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, 2. März 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

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 [Fre18d]. 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 Bearbeitung des Systemaufrufs noch ändern. Auch 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 Variable 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

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

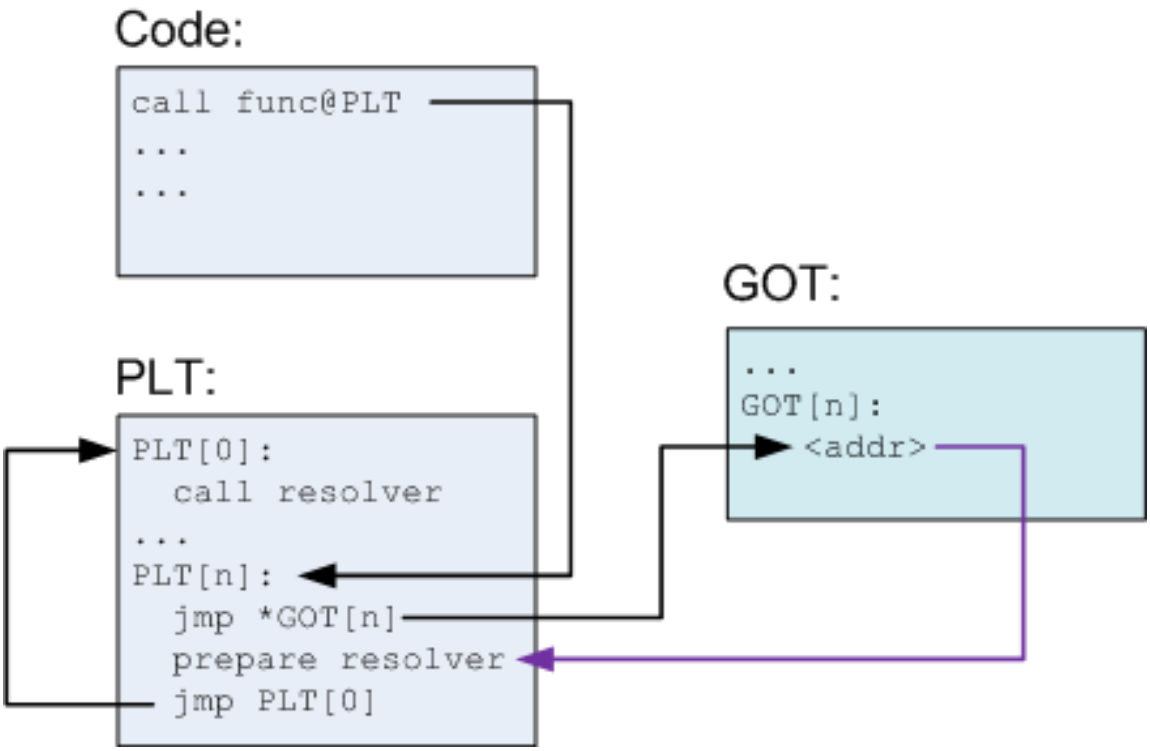

Abbildung 1: *Kommunikation zwischen Code, PLT und GOT [Ben11]*

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 [Fre18c].

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:

```
1 long ptrace (                         \\ return value
2     enum _ptrace_request request,     \\ PTRACE-Variante
3     pid_t pid,                        \\ Prozess Id des Tracee
4     void *addr,                       \\ Pointer eines Offset
5     void *data,                       \\ Pointer auf eine Speicherstelle
6 );
```

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 [Fre18b].
- `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. [Fre18b]

```
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. [Fre18b]

```
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. [Fre18b]

```
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. [Fre18b]

```
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:

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

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

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

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]

