen Zeile in der PLT 400400 wird dann die GOT aufgerufen und die absolute Adresse für `printf` angefragt. Diese wird in der PLT abgespeichert und wenn in Zeile 400543 `printf` erneut aufgerufen wird, liegt der Adresswert bereits in der PLT.

2.1.4 Address Space Layout Randomisation (ASLR)

Lennart Hein

Address Space Layout Randomisation (ASLR) bezeichnet eine Technologie, durch die das Layout des virtuellen Speichers eines Prozesses beim Start zufällig zugewiesen wird [MGR14]. Durch ASLR wird das Speicherlayout so randomisiert, dass auch durch das Begutachten des Maschinencodes die Adressen der Funktionen, Variablen und Instruktionen bei der Ausführung nicht mehr ermittelt werden können, die Auflösung von Symbole erfolgt erst zum Programmstart. Weil auf Linuxsystemen ASLR jeden Prozess neu randomisiert, wird ASLR auch als „per process randomisation“ bezeichnet. [MGR14]

Im folgenden Abschnitt werden Beispielangriffe genannt vor denen ASLR schützt. Shellcode Injection bezeichnet Angriffe, bei denen ein bösartiges Payload in Form eines Shellcode auf den Stack abgelegt wird, und die Rücksprungadresse so modifiziert wird, dass die CPU den Shellcode ausführt. Dies ist nur möglich, da die klassische von Neumann Architektur keine Trennung von

```
Disassembly of section .plt:

00000000004003f0 <printf@plt-0x10>:
  4003f0:       ff 35 12 0c 20 00       pushq  0x200c12(%rip)        # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  4003f6:       ff 25 14 0c 20 00       jmpq   *0x200c14(%rip)        # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  4003fc:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400400 <printf@plt>:
  400400:       ff 25 12 0c 20 00       jmpq   *0x200c12(%rip)        # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
  400406:       68 00 00 00 00          pushq  $0x0
  40040b:       e9 e0 ff ff ff          jmpq   4003f0 <_init+0x28>

Disassembly of section .plt.got:

0000000000400420 <.plt.got>:
  400420:       ff 25 d2 0b 20 00       jmpq   *0x200bd2(%rip)        # 600ff8 <_DYNAMIC+0x1d0>
  400426:       66 90                   xchg   %ax,%ax

0000000000400526 <main>:
  400526:       55                      push   %rbp
  400527:       48 89 e5                mov    %rsp,%rbp
  40052a:       bf d4 05 40 00          mov    $0x4005d4,%edi
  40052f:       b8 00 00 00 00          mov    $0x0,%eax
  400534:       e8 c7 fe ff ff          callq  400400 <printf@plt>
  400539:       bf e0 05 40 00          mov    $0x4005e0,%edi
  40053e:       b8 00 00 00 00          mov    $0x0,%eax
  400543:       e8 b8 fe ff ff          callq  400400 <printf@plt>
  400548:       90                      nop
  400549:       5d                      pop    %rbp
  40054a:       c3                      retq
  40054b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

```c
#include <stdio.h>
void main ()
{
    printf("Hello World");
    printf("Hello again");
}
```

Abbildung 2: *Hello World dissassembiliert (gekürzt)*

Programm und Daten vorsieht. Um solchen Angriffen nun vorzubeugen, ist das NX-Bit eine mittlerweile weit verbreitete Gegenmaßnahme. Hierbei werden Teile des virtuellen Speichers mit Flags versehen, die der CPU indizieren sollen, dass Code aus diesen Speicherbereichen nicht ausgeführt werden darf. Eine Abwandlung der Shellcode Injection sind die Return to libc Angriffe. Bei diesen wird die Rücksprungadresse nicht mit der Adresse eines eigenen Payloads, sondern mit der Adresse einer bereits in den Speicher geladenen Funktion überschrieben. Diese Funktion kann aus einer externen Bibliothek geladen oder aber eine programmeigene Funktion sein. Abhängig von der Architektur können sogar Parameter an diese Funktion übergeben werden, indem etwa bei intel x86 jene auf den Stack gepusht oder geschrieben werden. Das NX-Bit kann einen Return to libc Angriff nicht verhindern, da die Funktion, die letztendlich aufgerufen wird, nicht mit einem NX-Bit versehen ist und die Ausführbarkeit intentional ist. Da ASLR jedoch die Startadressen zufällig zur Laufzeit anpasst, können die Adressen der Zielfunktionen nicht ohne Weiteres ermittelt werden.

2.1.5 POSIX Signale

Maurice Happe

Im folgenden Abschnitt werden POSIX Signale erläutert und warum diese für einen Debugger beachtet werden müssen. POSIX Signale sind Nachrichten, die von einem Prozess zu einem anderen gesendet werden. Diese Nachrichten führen zu Statusänderungen des Empfängers. Signale sind Integer, deren Bedeutungen in der `<signal.h>` definiert sind. Sie werden durch Systemaufrufe oder Bibliotheksfunktion unter Linux aufgerufen. Ein bekanntes Signal ist `SIGSEGV`,welches bei der Benutzung einer Speicherstelle, die nicht vom Prozess reserviert wurde, ausgelöst wird und sofort zum Abbruch des Programms führt. Eine vollständige Liste der POSIX Signale befindet sich auf der Manpage [Fre18d].

Signale, die an einen Prozess geschickt werden, können den Programmfluss des Debuggers beeinflussen. Wird zum Beispiel das Signal `SIGSEGV` an den zu debuggenden Prozess geschickt kommt es zum Absturz des Debugging-Vorgangs. Damit die Integrität eines Debuggers nicht verletzt wird, sollten Signale durch einen Signal Handler im Debugger abgefangen werden. Details zum Signal Handler befinden sich im Abschnitt 2.2.5. Sobald ein Signal von einem Prozess erfasst wird, wird die dazugehörige Funktion ausgelöst. Ein Signal Handler ermöglicht es die, bei Ankunft der Signale ausgelösten, Funktionen neu zu definieren. Signale können durch den Debugger an den zu debuggenden Prozess weitergereicht oder ignoriert werden. Der Linux Debugger GDB verfügt über einen Signal Handler, der den Umgang mit den relevantesten Signalen neu definiert. Mit dem Befehl `info signals` in GDB erhält man eine Liste, in welcher der Signal Handler des Linux Debuggers definiert ist.[SPS+88]

2.1.6 ptrace

Kaywan Katibeh

Der `sys_ptrace` Systemaufruf ermöglicht dem Tracer, einem Process, die Kontrolle und die Observation eines Tracees, einem anderen Prozess. Dies wird durch eine Parent-Child Beziehung zwischen Tracer als *Parent* und Tracee als *Child* realisiert [Pado2].

Die C-Funktion `ptrace` ist in Linux in der `<sys/ptrace.h.>` deklariert. Die Funktionssignatur ist wie folgt:

```
long ptrace (                       \\ return value
    enum _ptrace_request request,   \\ PTRACE-Variante
    pid_t pid,                      \\ Prozess Id des Tracee
    void *addr,                     \\ Pointer eines Offset
    void *data,                     \\ Pointer auf eine Speicherstelle
);
```

Die Parameter haben folgende Bedeutung:

- `enum _ptrace_request request`: Bestimmt die genauere Ausführung der Funktion, damit auch die Rückgabe der Funktion. Die Manpage enthält eine ausführliche Auflistung aller möglichen Werte [Fre18c].
- `pid_t pid`: Gibt die Prozess ID des Ziels der Funktion an.
- `void *addr`: Beschreibt eine Speicheradresse im Speicherbereich des Tracee.
- `void *data`: Beschreibt einen Speicheradresse im Speicherbereich des Tracer.
- `return value`: Bei einem error wird der Return-Wert auf -1 gesetzt. Beim erfolgreichen Ausführen der Funktion erhält man eine 0 oder vom Parameter 1 abhängigen Wert, die im Abschnitt 2.2.1 näher erläutert wird.

Tracer und Tracee

Lennart Hein

Es gibt zwei Möglichkeiten die Initialisierung zu beginnen.

1. Möglichkeit: Initialisierung durch Tracee mit PTRACE_TRACEME:

```
pid_T child;
child = fork();                         // Tracer forkt seinen eigenen Prozess
if(child == 0)
{
    ptrace(PTRACE_TRACEME, 0, 0, 0);    // Der Child führt den ptrace Aufruf aus
    exec(argv[1], argv + 1);            // Child startet das gewünschte Programm
}
else
{
    wait(0);                            // Parent wartet auf Interrupt des Child
}
```

2. Möglichkeit: Initialisierung durch Tracer mit PTRACE_ATTACH:

```
ptrace(PTRACE_ATTACH,tracee_pid,NULL,NULL);
```

Anders als bei PTRACE_TRACEME ermöglicht PTRACE_ATTACH die Initialisierung bei bereits laufenden Programmen. Beide Varianten führen dazu, dass der Prozess mit der angegebenen ProcessID (PID) zum Tracee des Tracers wird. Dabei wird das Signal SIGSTOP zum Tracee geschickt, dadurch wird der Tracee gestoppt und der Tracer erhält die Kontrolle. Anschließend kann sys_ptrace mit anderen Parametern aufgerufen werden, um den Tracee zu debuggen. Details zu den Möglichkeiten folgen in Abschnitt 2.2. Da der Tracee gestoppt ist, muss er durch das Signal SIGCONT forgesetzt werden. Der Befehl für das Fortsetzen ist:

```
ptrace(PTRACE_CONT,tracee_pid,0,0);
```

Damit der Tracer zu einem weiteren fest gewältem Zeitpunkt des Programmflusses vom Tracee weitere Befehle ausführen kann, wird erneut eine STOP-Bedingung für den Tracee benötigt. Diese wird zum Beispiel durch einen Breakpoint ermöglicht. Die Realisierung eines Breakpoints durch sys_ptrace wird in Abschnitt 2.2.3 erläutert. Wird der Debugger nicht mehr benötigt, kann man den Childprozess terminieren oder man beendet die Parent-Child-Beziehung zwischen Tracer und Tracee durch den Aufruf:

```
ptrace(PTRACE_DETACH,tracee_pid,0,0);
```

2.2 Debugger Kernaufgaben

Ein Debugger muss gewisse Basisfunktionen besitzen, damit er effektiv benutzt werden kann. Im Folgenden werden die wichtigsten Methoden erklärt.

2.2.1 Speicher auslesen und verändern

Maurice Happe

Um einen laufenden Prozess besser zu untersuchen, ist es für den Benutzer hilfreich Speicheradressen und Register auslesen zu können. Zudem ist es hilfreich sein, diese zu verändern. Mit PTRACE_PEEKUSER werden einzelne Registern des Childprozessen ausgegeben. Somit ist auch das Nachvollziehen der Systemaufrufe des Childs möglich. [Fre18c]

```
uint64_t value = ptrace(PTRACE_PEEKUSER, pid, addr, 0);
```

Mit der Funktion PTRACE_PEEKTEXT kann der Wert in einer Adresse oder einem Register gelesen werden. Da in Linux der Text- und Code-Abschnitt nicht getrennt sind, sind PTRACE_PEEKTEXT und PTRACE_PEEKDATA äquivalent. Der Rückgabewert kann dann ausgelesen und verändert werden. [Fre18c]

```
1 uint64_t value = ptrace(PTRACE_PEEKTEXT, pid, addr, 0);
```

Mit PTRACE_GETREGS werden mehrere Register ausgelesen, die alle in einen struct kopiert werden. Der struct user_regs_struct ist in der <sys/user.h> definiert. [Fre18c]

```
1 ptrace(PTRACE_GETREGS, pid, 0, &struct);
```

Mit PTRACE_POKETEXT kann der Speicher bearbeitet werden. Auch hier sind PTRACE_POKETEXT und PTRACE_POKEDATA äquivalent. PTRACE_POKEUSER erlaubt das Verändern von Registerwerten. Alle drei Methoden können mit der gleichen Syntax benutzt werden. [Fre18c]

```
1 ptrace([PTRACE_POKETEXT/POKEDATA/POKEUSER], pid, addr, long_val);
```

Der vierte Parameter ist die Bytesequenz, mit welcher die Speicherzellen, auf die Parameter Drei verweist, überschrieben wird. Aufbauend auf den genannten Methoden lässt sich auch Code injizieren [2.2.2] und Breakpoints setzen [2.2.3].

2.2.2 Code injizieren

Kaywan Katibeh

Der Debugger sollte ebenfalls in der Lage sein, Code in das Tracee-Programm einzufügen und somit den Programmablauf zu verändern. Angenommen der Benutzer möchte vor *byte*1 (siehe 3.1) Code injizieren. Ist der injizierte Code von der Länge N, so werden die nächsten $N+1$ Bytes ab *byte*1 und der RIP mit Hilfe von PTRACE_PEEKDATA gespeichert. Ein möglicher C-Code sieht wie folgt aus:

```
1 uint64_t* backup = malloc(shellcode_len);
2 for(int i = 0; i < shellcode_len; i+=8)
3 {
4   backup[i] = ptrace(PTRACE_PEEKDATA, pid, addr + i, NULL);
5 }
```

Nun werden die gesicherten Bytes durch PTRACE_POKEDATA mit dem gewünschten Code überschrieben (siehe 3.2).

```
6 for(int i = 0; i < shellcode_len; i+=8)
7 {
8   ptrace(PTRACE_POKEDATA, pid, addr + i, code[i]);
9 }
```

Zusätzlich wird ein weiteres Byte mit 0xcc überschrieben. Dieses Byte löst einen Breakpoint aus, aber dazu mehr in 2.2.3. Der Instructionpointer (RIP) liest nun den eingefügten Code aus (siehe 3.3). Erreicht der RIP das Ende, so unterbricht er am Breakpoint und der Tracer lädt den ursprünglichen Wert des RIP vom Backup. Die gesicherten Bytes werden ebenfalls geladen und der Code wird auf den alten Stand zurückgesetzt (siehe 3.4). Der Programmverlauf wird daraufhin normal fortgesetzt. [SKK+01]

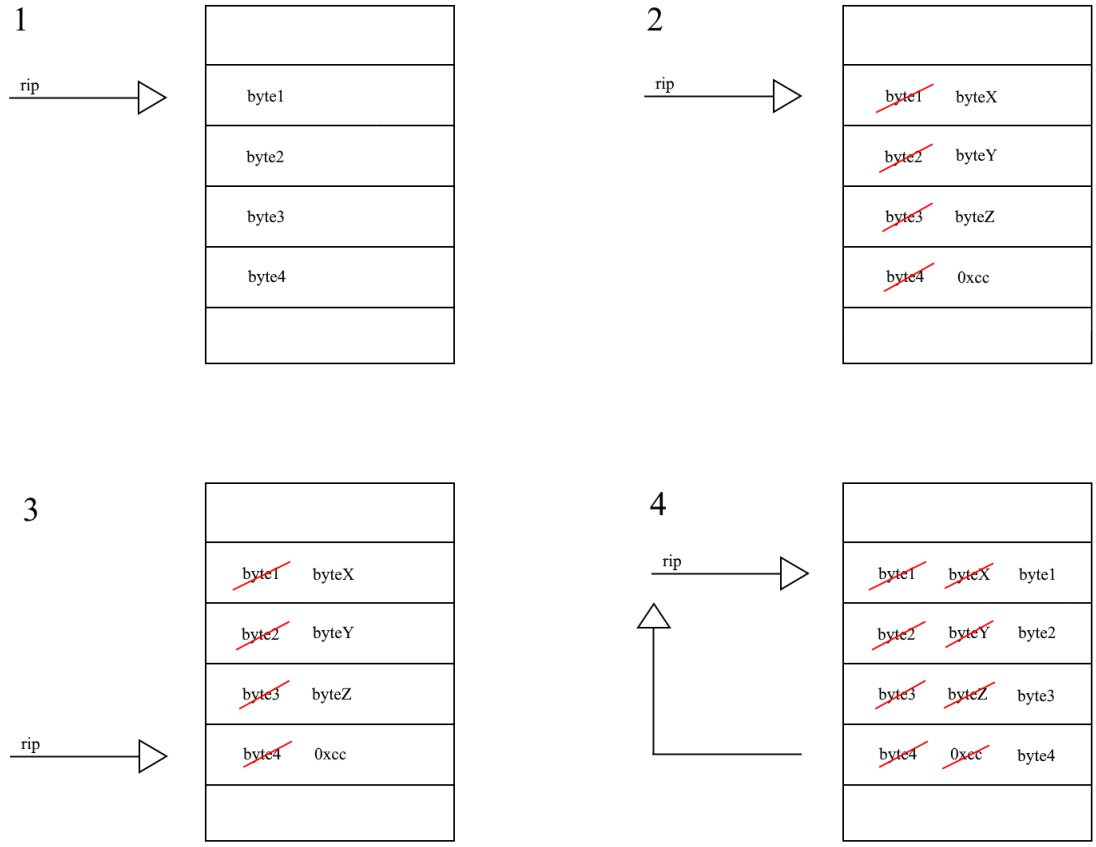

Abbildung 3: *Stackansicht für Codeinjektion*

Auf diese Weise lässt sich jeder ausführbarer Code in ein Tracee-Programm einfügen. Ein Anwendungsbeispiel für Codeinjektion ist `set_breakpoint`, das in 2.2.3 genauer beschrieben wird.

2.2.3 Breakpoint

Lennart Hein

Eine der wichtigsten Funktionen eines Debuggers ist das Setzen von Breakpoints. Diese Funktion erleichtert die Fehlersuche beim Debugging durch Abarbeitung des Programmflusses in logischen Abschnitten. Damit ein Programm einen Breakpoint ausführt, muss die Abfolge der Instruktionen des Programms unterbrochen werden. Eine „Trapfunktion“ muss also in das laufende Programm eingefügt werden. Eine mögliche Trapfunktion ist:

```
1 0xcc        // int 3 'Call to Interrupt'
```

Dieser Opcode sorgt bei Ausführung dafür, dass das Programm unterbrochen wird, und die Kontrolle and den Parentprozess übergeben wird. Das Einsetzen des Opcodes `0xcc` ist analog zum Vorgehen in 2.2.2. Da `set_breakpoint` nur einen Breakpoint, also den Opcode `0xcc`, setzt und keinen weiteren Code einfügt, ist die Anzahl eingesetzter Bytes eins. Nach Lesen der Breakpoint-Zeile wird der Programmfluss an den Tracer übergeben und der Benutzer hat die volle Kontrolle. Der folgende Code ist ein Beispiel für das Setzen und Entfernen eines Breakpoints, zur Vereinfachung wird die Existenz von anderweitigen Softwareinterrupts ignoriert.

```
uint8_t trapcode[8] = {0xcc, 0x90, 0x90, 0x90
                       0x90, 0x90, 0x90, 0x90};     // Padding mit NOPs auf 64Bit
backup = ptrace(PTRACE_PEEKDATA, pid, addr_ptr, 0); // Backup erstellen
ptrace(PTRACE_POKEDATA, pid, addr_ptr, trapcode);   // Trapcode hineinladen
ptrace(PTRACE_CONT, pid, 0, 0);                     // Tracee Trapcode ausführen
wait(0);                                            // Warten auf Breakpoint
ptrace(PTRACE_POKEDATA, pid, addr_ptr, backup);     // Backup laden
ptrace(PTRACE_POKEUSER, pid, RIP, addr_ptr);        // RIP zurücksetzen
```

Hierbei wird ein Breakpoint an der Stelle `addr_ptr` gesetzt. Der Breakpoint wird entfernt, indem das ursprüngliche Byte aus dem Backup an die Stelle `addr_ptr` geladen wird und der *RIP* auf das erste Byte der urspünglichen Instruktion gesetzt wird.

Ein Debugger soll mehrer Breakpoints setzten können. Jeder eingefügte Breakpoint wird deshalb in eine Liste im Tracer mit dem zugehörigen Backup und der Adresse `addr_ptr` gespeichert. Eine Liste sichert die Eindeutigkeit eines Breakpoints anhand seiner Adresse und ermöglicht das Ausgeben und Löschen der gesetzten Breakpoints. Bei einem ausgelösten Softwareinterrupt wird der aktuelle `RIP` des Tracee, abzüglich des ausgeführten Bytes `0xcc`, mit den Adressen der gesetzten Breakpoints verglichen. Falls es sich bei dem Interrupt um einen Breakpoint handelt, wird das entsprechende Backup geladen und der `RIP` dekrementiert.

Breakpointverwaltung

Zum Setzen von Breakpoints gehört auch eine Verwaltung, mit der eingesehen werden kann, an welcher Stelle sich Breakpoints befinden. Deswegen werden bei der Erstellung von Breakpoints in einer Liste die Adresse und der ursprünglichen Code an der Adresse eingetragen. Somit können nicht nur alle Breakpoints angezeigt werden, sondern so kann auch von einem Breakpoint `0xCC`-Interrupt und einem natürlichen Interrupt unterschieden werden. Zusätzlich können durch die Verwaltung auch Breakpoints gelöscht werden. Dafür wird `0xCC` mit dem alten Wert überschrieben und der Breakpoint aus der Liste gelöscht.

Probleme durch Address Space Layout Randomisation (ASLR)

Maurice Happe

Da für das Setzen von Breakpoints die Adresse, an jener sich der gewünschte Maschinenbefehl befindet, angegeben werden muss, muss der Debugger oder der Benutzer den Maschinencode interpretieren. Durch das ASLR ist die virtuelle Adresse bei der Ausführung des Debugee aber wahrscheinlich eine andere als die statisch Betrachtete. Für den Debugger ist es ohne Weiteres nicht möglich, auf die Adressen zur Laufzeit zu schließen. Das Benutzen eines eigenen Loaders, der auf Adressverwürfelung verzichtet, ist möglich, aber würde den Rahmen der Projektgruppe sprengen.

Deaktivierung von ASLR

Kaywan Katibeh

ASLR kann deaktiviert werden, dies ist durch den folgenden, mit Root-Rechten ausgeführten, Bashbefehl möglich:

```
echo '0' > /proc/sys/kernel/randomize_va_space
```

Für den Benutzer stellt dies aber möglicherweise ein Sicherheitsrisiko dar.

Lösung durch Vermeiden von Position Independent Code (PIC)

Lennart Hein

Position Independent Code (PIC) bezeichnet Code, bei dem alle Adressen nicht absolut, sondern relativ zum Programm Counter angegeben werden. [MGR14] Dadurch können Libraries einfach und effizient verwendet werden, ohne aufwändig zum Zeitpunkt des Ladens alle Verweise zu modifizieren. Alle Verweise auf geladenene Libaries werden durch die Global Offset Table (GOT) aufgelöst. Anders als PIC Libaries verfolgen Position Independent Executables (PIE) lediglich sicherheitsbezogene Ziele. PIE sind Anwendungen, bei denen auch der Anwendungscode positionsunabhängig kompiliert wurde. [MGR14] Dies führt, abgesehen von der erhöhten Sicherheit, lediglich zu einem Performance-Overhead. ASLR wird aber nur dann von dem Linux-Kernel eingesetzt, wenn die entsprechenden Sektionen position independent kompiliert wurden. [MGR14] Um nun also ASLR zu umgehen, kann das zu debuggende Programm als Position-Dependent-Executable kompiliert werden. Das Verwenden von PIC Libraries stellt kein Problem für die Anwendung des Debuggers dar, da die Basisadresse der Bibliothek aus der GOT entnommen, und damit die absoluten Adressen errechnet werden können. Um mit Gnu Compiler Collection(GCC) eine Anwendung als PIE zu kompilieren, wird folgender Befehl genutzt:

```
1 gcc input.c -no-pie
```

Lösung durch Errechnen der Adressen mit Hilfe der /proc/pid/maps

Die File `/proc/[pid]/maps` gibt Auskunft über die Sektionen im virtuellen Speicher des Prozesses mit der ID `pid`. [Fre18b] Über die Inhalte der `maps`-File können die Adressen im virtuellen Speicher errechnet werden. Der Vorteil liegt in der Kontingenz der Kompilierung als Position Dependent Code.

2.2.4 Schritt für Schritt - Singlestep

Maurice Happe

Wie in 2.2.3 erklärt, lässt sich ein Programm abschnittsweise abarbeiten, indem Breakpoints gesetzt werden. Aber wenn der Benutzer sein Programm ganz kleinschrittig abarbeiten möchte, so müssen jeweils einzelne Breakpoints gesetzt werden. Daher gibt es den Singlestep-Mode, mit dem der Benutzer auf das manuelle Breakpoint-Setzen verzichten und seinen Code kleinschrittig bearbeiten kann.

Der `sys_ptrace` Systemaufruf bietet zwei Methoden an, mit denen Singlestep-Mode umgesetzt werden kann. `PTRACE_SINGLESTEP` ermöglicht es Instruktionen einzeln auszuführen. [Fre18c]

```
1 ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
```

`PTRACE_SYSCALL` springt vor den nächsten Systemaufruf oder verlässt diesen, wenn er bereits im Systemaufruf ist. [Fre18c]

```
1 ptrace(PTRACE_SYSCALL, pid, 0, 0);
```

Mit `PTRACE_SYSCALL` lassen sich zum Beispiel die Registerwerte vor und nach einem Systemaufruf beobachten oder sogar manipulieren. Ausnahme beim Manipulieren vor dem Systemaufruf ist der `ORIG_RAX`, der die Nummer des Syscalls beinhaltet und `RAX`, der den Rückgabewert des Syscalls gespeichert hat. Bei Ausführung von `PTRACE_SINGLESTEP` und `PTRACE_SYSCALL` erhält der Tracee einen `SIG-TRAP` um zu signalisieren, dass die Kontrolle an den Tracer übergeben wird.

Intuitiver Singlestepper

Kaywan Katibeh

Die intuitive Vorstellung eines Singlesteps ist die Ausführung einer Instruktion oder eines Systemaufrufs, doch sys_ptrace ermöglicht lediglich entweder das Abwarten eines Systemaufrufs mit PTRACE_SYSCALL oder einer Instruktion verschieden von Systemaufrufen mit Hilfe von PTRACE_SINGLESTEP. Um nun den intuitiven Singlestep zu implementieren, müssen beide Typen einen Interrupt auslösen. Zwei Lösungsansätze existieren: Bei Verwendung von manuellen Breakpoints sind diese nach jeder Instruktion einzufügen, jedoch kommt diese Variante ohne die Verwendung von PTRACE_SYSCALL und PTRACE_SINGLESTEP aus, dafür muss Parsing der Opcodes durchgeführt werden, um die Anzahl der Bytes der bevorstehenden Instruktion, und damit die Position des nächsten Breakpoints zu bestimmen. Viel einfacher zu implementieren erscheint das Betrachten des nächsten Opcodes: Falls die nächsten 16 Bit der Instruktionen mit `0x0F 0x05` übereinstimmen, ist die nächste Instruktion PTRACE_SYSCALL. PTRACE_SYSCALL muss dementsprechend zweimal aufgerufen werden, damit der Syscall betreten und verlassen wird. Zusätzlich muss nach Breakpoints geprüft werden. Die nächste Instruktion wird mit `0xCC` (2.2.3) verglichen und wenn der Vergleich positiv ist, dann wird der Programmfluss gestoppt und der `0xCC` wird mit dem ursprünglichen Code an der Stelle überschrieben. Wenn der folgende Opcode weder `0x0F 0x05` noch `0xCC` ist, so wird PTRACE_SINGLESTEP ausgeführt.

2.2.5 Signale senden

Lennart Hein

Wenn ein Prozess ein Signal erhält, mit Ausnahme von SIGKILL, wählt der Kernel willkürlich einen Thread aus, der das Signal bearbeitet. Handelt es sich um den Tracee, der ein Signal erhält, so wird dieser in einen signal-delivery-stop Mode versetzt.[Fre18d] Das Signal wird nicht direkt zum Tracee geschickt. Der Tracer hat vorher die Möglichkeit diese Signale zu bearbeiten oder zu unterdrücken. Bei einem Debugger wird dies durch den Signal Handler vorgenommen, der in Abschnitt 2.1.5 kurz erläutert wurde. Damit der Tracer Informationen über das Signal, welches den Stopp verursacht hat, erhält, kann folgender Befehl verwendet werden:

```
1 | ptrace(PTRACE_GETSIGINFO,0,siginfo_t)
```

Dabei ist siginfo_t die Speicherstelle einer Struktur, in welcher das vom Tracee gespeicherte Signal kopiert wird.

Signale können zum Tracee mit folgendem Befehl gesendet werden:

```
1 | ptrace(PTRACE_CONT,pid,0,sig)
```

In sig wird der Integer des Signals weiter an den Tracee übergeben. Setzt man sig auf 0 wird kein Signal an den Tracee weitergeleitet und der Tracee setzt seinen Prozess fort.

2.2.6 Stackframe anzeigen

Maurice Happe

Der Stackframe ist der Arbeitsbereich des aktuellen Programmaufrufs. Hier befinden sich lokale Variablen und auf von der Prozedur auf den Stack gepushte Elemente. Das Betrachten des Layouts und der Inhalte hat Anwendungsbeispiele insbesondere in der IT Sicherheit im Kontext von Stack Overflow Angriffen. [SPS+88] Um den Stackframe anzuzeigen wird mit Hilfe der Techniken aus 2.2.1 die entsprechenden Bytes ausgelesen. Um die Grenzen des aktuellen Stackframe zu

bestimmen, werden RSP und RBP ermittelt. Dabei ist der Wert von RSP die unterste Adresse des aktuellen Stackframes, und RBP-8 ist die oberste. Falls der aktuelle Stackframe leer ist, liegt die obere Grenze 8 Bytes unter der Unteren. Hierbei ist zu Beachten, dass nur durch Begutachtung des Stackframes nicht klar ist, welche Elemente auf dem Stackframe liegen. Wenn eine Variable auf den Stack gepusht wird liegt der Wert auf dem Stack, aber beim Stackframe Anzeigen ist nicht direkt klar um welche Variable es sich handelt.

2.3 Debugger Implementation und API

Lennart Hein

Um die Funktionsweise eines Debuggers praktisch aufzuzeigen, wurde dieser in C implementiert [HKH19]. Dabei wurde strikt zwischen zwei Teilen unterschieden: einer Benutzeroberfläche und dem eigentlichen Debugger, der die Kernaufgaben implementiert. Dabei dient der Debugger als Server, der Befehle von der Benutzeroberfläche, also dem Client, annimmt und diese durchführt. Diese Befehle entsprechen den Debugger Kernaufgaben.

Die Unterteilung erfolgt aus mehreren Gründen: das Hauptziel dieser Arbeit besteht darin, die Konzepte der Funktionsweise eines Debuggers möglichst verständlich zu präsentieren, durch die Aufteilung ist der Debugger frei von Code zum Abhandeln und Überprüfen von Benutzereingaben und Formatierung der Ausgaben. Zudem soll das Programm leicht erweiterbar sein, durch die entwickelte Schnittstelle ist es möglich, die Funktionen des Debuggers für andere Benutzeroberflächen zu verwenden.

2.3.1 API

Lennart Hein

Die API stellt die Schnittstelle zwischen dem Debugger-Server und den Clients, also Benutzeroberflächen oder hardcoded Programmen, die immer gleiche Debuggingoperationen durchführen, dar.

Für die Kommunikation zwischen Client und Server wird die Zero Message Queue (ZMQ) verwendet. Die ZMQ ist eine Bibliothek für Nachrichtenaustausch. Es existieren dabei Bibliotheken für eine vielzahl von Programmiersprachen, und bildet eine Abtraktion jenseits von Kodierungen. Der Vorteil der ZMQ liegt in der einfachen Handhabung, insbesondere bei der Verwendung von verschiedenen Programmiersprachen. Dies erleichtert das Entwickeln von GUIs, etwa in JavaScript. Für dieses Projekt wird ein REQ-REP Socket verwendet: hierbei gibt es einen Server und einen Client die jeweils abwechselnd Nachrichten senden. Dieser Socket ist das simpelste Modell, und deshalb zur Einfachheit gewählt.

Die Abbildung 4 zeigt die Prozedur des Servers in Hinblick auf die API. Zunächst sendet der Client die Direktive des zu debuggenden Programms, der Server antwortet bei Gelingen mit 'START'. Nach der Initialisierung des Servers kann dieser durch den Client kontrolliert werden. Der Client sendet über ZMQ Strings, die einen bestimmten Befehl darstellen, etwa PEEK_REG. Nachdem der Server die Befehle des Clients empfängt, startet der Server die entsprechende Routine, weitere Kommunikation über ZMQ ist je nach Befehl unterschiedlich definiert. In der Routine werden die Befehle des Clients über sys_ptrace am Tracee durchgeführt, eventuelle Rückgabe an den Client erfolgt auch durch ZMQ. Dabei wird in jedem Fall ein String durch den Server versendet, dies

beinhaltet 'EXIT' oder 'RETURN', gefolgt von eventuellen Rückgabewerten. 'RETURN' signalisiert dabei das Anhalten des Tracee, nun können weitere Debugger Kernaufgaben ausgeführt werden. 'EXIT' signalisiert die Beendigung des Tracee, etwa nach einen CONTINUE ohne weiteren gesetzten Breakpoints. Falls von dem Client kein Befehl gesendet wird, sondern 'EXIT', wird der Server, und mit ihm auch der Tracee, beendet.

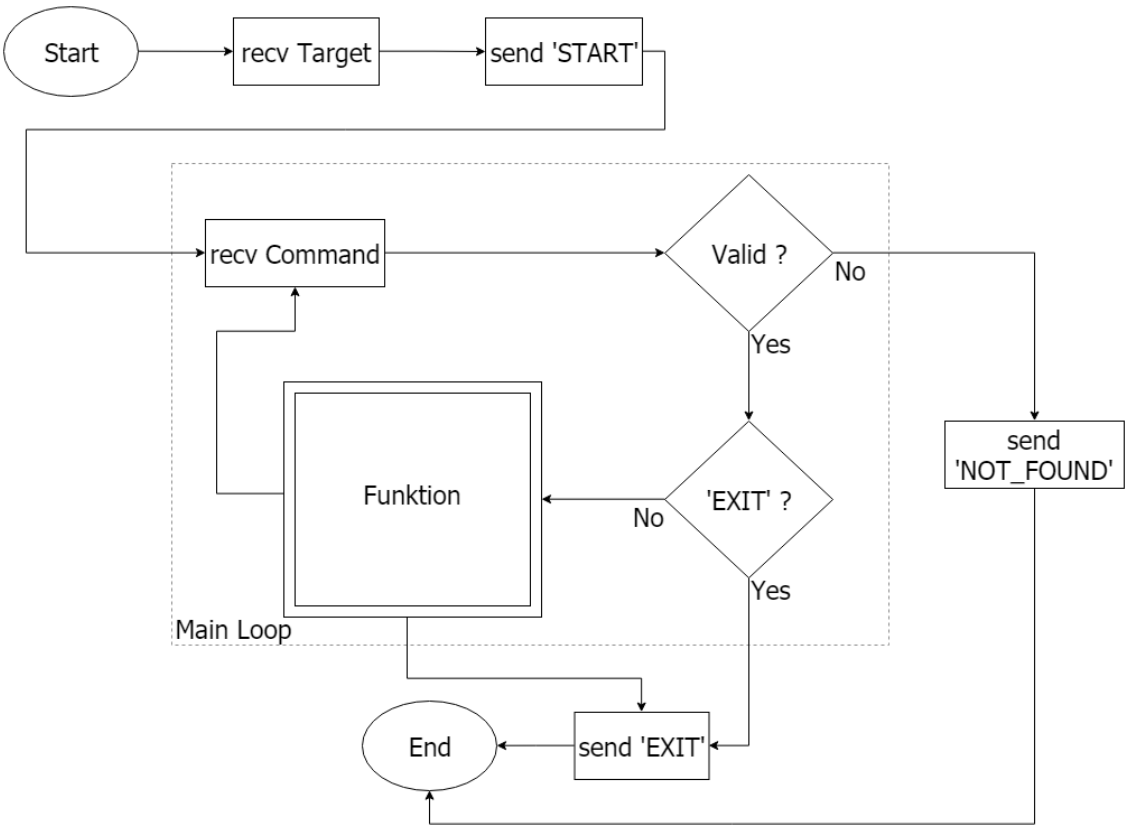

Abbildung 4: *Server bietet API an*

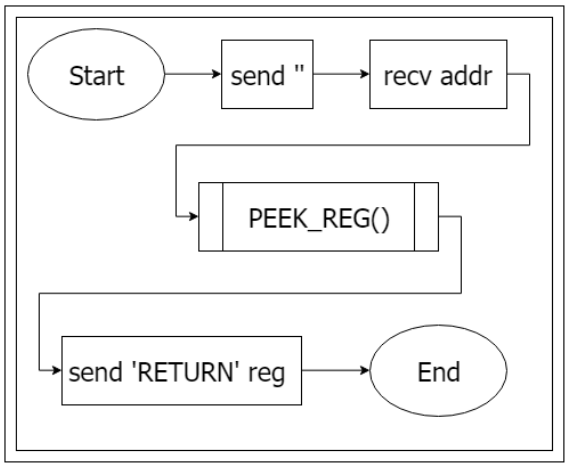

Abbildung 5: *Ablauf von PEEK_REG*

Beispielhaft wird nun PEEK_REG, wie durch 5 beschrieben, betrachetet: Hierbei sendet der Server zunächst einen leeren String, dies ist durch das Modell der ZMQ bedingt, Nachrichten müssen immer abwechselnd von Client und Server geschickt und empfangen werden. Danach sendet der Client die Adresse des gewünschten Registers. In der gewrappten Kernaufgabe wird der Inhalt des Registers bestimmt, und anschließend zusammen mit dem oben beschriebenen RETURN verschickt.

2.3.2 Debugger-Server

Lennart Hein

Der Server kapselt die Kernaufgaben des Debuggers. Außerdem stellt er einen Server, über welchen der Client die Schnittstelle verwenden kann. Als Pseudocode lässt sich die Funktionalität des Servers wie folgend darstellen:

```
1 network_init();          // Einrichten der ZMQ
2 tracee_init();           // Start des Tracee und PTRACE_TRACEME
3 loop:
4     receive_command();   // Erhalten des Befehls über ZMQ
5     execute_command();   // Durchführen des Befehls unter Zuhilfenahme von sys_ptrace
6 goto loop;
7 destroy();               // Beenden des Tracee und der ZMQ
```

Der Server forkt während der Initialisierung und das Child ruft PTRACE_TRACEME auf. Anschließend wird der Programmcode des Childs mit Hilfe von execvp() mit dem des Tracee überschrieben und pausiert. Dies ist analog zu 2.1.6. Die Kernaufgaben des Debuggers sind in jeweils eigenen Funktionen gekapselt, diese werden dann aus einer der Wrappermethoden aufgerufen, die sich um die ZMQ Kommunikation kümmern. Dies führt dazu, dass die eigentlichen Methoden komplett auf das Debuggen mit ptrace konzentriert sind, was die Verständlichkeit erleichtern soll.

Hardcoded-Client

Lennart Hein

Für die Verwendung des Debugger-Servers ist nicht zwingend eine Benutzeroberfläche in Form einer GUI oder CLI notwendig. Es kann auch ein hart gecodeter Client geschrieben werden. So kann ein solcher Client den Server anweisen, bestimmte Kernaufgaben durchzuführen. So lässt sich etwa ein Programm schreiben, welches alle Systemaufrufe des Tracee traced:

```
1 network_init();                        // Einrichten der ZMQ
2 loop:
3     rax      = req_peek_reg(RAX);      // Speichert Rückgabewert des Systemaufrufs
4     orig_rax = req_peek_reg(ORIG_RAX); // Speichert Nummer des Systemaufrufs
5     print(rax, orig_rax);              // Ausgabe
6     req_next_syscall();                // Warten auf den nächsten Systemaufruf
7 goto loop
8 destroy();                             // Beenden der ZMQ
```

2.3.3 Front End

Maurice Happe

Zu einem funktionierendem Debugger gehört ein Anwendungsfenster in Form einer GUI (graphical user interface) oder CLI (command line interface). Ziel dabei ist, dass der Debugger einfach zu bedienen ist.

Für den Debugger war ursprünglich eine GUI geplant. Diese sollte in javascript und html mit NodeJS kompiliert werden. Der Vorteil darin wäre, dass es nicht nur anschaulich sondern auch mit Buttons leicht zu bedienen wäre. Es wurde sich aber gegen eine GUI entschieden, da das Senden von Variabeln innerhalb GUI nicht einwandfrei funktionierte.

Demnach wurde sich für eine CLI entschieden, die in C geschrieben wird. Die Schnittstelle zwischen der CLI und dem Debugger wird mit ZeroMQ über Sockets realisiert. Beim Ausführen der CLI wird geforkt und das child startet dann den Debugger, der dann wiederum den Debuggee startet. Der CLI muss beim Ausführen der Name des Debugee als Parameter mitgegeben werden (genauere Details zum Ausführen siehe readme.md). Sobald alles gestartet ist, wird dem Benutzer die Kontrolle und eine Liste alle Befehle übergeben. Die Eingabe der Befehle ist nicht case-sensitive und die Eingabe der Adressen muss in Dezimalzahlen erfolgen. Bei der CLI ist zu beachten, dass diese nicht optimal ist, aber ausreichend zur Bedienung des Debuggers. Es werden nicht alle ungültigen Eingaben abgefangen und wenn ein Register beim mehrfachen Ausführen von `singlestep` beobachtet werden soll, muss nach jedem singlestep `peek_reg` erneut eingegeben werden.

Die CLI ist an manchen Stellen umständlich zu Bedienen, dennoch gibt es dem Benutzer volle Kontroller über den Debugger und reicht aus um die implementierten Funktionen des Debuggers anzuwenden.

2.3.4 Minimum Working Example

Maurice Happe

Ein Hauptziel des Debuggers ist, dass er leicht zu bedienen ist. Dennoch kann es helfen ein Anwendungsbeispiel zu sehen für noch leichteres Verständnis.

Als Minimum Working Example wird ein Assembler-Programm namens „test_tracee“(siehe 6) verwendetverwendet. Dieses Programm befindet sich ebenfalls auf dem Gitlab [HKH19] und benötigt zusätzlich nasm vorinstalliert. Um die CLI korrekt auszuführen, muss sie mit `./cli.sh` kompiliert und mit `./cli ./test_tracee` ausgeführt werden.

Als Beispiel wird angenommen, dass der Syscall in Zeile 30 korrekt ausgeführt wird und danach mit einem syscall `exit` das Programm beendet. Zu Beginn kann mit `peek_reg` der `RIP` ausgelesen werden um herauszufinden an welcher Adresse der Programm-Code beginnt. Für den weiteren Verlauf müssen die Adresszeilen der folgenden Instruktionen bekannt sein. Diese können mit objdump angezeigt werden. Um den nächsten Syscall aufzurufen, kann mit `next_syscall` in den nächsten Systemaufruf gesprungen werden. Nun kann mit `poke_reg` der Instruction Pointer `RIP` zurückgesetzt werden. In dem Beispiel wird er auf 4194512 gesetzt. Die Adresseingabe erfolgt nur über Dezimalzahlen und nicht Hexadezimalzahlen. Die exakte Position hierbei ist nicht relevant. Wichtig ist, dass es vor dem Syscall im Code-Bereich liegt und die Adresse auf eine gültige Instruktion zeigt. Da das Programm immer noch in einem Syscall entry ist, muss der Programmfluss fortgesetzt werden. In diesem Beispiel soll nach dem `sys_write` aber ein `sys_exit` folgen. Innerhalb eines Syscall entrys kann der `RAX` nicht mehr verändert werden (2.2.4), also muss das Programm erst fortgesetzt werden und an einem Breakpoint (2.2.3) vor dem Syscall angehalten werden. Breakpoints werden mit `create_breakpoint` gefolgt von der exakten Adresse in Dezimalzahlen als neue Eingabe. Mit `continue` kann der Programmfluss nun fortgeführt werden.