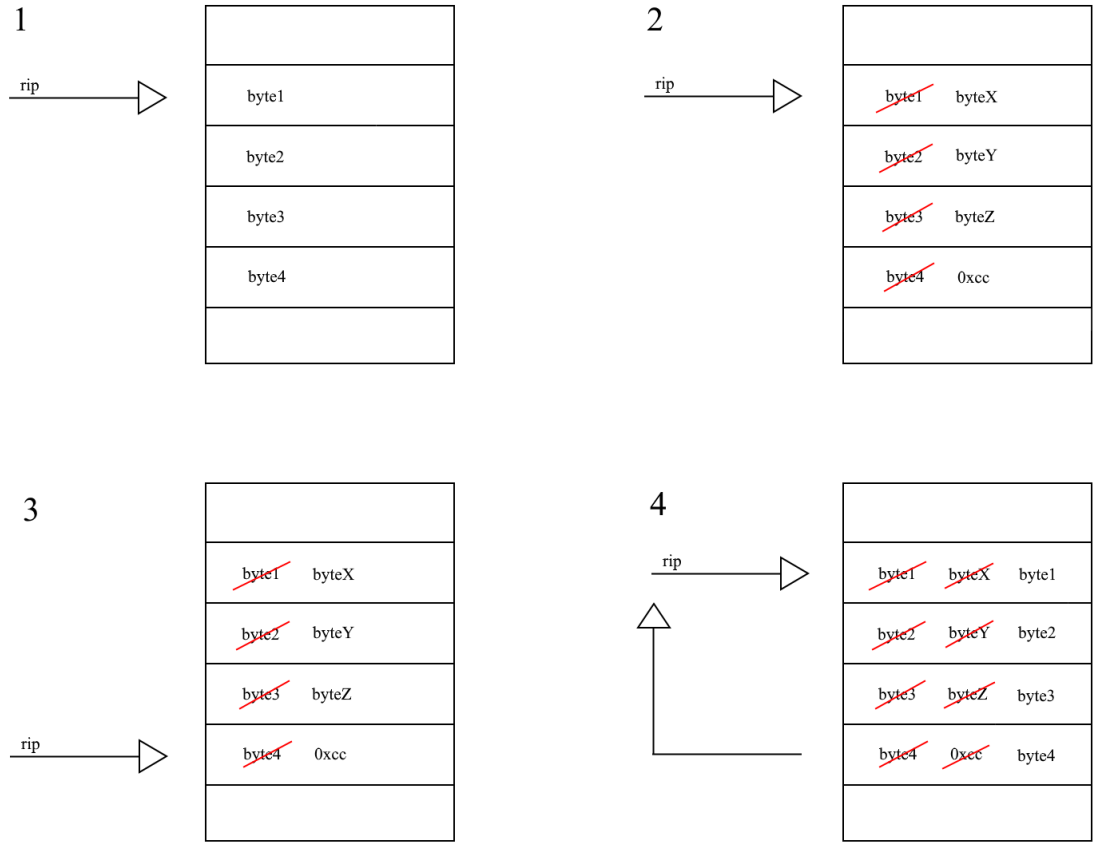

Abbildung 3: *Stackansicht für Codeinjektion*

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.

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 ein-

gesetzt, 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:

```
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`. [Fre18a] 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. [Fre18b]

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

`PTRACE_SYSCALL` springt vor den nächsten Systemaufruf oder verlässt diesen, wenn er bereits im Systemaufruf ist. [Fre18b]

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

Mit `PTRACE_SYSCALL` lassen sich zum Beispiel die Registerwerte vor und nach einem Systemaufruf beobachten oder sogar manipulieren. 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 sys_ptrace 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 `syscall`.

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.[Fre18c] 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.

2.3 Proof of concept - Debugger

Kaywan Katibeh

Um die grundsätzliche Vorgehensweise des Debuggers zu demonstrieren wurde eine Beispielanwendung implementiert [HKH19]. Hierbei ist der Debugger in zwei Programme, dem Server und dem Client, unterteilt. Die Aufgabe des Servers besteht darin, die Befehle des Clients, das kann eine GUI, CLI oder aber auch eine hardcoded Prozedur sein, zu bedienen. Der Client kontrolliert den Server und bestimmt damit die Funktionalität für den Benutzer.

Debugger-Server

Lennart Hein

Als Pseudocode lässt sich die Funktionalität des Servers folgend darstellen:

```
network_init();          // Einrichten der IPC
tracee_init();           // Start des Tracee und PTRACE_TRACEME
loop:
    receive_command();   // Erhalten des Befehls über IPC
    execute_command();   // Durchführen des Befehls unter Zuhilfenahme von sys_ptrace
goto loop;
destroy();               // Beenden des Tracee und der IPC
```

Nach der Initialisierung kann der Server durch den Client kontrolliert werden. Der Client sendet über Berkeley Sockets definierte Bytesequenzen, die einen bestimmten Befehl darstellen, etwa `uint64_t peek_reg(ENUM reg)`. Nachdem der Server die Befehle des Clients empfängt, startet der Server die entsprechende Routine, weitere Kommunikation über IPC 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 IPC. Der Debugger-Server hat im Proof of Concept die Funktionen `void next_syscall(uint32_t syscall)` und `uint64_t peek_reg(ENUM reg)`.

Debugger-Client

Kaywan Katibeh

Der Client kann nun den Server anweisen, diese zwei Funktionen auszuführen. Damit lässt sich bereits ein Debug-Programm schreiben, welches alle Systemaufrufe des Tracee verfolgt.

```
network_init();                        // Einrichten der IPC
loop:
    rax      = req_peek_reg(RAX);      // Speichert Rückgabewert des Systemaufrufs
    orig_rax = req_peek_reg(ORIG_RAX); // Speichert Nummer des Systemaufrufs
    print(rax, orig_rax);              // Ausgabe
    req_next_syscall();                // Warten auf den nächsten Systemaufruf
goto loop
destroy();                             // Beenden der IPC
```

2.4 CreateRemoteThread

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

```
HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

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 bei Erfolg der Handle zu dem neuen Thread. Bei einem Fehler wird *NULL* zurückgegeben. .... (Remarks)

3 Stand und Ausblick

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

3.1 Stand des Debuggers

Maurice Happe

Ein Prototyp ist bereits in 2.3 beschrieben, der Systemaufrufe des Tracee verfolgen kann. Ebenso existiert ein Prototyp zu einer graphical user interface (GUI), dennoch gibt es noch keine Schnittstelle zwischen Debugger und GUI. Für den Debugger sind, neben den in 2.2 beschriebenen, folgende Funktionalitäten geplant, für die noch kein detaillierter Ansatz zur Implementation existiert:

3.1.1 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.

3.1.2 Breakpointverwaltung

Lennart Hein

Der Benutzer soll Breakpoints nicht nur setzen können, sondern diese auch einsehen und daraufhin löschen können. Die Breakpoints werden in einer Datenstruktur gespeichert und bekommen jeweils die Adresse zugewiesen, an welcher sie den Programmfluss unterbrechen.

3.1.3 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 Nameen zugewiesen werden. Dies ermöglicht eine direkte Ausgabe zu einem angefragten Variabel-Namen.

3.1.4 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.

3.1.5 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 }
```

3.2 CreateRemoteThread - Jugaad

Lennart Hein

Jugaad ist ein Programm, welches das Injizieren von Threads, die nutzerdefinierte Funktionen ausführen, in beliebige Prozesse ermöglicht. Jugaad ist eine Implementation für 32-Bit Linux nach dem Vorbild der Windows-Funktion „CreateRemoteThread“. Der Vorteil von Jugaad gegenüber dem älteren Tool „injectSo“, welches eine Bibliothek in den Prozess lädt, liegt insbesondere darin, dass durch Jugaad injizierter Shellcode nicht in Maps-Files sichtbar ist. Mit injectSo in den Prozess mit der ID PID injizierter Shellcode ist in proc/PID/maps aufgelistet. [Jak13]

Jugaad verwendet den Systemaufruf sys_ptrace, um den Thread zu injizieren. Nachdem Speicher für den Thread reserviert wurde, wird sys_clone verwendet, um den Thread zu kreieren und den Shellcode auszuführen. [Jak13] Darauf aufbauend ist geplant, Jugaad nach 64-Bit Linux zu portieren.

3.3 Stand der GUI und Planung

Maurice Happe

Zu einem Debugger gehört auch eine Oberfläche, mit welcher der Benutzer Eingabewerte übergeben und Ausgabewerte einlesen kann. Der GDB in Linux bietet zum Beispiel eine Kommandozeile. Da der Fokus des Debuggers auf einfacher Bedienbarkeit liegt, erscheint die Verwendung einer GUI (graphical user interface) sinnvoll. Es existiert bereits ein Prototyp zu einer HTML-Anwendung, die mit NodeJS kompiliert ist. Vorteile sind einfache Bedienung der Benutzeroberfläche, Kompatibilität mit jedem Browser und Modifizierbarkeit für bessere Anschaulichkeit. Dennoch ist geplant, die GUI nur lokal zu benutzen, da eine Netzkommunikation Sicherheitsmaßnahmen benötigt, die den Rahmen dieser Ausarbeitung sprengen würden. Daher fiel die Entscheidung auf interprocess communication zwischen Debugger und GUI, welche noch in Planung ist.

Literaturverzeichnis

[Ben11] Bendersky, Eli: Position Independent Code (PIC) in shared libraries. 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. 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. 2006. – Online erhältlich unter http://doursat.free.fr/docs/CS446_S06/CS446_S06_3_Memory2.pdf; abgerufen im Februar 2019.

[Fre18a] 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.

[Fre18b] 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.

[Fre18c] 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.

[Fre18d] 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.

[HKH19] Hein, Lennart ; Katibeh, Kaywan ; Happe, Maurice: PG Prozessinjektion. https://github.com/lennihein/PG/tree/v0.3-proof-of-concept, 2019

[Jak13] Jakhar, Aseem: Jugaad Linux Thread Injection Kit. Vorgestellt an: nullcon, 2013. – Online erhältlich unter https://www.defcon.org/images/defcon-19/dc-19-presentations/Jakhar/DEFCON-19-Jakhar-Jugaad-Linux-Thread-Injection.pdf; abgerufen im Januar 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

[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)

Abbildungsverzeichnis

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, 2. März 2019

Lennart Hein, Maurice Happe, Kaywan Katibeh