```
now init...sending target... done

peek_reg
Type the register you want to peek in
rip
128: 4194507

next_syscall
syscall entry

poke_reg
Type the register you want to overwrite
rip
Type the data you want to overwrite
4194512
Poke succesful

create_breakpoint
Type the address
4194517
breakpoint set

continue
tracee continued

poke_reg
Type the register you want to overwrite
rax
Type the data you want to overwrite
60
Poke succesful

continue
```

```
23  _start:
24
25      ; sys_write
26      mov RAX, 1
27      mov RDI, 1          ; stdout
28      mov RSI, str
29      mov RDX, 12
30      syscall
```

Abbildung 6: *Minimum Working Example*

Die Systemaufruf-Nummer für `sys_exit` ist 60. `sys_exit` benötigt keine weiteren Parameter, also können die Hilfsregister ignoriert werden. Mit `poke_reg` wird der `RAX` auf 60 gesetzt. Der `sys_exit` ist vorbereitet und kann mit `continue` ausgeführt werden.

3 Jugaad - CreateRemoteThread für Linux

CreateRemoteThread ist eine Funktion unter Windows, mit der ein Thread in einen Prozess injiziert werden kann. Die Funktion ist unter Linux standardmäßig nicht verfügbar, aber das Programm Jugaad ist eine Implementierung von CreateRemoteThread für Linux 32-Bit. Ein weiteres Ziel der Projektgruppe ist es Jugaad für 64-Bit verfügbar zu machen, aber dafür muss erst ein Verständnis für die Funktion CreateRemoteThread und deren Implementation in Jugaad vorhanden sein.

3.1 CreateRemoteThread

Maurice Happe

Mit der Windowsfunktion CreateRemoteThread lässt sich ein Thread in den virtuellen Speicher eines anderen Prozesses ausführen [Cen18]. Der Funktion können bis zu sieben Parameter übergeben werden. Die Funktionssignatur sieht wie folgt aus:

```
1 HANDLE CreateRemoteThread(
2   HANDLE                 hProcess,
3   LPSECURITY_ATTRIBUTES  lpThreadAttributes,
4   SIZE_T                 dwStackSize,
5   LPTHREAD_START_ROUTINE lpStartAddress,
6   LPVOID                 lpParameter,
7   DWORD                  dwCreationFlags,
8   LPDWORD                lpThreadId
9 );
```

Die Parameter sind im Folgenden definiert.

- `hProcess` ist ein Handle zu dem Prozess, in dem der Thread eingefügt werden soll. Für den Handle ist wichtig, dass er Rechte hat um einen Thread zu erstellen, Prozessinformationen auslesen kann, auf dem Prozess schreiben, lesen und Operationen ausführen kann.
- `lpThreadAttributes` ist ein Pointer auf Sicherheitsattribute für die definiert ist, ob ein Child den Handle erben kann. Wenn der Wert `NULL` ist, wird der Default-Wert angenommen und der Handle kann nicht vererbt werden.
- `dwStackSize` definiert die Stackgröße in Bytes. Ist der Wert 0, so wird ein Default-Wert angenommen.
- `lpStartAdress` ist ein Pointer auf die Funktion `LPTHREAD_START_ROUTINE`, welche auch als Startadresse des Threads gilt. Die Funktion muss, daher in dem Prozess sein.
- `lpParameter` ist ein Pointer zu dem Parameter, der dem Thread zugewiesen wird.

- `dwCreationFlags` ist ein Flag mit dem der Thread gestartet wird. Bei 0 wird der Thread nach Erstellung direkt ausgeführt. Wenn der Thread in einem suspended State gestartet wird, so wird er direkt gestoppt und kann mit einer Funktion fortgeführt werden.
- `lpThreadId` ist ein Pointer zu einem Thread Identifikator. Wenn der Wert 0 ist, dann wird keine ThreadId ausgegeben.

Der Rückgabewert ist der Handle zu dem neuen Thread. Bei einem Fehler wird aber *NULL* zurückgegeben. Der Thread hat daraufhin Zugriff auf alle Objekte die von seinem Prozess geöffnet werden. Wenn mit `lpThreadAttributes` ein Sicherheits-Deskriptor referenziert wird, so hat der Thread nicht unbegrenzten Zugriff und es muss jede Aktion überprüft werden. Das verlangsamt den Programmfluss des Threads, aber gewährt erhöhte Sicherheit. [Cen18]

CreateRemoteThread kann genutzt werden um beim Debugging einen Breakpoint zu simulieren. Jedoch ist es nicht empfohlen, da eine „single-threaded application“in eine „multithreaded application“umgewandelt werden würde [Cen18]. Zudem wird auch das Speicherlayout durch den Remote Thread verändert [Cen18]. CreateRemoteThread kann beim Debugging nützlich sein, aber dem Benutzer müssen die Konsequenzen bewusst sein.

3.2 Speicher reservieren mit mmap

Lennart Hein

Auf Linux-Systemen lässt sich mit `sys/mman.h` die C-Funktion `mmap` verwenden. Es ermöglicht die Reservierung von Arbeitsspeicher. Mmap hat folgende Signatur[Fre18a]:

```
1 void *mmap(void *addr,
2            size_t length,
3            int prot,
4            int flags,
5            int fd,
6            off_t offset);
```

- `void *addr` ist ein Pointer für eine präferenzierte Startadresse des Zielspeichers. Es ist möglich auch `NULL` anzugeben. Dadurch übergibt man dem Kernel keine Präferenz für die Startadresse für des neue Mapping.
- `size_t length` gibt die Länge des zu reservierenden Speicher an
- `int prot` ist ein Parameter für die sicherheitsrelevanten Eigenschaften des Speichers. Er definiert ob der Speicher ausgelesen, verändert und/oder ausgeführt werden darf.
- `int flags` ist ein optionales Feld für Flags, durch die definiert ist ob der Mapping-Bereich privat, eingeschränkt oder offen für andere Prozesse bzw. Threads ist.
- `int fd` definiert den File Descriptor
- `off_t offset` ist der Offset einer file welcher im File Descriptor geöffnet ist.[Fre19b].

`mmap()` erstellt ein neues Maping mit dem Inhalt eines `files`, der im File Descriptor geöffnet ist. Der Inhalt des Files wird ab dem `offset` bis zu `offset + length -1` übertragen. Der neue zugewiesene virtuelle Adressraum kann mit dem Caller der `mmap()` Funktion geteilt werden. [Fre19b] Eine neue Referenz der `file` wird erstellt, die erhalten bleibt, selbst wenn der File Descriptor geschlossen wird [Fre19b].

Bei Erfolg ist der Rückgabewert die Startadresse des reservierten Speicherbereiches und bei Misserfolg wird MAP_FAILED, was als (void *)-1 definiert ist, zurückgegeben [Fre18a]. Untersucht man die C-Funktion malloc() mit dem Linux Debugging Programm strace erkennt man, dass diese Funktion mit einem Syscall mmap realisiert wurde. Die Funktion munmap() ist vergleichbar mit free(), da es den allokierten Speicher befreit. Wenn der Prozess terminiert, so wird der Speicher automatisch befreit [Fre18a]. mmap() bietet aber auch eine Möglichkeit einen Speicherbereich privat zu allozieren, der ohne Referenzen im Caller realisiert wird. Diese spezielle Funktionalität wird für Jugaad benötigt und wird in Abschnitt3.4.1 erneut aufgegriffen.

3.3 Prozesse erschaffen mit clone

Lennart Hein

Unter Linux lässt sich mit #define _GNU_SOURCE und #include <sched.h> die C-Funktion clone verwenden. Clone erschafft einen neuen Thread der eine vom Nutzer definierte Funktion ausführt[Fre19a]. Die Signatur von clone ist folgende:

```
1 int clone(int(*fn)(void *),
2         void *child_stack,
3         int flags,
4         void *arg)
```

- int(*fn)(void*) ist ein Funktionspointer einer Funktion, welcher der erschaffene Prozess ausführen soll
- void *child_stack ist ein Pointer auf eine Speicheradresse, die als Stack für den Thread verwendet wird
- int flags spezifiziert das Verhältnis zwischen Parent und Child
- void *arg sind Argumente die der Nutzer dem Thread vor der ausführung übergeben kann

Da der Thread den Virtuellen Aressraum mit dem aufrufenden Prozess teilt benötigt der Thread einen eigenen Stack. Auf allen Linux Distributionen wächst der Stack nach unten. Das Argument void *child_stack beinhaltet deswegen die höchste Adresse der jeweiligen Speicherregion. Ein Child_Stack kann mit mmap() erschaffen werden. Die an clone übergeben Adresse aus der mmap Funktion ist dann *Rückgabe von mmap()* + Stackgröße - 1.

Bei erfolgreichem ausführen wird die Thread ID des Childs zurück gegeben und der Child Thread wird ausgeführt. Bei einem Fehler wird -1 zurück gegeben. Die C-Funktion clone() wurde mit dem Syscall sys_clone realisiert. Der sys_clone hat von Architektur zu Architektur unterschiedlich reihenfolgen der Paramter. Die Argumente int(*fn)(void*) und void *arg werden nicht in einem Register als Parameter an den Syscall weitergegeben. Der Child führt seinen Prozess ab dem Call des Parents aus.sys_clone wird in Jugaad zum erschaffen des Threads benötigt. Die genau verwendung des sys_clone in Jugaad wird in Abschnitt3.4.2 aufgegriffen[Fre19a].

3.4 Jugaad

Lennart Hein

Jugaad ist eine Linux 32 Bit Implementierung von der Windows Funktion CreateRemoteThread 3.1. Sie wurde 2011 von Aseem Jakhar auf Github veröffentlicht [Jak11a] und ist mit Hilfe der

Funktion ptrace 2.1.6 geschrieben. Die Funktionssignatur für CreateRemoteThread für Linux sieht wie folgt aus:

```
int create_remote_thread_ex(pid_t pid,
                            size_t stack_size,
                            unsigned char *tpayload,
                            size_t tpsize,
                            int thread_flags,
                            int mmap_prot,
                            int mmap_flags,
                            void * bkpaddr)
```

- `pid_t pid` ist die Nummer des Prozesses in den der Thread injiziert wird.
- `size_t stack_size` ist die Größe des Stacks in Bytes.
- `unsigned char *tpayload` ist ein Pointer auf die Payload die eingefügt wird.
- `size_t tpsize` ist die Länge des Payload.
- `int thread_flags` ist ein optionales Feld für flags.
- `int mmap_prot` definiert den Schutz des Speichers.
- `int mmap_flags` gewährt Zugang zu weiteren optionalen Flags für das Mapping.
- `void * bkpaddr` ist ein Pointer auf die Speicherstelle für die Backup-Adresse und wird bei Eingabe von 0 mit einem Default-Wert versehen.

3.4.1 CreateRemoteThread Umsetzung in Linux

Maurice Happe

Für die Umsetzung von CreateRemoteThread auf Unix Systemen muss, wie auf Windows, der Speicher und der Programmcode des Ziel Prozesses zeitweise modifiziert werden. Dafür benötigt das Programm Lese- und Schreibrechte auf dem jeweiligen Prozess. Wenn diese Vorraussetzungen erfüllt sind, dann lässt sich ein Thread in einen Prozess injizieren [Jak11a]. Der Programmfluss von Jugaad wird in den folgenden Zeilen kurz erläutert.

Der Grundbaustein von Jugaad ist Ptrace. Die CreateRemoteThread Funktion beginnt mit einem `PTRACE_ATTACH`, welche die Tracer-Tracee Beziehung zwischen Jugaad und dem Ziel Process initailisiert und den Zielprozess stoppt. In Abschnitt 2.2.2 wurde bereits erläutert wie eine Codeinjektion mit Ptrace möglich ist. Um Jugaad's CreateRemoteThread auf Linux zu realisieren müssen 3 unterschiedliche Shellcodes in den Zielprozess injiziert und ausgeführt werden. Daher wird zuerst ein Backup der Register und der Speicheraddressen des Injektionsbereichs erstellt. Diese Backups sind notwendig, da dieser Bereich im Verlauf überschrieben wird.

Der erste Shellcode wird in den Prozess geladen und ausgeführt. Dieser vergrößert den reservierten Speicher des Prozess im Arbeitsspeicher für unseren neuen Thread. Dieser Speicherbereich wird als Stack für unseren Thread verwendet. Beim ausführen des Shellcodes erhalten wir als Rückgabe die Addresse dieses Stacks für unseren dritten Shellcode. Der zweite Shellcode erweitert den reservierten Speicher um den benötigten Speicher für den Threadcode und der Thread Payload. Dieser wird an der selben Stelle des ersten Shellcodes eingefügt und ebenfalls ausgeführt. Dies beendet die Vorbereitungen um den Threadcode laden zu können. Der dritte Shellcode beinhaltet einen Clone-Syscall und die Payload, die ausgeführt werden soll. Dieser Shellcodes wird dieses

mal im neu allozierten Bereich des zweiten Shellcodes eingefügt und anschließend ausgeführt. Nachdem die Shellcodes ausgeführt werden, wird der Prozess um einen Thread mit einer vom Benutzer definierten Payload modifiziert. Danach muss nur noch die ursprungliche Funktionalität des Prozesses wiederhergestellt werden um CreatRemoteThread auf Linux vollständig zu realisieren. Dafür werden die zuvor hergestellten Backups wieder zurück an ihrer Urspungsaddressen geladen und die Tracee Tracer Beziehung wird beendet.

3.4.2 Shellcodes

Damit das genaue Verfahren von Jugaad deutlich wird, werden die Shellcodes einzeln, im Detail, im folgenden Abschnitt näher erläutert.

Shellcode 1: Speicher allozieren für Threadstack

Kaywan Katibeh

Der erste Shellcode, in Jugaad, wird an der in CreateRemoteThread übergebenen Adresse geladen. Falls keine Adresse übergeben wird, wird dieser Code an einer von Jugaad vordefinierten `DEFAULT_ADRESSE` geladen. Deswegen wird mit `PTRACE_GETREGS` ein Backup der Register, des Zielprozesses, und mit `PTRACE_PEEKTEXT` ein Backup der Größe `length`, der Inhalt der Adressen, erstellt. Da der erste Shellcode den Threadstack erstellt, handelt es sich hier um einen Syscall der Funktion `mmap()`. Der Shellcode wurde mithilfe des x86 Assembler hergestellt[Jak11b]. Die Werte von `length`, `prot` und `flags` können in Jugaad vom Benutzer optional verändert werden. Dazu werden im Shellcode die Opcodes zum laden der Register mit neuen Werten überschrieben. Der `DEFAULT` sieht die Verwendung von `MAP_PRIVATE` und `MAP_ANONYMOUS` als `flags` und `PROT_READ`, `PROT_WRITE` und `PROT_EXEC` als `prot` vor.

`MAP_PRIVATE` und `MAP_ANONYMOUS` ermöglichen, dass Veränderung dieses erstellten Mappings für andere Prozesse, welche die selbe Datei `mappen`, nicht sichtbar sind und keine Referenzen einer anderen Map auf diese neu erstellten Map verweisen. `PROT_READ`, `PROT_WRITE` und `PROT_EXEC`: erteilen Lese-, Schreib- und Ausführungsrechte der Map.

Eine äquivalentes C-Programm zum Shellcode sieht folgend aus:

```
1 int main()
2 {
3     mmap(0,20000,PROT_EXEC | PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS,
      -1,0);
4 }
```

Nach der Codeinjektion wird der Tracee, welcher durch das `PTRACE_ATTACH` zuvor gestoppt wurde, mit `PTRACE_CONT` fortgesetzt. Dadruch wird der Shellcode ausgeführt und der Prozess führt den Syscall `mmap()` aus. Jugaad muss weiter Shellcodes für einen CreateRemoteThread einfügen. Um unerwartetes Verhalten des Prozesses zu verhindern muss dieser erneut gestoppt werden. Jede Codeinjektion beinhaltet deswegen den `0xcc` Opcode am Ende. Der Tracee wird durch das `SIGTRAP` Signal gestoppt. Zudem wird der Shellcode mit `NOP` Instruktionen erweitert, damit die Größe des Shellcodes, für die `PTRACE` Funktionen, Vielfachen von 4 entsprechen. Beim Syscall `mmap` wird die Rückgabe in das eax Register geladen. Damit behinhaltet das eax Register einen pointer auf den Anfang der Speicheradresse der neu erstellten Map. Sobald Jugaad erneut die Kontrolle erhält wird ein `PTRACE_GETREGS` ausgeführt um die Register nach dem Ausführen des Shellcodes zu erhalten.

Dieser Pointer wird an den dritten Shellcode weitergegeben. Der dritte Shellcode wird direkt modifiziert, weil das Ausführen des zweiten Shellcodes den eax Register überschreiben würde und dadurch die Adresse verloren geht.

Shellcode 2: Speicher allozieren für ThreadPayload und Threadcode

Kaywan Katibeh

Der Shellcode zur Speicherallokation für den Threadshellcode ist in Jugaad fast identisch mit dem ersten Shellcode für den Threadstack. Jugaad setzt hier die selben Paramter für prot und flags ein. Der Parameter length wird aber der Größe des dritten Shellcodes zusätzlich der Größe der Payload angepasst. Beide Shellcodes haben aber dennoch die selbe Größe. Dadurch ergibt sich der Vorteil, dass kein erneutes Backup des Prozesses erstellt werden muss. Der zweite Shellcode wird an der gleichen Adresse wie der erste Shellcode geladen. Mit PTRACE_SETREGS wird der Instruktionspointer zurück auf das erstellte Backup eip gesetzt und der Shellcode kann ausgeführt. Die Rückgabe des Syscall mmap() liegt im eax Register. Mit PTRACE_GETREGS kann auf den Inhalt des eax Registers zugegriffen werden. Dieser Register wird als Adresse in der PTRACE_POKETEXT Funktion bei der Codeinjektion für den dritten Shellcode benötigt.

Alternativ besteht die Möglichkeit den ersten Shellcode 2-mal auszuführen. Dazu muss length hinreichend groß gewählt werden, sodass es jeweils für die Stackgröße und für den dritten Shellcode ausreichend ist. Es werden auch hier 2 individuelle Maps erstellt, aber beide mit identischer Größe. Der Unterschied liegt hier, dass keine Codinjektion für den zweiten Shellcode mehr notwendig ist, sondern der Instruktionspointer zurück auf das Backup geladen werden muss. Dannach ist der Code ausführbereit und es kann wie oben gennant fortgesetzt werden. Jugaad bietet diese Möglichkeit nicht direkt an. Der Code muss hierführ entsprechend verändert werden[Jak11a].

Der Autor Aseem Jakhar spezifiziert den mmap() Shellcode in x86 auf den Vortragsfolien der Defcon 19 [Jak11b]

Shellcode 3: Thread & Payload

Kaywan Katibeh

Der dritte und letzte Shellcode ist der wichtigste Shellcode in Jugaad. Er beinhaltet einen sys_clone und erzeugt damit einen Child Thread des Zielprozesses. Wie bereits in Abschnitt 3.3 erläutert, benötigt der sys_clone einen Childstack, flags und einen Pointer auf eine Child-Funktion. In Abschnitt 3.4.2(Shellcode 1: Speicher allozieren für Threadstack) wurde erläutert wie man eine Adresse einer Speicherregion durch sys_mmap erhalten kann. Der dritte Shellcode wird nach dem gleichen Verfahren wie bei den erste beiden Shellcodes modifiziert. Diesesmal nachdem der erste Shellcode ausgeführt wurde und bevor der zweite Shellcode ausgeführt wird. Die erhaltene Adresse durch den ersten Shellcode übergibt Jugaad an den dritten Shellcode weiter als die höchste Adresse der neuen Speicherregion. Es wird also eax+STACK_SIZE-1 als Pointer für den Childstack gesetzt. Die flags des sys_clone werden auf CLONE_THREAD,CLONE_VM und CLONE_SIGHAND gesetzt.

CLONE_THREAD setzt den erzeugten Child in der selben Thread Gruppe wie der aufrufende Prozess. Außerdem wird Jugaad als Parent des Prozesses gesetzt [Fre19a]. Das Setzen von CLONE_THREAD zwingt seit Linux 2.6.0 den Nutzer auch die flags CLONE_VM und CLONE_SIGHAND zu setzen. CLONE_SIGHAND setzt den gleichen Signal Handler Table vom Parent für den Child. CLONE_VM sorgt dafür, dass der Childprozess im selben Speicherbereich wie der Parentprozess läuft.

Nach dem sys_clone folgt die Payload Funktion, gefolgt von sys_exit. Der Syscall Exit wird am Ende der Payload gesetzt. Endet die Payloadfunktion nach dem Ausführen und somit der Childthread, so wird kein return mehr dem Parentprozess gesendet, welches zu einer Programmfluss-Unterbrechung hätte führen können. Der dritte Shellcode wird nun an der Rückgabeadresse des zweiten Shellcodes geladen. Wird der Syscall ausgeführt so wird ein Childprozess erstellt, der durch einen JMP-Befel zur Payload Funktion weitergeleitet wird.[Jak11a] Leider wird dieser Shellcode in Jugaad vom Autor Aseem Jakhar nicht genauer spezifiziert in x86 dargestellt, wie es beim ersten Shellcode der Fall war. In Abschnitt 3.5 wird aber ein möglicher dritter Shellcode für die 64 Bit implementierung von Jugaad in x86_64 hergeleitet. Während der Childprozess die Payloadfunktion ausführt, erhält der Parentprozess das SIGTRAP Signal durch den 0xcc Opcode, welches auf den den sys_clone

Zum Abschluss von Jugaad werden die Backupadressen zurück geladen

3.5 Jugaad 64 Bit

Kaywan Katibeh

Ein Ziel der Projektgruppe ist die Realisierung von Jugaad auf 64 Bit. Der entscheidende Unterschied einer 64 Bit Implementierung liegt in den verwendeten Registern der Ptrace Funktionen und den Shellcodes.

Rückgabewerte eines mmap() Shellcodes werden zum Beispiel in das rax Register geladen anstatt des eax. Jugaad64Bit muss das berücksichtigen. Der einfachhalthalber haben wir uns dafür entschieden den zweiten Shellcode und den dritten Shellcode zusammen zufassen. Dadurch wird die laufzeit von Jugaad um eine Codinjection verkürzt und der dritte Shellcode wird in seiner Funktionalität verdeutlichter dargestellt.

3.5.1 Shellcodes in 64 Bit

Kaywan Katibeh

Im folgenden Abschnitt wird die Herleitung beider Shellcodes für die 64 Bit Implementierung dargestellt.

mmap()

Der mmap() Shellcode orientiert sich an dem originalen Jugaad Shellcode, welcher in den Vortragsfolien der Defcon19 dargestellt wurde [Jak11b]. Der Syscall wird in x86_64 wie folgt realisiert:

```
1 mov rdi,0           //void* addr
2 mov rsi,STACK_SIZE  //Die größe des Stacks haben wir auf  4096*1026 gesetzt (4Mb)
3
4 mov rdx,PROT_EXEC|PROT_READ|PROT_WRITE
5 mov r10,MAP_PRIVATE|MAP_ANONYMOUS
6
7 mov rax,9           //Syscall Nummer von Sys_mmap()
8 syscall
9 int3                //Debugging interupt Signal
```

In einem mmap_syscall wird das r8 Register für den Filedescriptor und das r9 Register als offset der angegebenen geöffneten Datei im Filedescriptor verwendet. Beide Werte sind für Jugaad

irrelevant. Das Setzen des Flags MAP_ANONYMOUS ignoriert deshalben die Werte in den Register r8 und r9. Auf einige Systemen verlangt die Implementierungen des Syscalls dennoch das Setzen vom r8 Register auf den Wert -1 [Fre18a]. Das rdi Register wird auf 0 gesetzt, da keine präferenzierte Adresse der erstellten Map notwendig ist. Das rsi Register wird mit der definerten STACK_SIZE gesetzt. Beim gesetzten Wert von $4096 * 1026$ handelt es sich um einer vielfachen einer Pagesize des virtuellen Speichers. Dadurch wird die neu erstellte Speicherregion eindeutig von anderen virtuellen Adressen getrennt. Die Werte der unterschiedlichen Prots werden mit einer Bitweisen OR Opertation miteinander verknüpft. Das rdx Register wird nun mit dem Ergebnis dieser Operation geladen. Nach dem selben Verfahren wird das r10 Register mit den Flags geladen. Um die einzelnen Werte der Flags und Prots herauszufinden kann man diesen folgenden C Code nutzen:

```
1 #include <sys/mman.h>
2 int main(){
3  return MAP_PRIVATE | MAP_ANONYMOUS;
4 }
```

Mit Hilfe des Befehls gcc -E file.c erhält man die jeweiligen Werte auf der Konsole. Der x86_64 Assembler code wird mit nasm compeliert. Nach dem Linken 2.1.2 werden die Opcodes des X86_64 Codes mithilfe von Objdump ermittelt. Der Befehl dafür ist: objdump -D mmap -M intel>objdump.txt. Der erstellte Shellcode wird nach dem Opcode 0xcc,dem Trap Signal, um Nop-Instruktionen erweitert.

Threadshellcode

Für die Herleitung des Threadshellcodes haben wir einen Thread in Linux mithilfe eines C Codes erstellt. Mithilfe der C Funktion mmap() und der Funktion clone() soll ein Thread erstellt werden. Mit diesem folgenden C Code kann ein Thread in Linux erstellt werden, welcher eine beliebige Funktion ausführt:

```
1  int payload(void *arg)
2  {
3      do stuff ....
4    exit(0);
5  }
6  void shellcode(int (*fn)(void *)){
7      int *stack_pointer = mmap(NULL,Stacksize,Prots,Flags, -1, 0);
8    clone(fn,stack_pointer+Stacksize,Flags,NULL);
9  }
10 int main()
11 {
12   shellcode(payload);
13   return 0;
14 }
```

Nun gilt es diesen C code in x86_64 Code umzuwanden, damit der Shellcode nur aus Syscalls besteht und keine Referenzen auf die Plt und der GlobaloffsetTabel beinhaltet. Zusätzlich muss der Assembler Code relative adressierung nutzen. Um einen Thread in Linux zu erschaffen wird der sys_clone benötigt. Die Information welcher Register mit welcher Varibale gefüllt werden muss, entnehmen wir folgender System Call Table [Cha12].

Das Ergebins ist folgender Shellcode:

```
call get_stack; mmap shellcode
lea rsi,[rax + STACK_SIZE]
mov rdi, CLONE_THREAD|CLONE_VM|CLONE_SIGHAND
mov rax, SYS_clone
syscall
ret
```

Das bestehende Problem ist,dass nach dem Syscall der aufrufende Thread zurück zum Caller gesetzt wird, aber der neue erstellte Thread auf einen leeren Stack zurück gesetzt wird. Beide Threads haben identische Register bis auf das rax Register, welches beim neuen Thread 0 ist und das rsp Register welcher den gleichen Wert hat wie das rsi Register[Wel15].Es fehlt der Pointer auf die Threadfunktion für den neu erstellten Thread. Der Software Developer Chris Wellons beschreibt auf seinem Blog wie man dieses Problem lösen kann[Wel15]. Die Idee ist den Threadstack um einen Pointer verkleinert an den sys_clone weiterzugeben. An dieser Pointer Stelle zeigt nun das rsi Register. Der Threadfunktionspointer wird dort geladen,wodurch nach dem Syscall die Childfunktion ausgeführt wird. Im folgenden Beispielcode befindet sich der Threadfunktionspointer im rdi:

```
thread_create:
    push rdi
    call get_stack
    lea  rsi,[rax + STACK_SIZE -8]
    pop qword[rsi]
    mov rdi, CLONE_THREAD|CLONE_VM|CLONE_SIGHAND
    mov rax, SYS_clone
    syscall
    ret
get_stack:
    ..;mmap shellcode ohne int3
    ret
```

[Wel15] Da dieser Code in einen laufenden Prozess iniziert werden soll muss der Call auch im Shellcode selbst enthalten sein. Zur Vorbereitung des thread_create wird deshalb in das rdi Register ein jmp befehl zu einer Threadfunktion geladen welche vom neu erstellten Thread ausgeführt werden soll. Es folgt der Call Befehl. Die Rückgabe des sys_clone führt den alten Thread nun zum int3, dem Breakpoint Signal des Tracee für Jugaad, und der neue Thread wird durch den Jump Befehl zur Threadfunktion weitergeleitet. Zum Schluss folgt ein sys_exit, damit das Terminieren der Threadfunktion kein unerwünschtes Fehlverhalten im Programmfluss verursacht. Das Konzept des Shellcodes sieht wie folgt aus:

```
lea rdi,[rel thread_funk]
call thread_create
int3
thread_create:
    Call get_stack
    ...
    ret
get_stack:
    ...
    ret
thread_funk:
    ... ;Payload funktion
    mov rax,60
    syscall
```

Als Payload-/Threadfunktion haben wir ein simples Hello World Programm verwendet. Die einzige Bedingung an die Payloadfunktion die gilt ist, dass sie relative Adressierung verwendet.Der *x*86_64 Code wird mit Nasm compeliert und die Opcodes für den Shellcode werden mit Objdump herausgefunden.

3.5.2 Probleme mit Jugaad

Maurice Happe

Nach dem Ändern der jeweiligen Register von 32 Bit auf 64 Bit in den Injektionsfunktionen und den Backup Funktionen von Jugaad, werden die alten Shellcodes mit den neuen Shellcodes ersetzt. Die zweite Shellcode Injektion vom alten Jugaad muss dabei aus kommentiert werden, da diese nicht mehr benötigt wird. Nach dem Kompilieren und Ausführen von Jugaad auf einem Testprozess kommt es zu einer Speicherschutzverletztung des Zielprozesses. Da Jugaad mit Ptrace arbeitet ist es nicht möglich den Testprozess mithilfe von GDB zu untersuchen. Ein Coredump file kann zur Analyse des Absturzes genutzt werden. Jugaad ist im Verhältnis zu seiner eigentlichen Funktionalität ein großes C-Programm. Zuzüglich zu den Injektionsfunktionen befindet sich ebenfalls ein sehr großer Anteil an Debugging Funktionen, die für den eigentlichen Create Remote Thread keine Relevanz haben. In diesen Debugging Funktionen kommt es zur Manipulation des Tracee Prozesses. Deswegen haben wir die wichtigisten Funktionalitäten von Jugaad in einem seperaten C-Programm neu verfasst. Diese Implementierung basiert auf den Methoden der Foreneinträge von `Linux Infecting Running Processes`[pic16] und Jugaad. Der folgende Abschnitt befasst sich mit dessen genauerer Implemtierung.

3.5.3 jugaad2.0

Lennart

Die neue Implementierung folgt dem gleichen Prinzip wie in Jugaad. Unterschied ist, dass keine Ptrace Funktionen für das Debugging verwendet werden und dadurch direkt nach dem `Ptrace_Attach` ein Backup der Größe des ersten Shellcodes erstellt wird. Die Shellcodes werden nicht in Programm mehr verändert. Es ist nicht möglich neue Flags oder die größe des Stacks zur laufzeit in Jugaad2.0 zu verändern. Der Shellcode müsste dementsprtechend vom Nutzer an diesen Stellen manuell verändert werden.Dabei muss auf die größe des Shellcodes geachtet werden. Diese muss aufgrund von `ptrace` vielfachen von 4 betragen. Die Injektion der Shellcodes findet im Programmfluss des Prozesses statt und nicht an einer beliebigen Adresse des Zielprozesses. Anschließend wird ein Backup der Register mit `PTRACE_GETREGS` erstellt. Das Struct mit den erhal-

tenen Registern wird kopiert. Ein Struct dient als Backup. Das andere Struct wird zur Ausführung der Shellcodes verwendet. Dehalb werden die Relevanten Register für die Syscalls im Programm auf 0 gesetzt bevor der erste Shellcode geladen wird. Nachdem alle Backups generiert und der Shellcode injeziert wurde, wird der `mmap()` Shellcode ausgeführt. Die Rückgabeadresse wird als neu Injektionsadresse für den zweiten Shellcode verwendet. Hier wird kein Backup benötigt. Bevor der zweite Shellcode ausgeführt wird, werden auch hier die Register, bis auf das `rax` auf 0 gesetzt. Zusätzlich wird das `rip` Register gleich gesetzt mit dem `rax` Register, sodass der Tracee den zweiten Shellcode in der neuen Map Region ausführt. Nach dem auch der zweite Shellcode ausgeführt wurde, wird das Backup der Adressen und der Register geladen und Jugaad führt einen `Ptrace_detachte` aus.

4 Stand und Ausblick

In diesem Kapitel wird der aktuelle Stand des Debuggers beschrieben, sowie die geplanten Ergänzungen.

4.1 Stand des Debuggers

Maurice Happe

Die Funktionen des Debuggers sind implementiert und können auf einen beliebigen Tracee angewendet werden. Der Debugger wird über eine CLI gesteuert. Die CLI ist für das Benutzen ausreichend, aber an einigen Stellen noch optimierbar. Dennoch reicht die CLI aus um den Debugger vollständig und korrekt zu bedienen.

4.1.1 Disassemblieren

Kaywan Katibeh

Disassemblieren ist eine Funktion, die Objectfiles in Assembler-Code übersetzt. Der Assembler-Code gewährt dem Benutzer einen detailierteren Einblick in sein Programm. Somit lässt sich der Singlestepper auch leichter verstehen, wenn jeder Schritt einzeln aufgelistet wird. Da das Schreiben einer Disassemblieren-Methode sehr aufwendig ist, ist die Idee, sich an das bereits existierendes Programm „objdump“ zu wenden.

4.1.2 Auflösen von Symbolen

Maurice Happe

Wenn eine Variable in einem Programm zugewiesen wird, wird der Wert in das Data-Segment geschrieben, aber der Name ist beim Auslesen nicht klar. Mit dem Auflösen von Symbolen sollen den Werten im Data-Segment den korrekte Namen zugewiesen werden. Dies ermöglicht eine direkte Ausgabe zu einem angefragten Variabel-Namen.

4.1.3 Parsen von Opcodes

Kaywan Katibeh

Möchte der Benutzer die nächste Instruktion auslesen, so kann er den Wert im `RIP` auslesen. Doch erhält er nur den reinen Opcode. Mit der geplanten Methode ließe sich der Opcode in Assembler-Code parsen und wäre somit verständlich. Eine Implementierung wäre aber sehr umständlich, da eine Tabelle von jedem möglichen Opcode definiert sein muss.

4.1.4 Step in und Step over

Lennart Hein

Funktionen stellen logisch abgegrenzte Teile eines Programms dar. Daher ist es häufig naheliegend, Funktionen durch den Stepper gesondert zu behandeln. Falls die nächste Instruktion ein Funktionsaufruf ist, gibt es zwei Möglichkeiten mit dem Stepper zu verfahren:

Step in

Der nächste Breakpoint soll vor der Ausführung der ersten Instruktion innerhalb der aufzurufenden Funktion sein.

Step over

Die aufzurufende Funktion soll komplett ausgeführt werden, danach soll der Tracer gestoppt werden.

Dies kann durch den folgenden Pseudocode veranschaulicht werden:

```
1  void main()
2  {
3      foo();      // <- RIP aktuell
4      bar();      // <- RIP nach 'Step over'
5  }
6
7  void foo()
8  {
9      baz();      // <- RIP nach 'Step in'
10     qux();
11     return;
12 }
```

Step in ist prinzipiell ein Singlestep, hierbei wird lediglich die nächste Call Instruktion ausgeführt. Step over benötigt jedoch eine neue Funktion des Debuggers: Ein Breakpoint soll genau nach der Call Instruktion im Speicher gesetzt werden, an jene Adresse an die der Program Counter nach der Abarbeitung der Unterfunktion springt. Ähnlich wie bei dem Intuitiven Singlestepper 2.2.4 muss hierfür der nächste Opcode interpretiert werden. Problematisch wird hierfür, dass es eine vielzahl von verschiedenen Call Instruktionen gibt, die unterschiedlich lange Operanden erwarten. [Int] Das Behandeln von allen möglichen Opcodes erscheint im Rahmen dieser Arbeit als zu für Aufwand für die Implementierung des Step Over.

Literaturverzeichnis

[Ben11] Bendersky, Eli: Position Independent Code (PIC) in shared libraries. Website, 2011. – Online erhältlich unter https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/; abgerufen im Februar 2019.

[Cen18] Center, Microsoft Windows D.: CreateRemoteThread function. Website, 2018. – Online erhältlich unter https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread; abgerufen im Februar 2019.

[Cha08] Chattopadhyay, Santanu: System Software. 2008

[Cha12] Chapman, Ryan A.: Linux System Call Table for x86 64. Website, 2012. – Online erhältlich unter http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64; abgerufen im Januar 2019.

[Dou06] Doursat, René: Principles of Operating Systems. Website, 2006. – Online erhältlich unter http://doursat.free.fr/docs/CS446_S06/CS446_S06_3_Memory2.pdf; abgerufen im Februar 2019.

[Fre18a] Free Software Foundation: mmap Manual Page. Website, 2018. – Online erhältlich unter http://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html; abgerufen im Februar 2019.

[Fre18b] Free Software Foundation: Proc manual page. Website, 2018. – Online erhältlich unter http://man7.org/linux/man-pages/man5/proc.5.html; abgerufen im Januar 2019.

[Fre18c] Free Software Foundation: Ptrace manual page. Website, 2018. – Online erhältlich unter http://man7.org/linux/man-pages/man2/ptrace.2.html; abgerufen im Januar 2019.

[Fre18d] Free Software Foundation: Signal manual page. Website, 2018. – Online erhältlich unter http://man7.org/linux/man-pages/man7/signal.7.html; abgerufen im Januar 2019.

[Fre18e] Free Software Foundation: Syscall manual page. Website, 2018. – Online erhältlich unter http://man7.org/linux/man-pages/man2/syscall.2.html; abgerufen im Januar 2019.

[Fre19a] Free Software Foundation: clone Manual Page. Website, 2019. – Online erhältlich unter http://man7.org/linux/man-pages/man2/clone.2.html; abgerufen im Februar 2019.

[Fre19b] Free Software Foundation: Memory-mapped I/O. Website, 2019. – Online erhältlich unter http://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html; abgerufen im Februar 2019.

[HKH19] Hein, Lennart ; Katibeh, Kaywan ; Happe, Maurice: PG Prozessinjektion. https://git.cs.uni-bonn.de/boes/pg_injection1819/tree/cb22e732b666569d5d864c10e674a17e5751fa89, 2019

[Int] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. – Online erhältlich unter https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[Jak11a] Jakhar, Aseem: Jugaad. https://github.com/aseemjakhar/jugaad, 2011

[Jak11b] Jakhar, Aseem: Jugaad Slides from DEF CON 19. PDF, 2011. – Online erhältlich unter https://www.defcon.org/images/defcon-19/dc-19-presentations/Jakhar/DEFCON-19-Jakhar-Jugaad-Linux-Thread-Injection.pdf; abgerufen im März 2019.

[Lev99] Levine, John R.: Linkers and Loaders. Trumansburg : Morgan Kaufmann, 1999

[Man13] Mandl, Peter: Grundkurs Betriebssysteme - Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation. Wiesbaden : Springer Vieweg, 2013

[MGR14] Marco-Gisbert, Hector ; Ripoll, Ismael: On the Effectiveness of Full-ASLR on 64-bit Linux. 2014

[MM12] Michael Matz, Andreas Jaeger Mark M. Jan Hubi ˇcka H. Jan Hubi ˇcka: System V Application Binary Interface - AMD64 Architecture Processor Supplement. 2012. – Online erhältlich unter http://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf; abgerufen im Februar 2019.

[Pad02] Padala, Pradeep: Playing with ptrace, Part I. In: Linux Journal 2002 (2002), Nr. 103, S. 5

[pic16] pico oxoopf: [Linux] Infecting Running Processes. Website, 2016. – Online erhältlich unter https://0x00sec.org/t/linux-infecting-running-processes/1097;abgerufen im März 2019.

[SKK+01] Some, Raphael R. ; Kim, Won S. ; Khanoyan, Garen ; Callum, Leslie ; Agrawal, Anil ; Beahan, John J.: A software-implemented fault injection methodology for design and validation of system fault tolerance. In: Dependable Systems and Networks, 2001. DSN 2001. International Conference on IEEE, 2001, S. 501–506

[SPS+88] Stallman, Richard ; Pesch, Roland ; Shebs, Stan u. a.: Debugging with GDB. In: Free software foundation 675 (1988)

[Wel15] Wellon, Chris: Raw Linux Threads via System Calls. Website, 2015. – Online erhältlich unter https://nullprogram.com/blog/2015/05/15/;abgerufen im März 2019.

Abbildungsverzeichnis

1 Kommunikation zwischen Code, PLT und GOT [Ben11] 4
2 Hello World dissassembiliert (gekürzt) 5
3 Stackansicht für Codeinjektion 9
4 Server bietet API an 14
5 Ablauf von PEEK_REG 14
6 Minimum Working Example 17

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Seminarausarbeitung ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn, 10. Mai 2019

Lennart Hein, Maurice Happe, Kaywan Katibeh