UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Praktikum 8

Buffer Overflows

Zu bearbeiten bis zum 27. Juni 2023

Cybersecurity im Sommersemester 2023

Jan Wichelmann, Anja Köhl

Einführung

Ziel dieses Praktikums ist das Ausnutzen verschiedener Buffer-Overflow-Schwachstellen. Hierzu steht Ihnen eine Reihe von Zielprogrammen zur Verfügung, die mithilfe bestimmter Eingaben angegriffen werden können.

Material

Sie finden die Zielprogramme auf einem für dieses Praktikum aufgesetzten Linuxsystem:

```
ssh -p 10003 <user>@teaching.its.uni-luebeck.de
```

Die Zugangsdaten finden sich wie immer auf dem Praktikumsserver.

Das gegebene System wird auch für die Auswertung der erstellten Exploits genutzt. Es steht Ihnen frei, die Zielprogramme herunterzuladen und dann lokal zu arbeiten – stellen Sie jedoch vorm Abgeben sicher, dass Ihr Exploit auch auf unserem Linuxsystem funktioniert!

Die Materialien finden sich unter `targets/` in Ihrem Homeverzeichnis. Dort sind zusätzliche einige Quelltexte (als Referenz) enthalten, und das Programm `hex2raw`. Letzteres wandelt eine beliebige Folge von Hexadezimalzeichen in die entsprechende Byte-Repräsentation um. Zum Beispiel wird

```
44 69 65 73
20 69 73 74
20 65 69 6E
20 54 65 78
74 21
```

durch `hex2raw` in (ASCII)

```
Dies ist ein Text!
```

umgewandelt. Wie Sie sehen, werden Leerzeichen und Zeilenumbrüche ignoriert. Auf diese Weise können Sie Ihren Exploit in lesbarer Hexadezimalform konstruieren, und diesen anschließend in Binärform konvertieren und dem Zielprogramm übergeben:

```
cat exploit_hex.txt | ./hex2raw | ./run_prog.sh ./progX
```

Zielprogramme

Sämtliche Zielprogramme lesen von der Standardeingabe. Das Einlesen geschieht über die Funktion `read_input`. Diese ist wie folgt definiert:

```
int read_input(void)
{
    // Buffer to be exploited
    char inputBuffer[32];

    // Read user input from stdin
    fprintf(stderr, "Please enter your input: ");
    fflush(stderr);
    gets2(inputBuffer);

    // Done
    return 1;
}
```

Die Funktion `gets2` verhält sich dabei wie die Standard-Bibliotheks-Funktion `gets` – sie liest einen String ein und speichert ihn zusammen mit einem terminierenden 0-Byte an der angegebenen Stelle. In diesem Code ist das Speicherziel das auf dem Stack allozierte Array `inputBuffer`, welches 32 Bytes fassen kann (inklusive des terminierenden 0-Bytes).

Die Funktion `gets2` hat keine Möglichkeit zu entscheiden, ob das Ziel-Array ausreichend Speicherplatz für den einzulesenden String bietet. Sie kopiert einfach Byte-Sequenzen und schreibt dabei möglicherweise auch über die Arraygrenzen hinaus.

Ist der Eingabestring, der von `read_input` eingelesen wird, kurz genug, wird das Programm erfolgreich terminieren:

```
$ ./prog1
Please enter your input: shortinput
Everything went well :)
```

Jedoch tritt typischerweise ein Fehler auf, falls Sie einen zu langen String eingeben:

```
$ ./prog1
Please enter your input: someveryverylonginputthatisdesignedtobreakthisprogram
Segmentation fault (core dumped)
```

Ihre Aufgabe ist es, intelligentere Eingaben zu finden, sodass die Zielprogramme andere Dinge tun als bloß abzustürzen. Solche Eingaben nennen wir *Exploits*.

Wichtige Anmerkungen:

- Ihre Exploits dürfen das Byte `0x0a` (außer ggf. am Ende) nicht enthalten, da es der ASCII-Code für einen Zeilenumbruch ist. Wenn `gets2` dieses Byte liest, wird die Funktion annehmen, dass Ihre Eingabe dort endet.
- x86 nutzt die *Little Endian*-Bytereihenfolge. Das bedeutet, dass bei einer größeren Zahl zuerst das niedrigstwertige Byte angegeben wird. Beispiel: Die 32-Bit-Hexadezimalzahl

  ```
  0x12345678
  ```

  wird im Speicher als

  ```
  78 56 34 12
  ```

  dargestellt. Berücksichtigen Sie dies, wenn Sie Zahlen oder Adressen in Ihren Exploit einbauen.

Aufgabe 1 ASLR abschalten

Machen Sie sich mit der Funktion von `run_prog.sh` vertraut. Was bewirkt das darin enthaltene Kommando, und warum ist es notwendig?

Aufgabe 2 Einstieg

◀ 60 / 0

Erstellen Sie eine Eingabe, die das Programm `prog1` dazu bringt, die folgende Funktion aufzurufen:

```
void target_function(void)
{
    // Successful exploitation
    printf("Exploited!\n");

    // Exit program (else it will crash)
    exit(0);
}
```

Die Funktion erwartet keine Parameter; damit genügt es, auf dem Stack die Rücksprungadresse durch die Adresse von `target_function` zu ersetzen.

Hinweis: Disassemblieren Sie das Programm. In der Disassembly finden Sie alle Informationen, die für diesen Exploit benötigt werden.

Aufgabe 3 Ausführbarer Stack 1

◀ 85 / 0

Erstellen Sie eine Eingabe, die das Programm `prog2` dazu bringt, die folgende Funktion aufzurufen:

```
void target_function(uint64_t param)
{
    // Output message and parameter
    printf("Exploited with parameter '%" PRIx64 "'!\n", param);

    // Exit program (else it will crash)
    exit(0);
}
```

Wie Sie sehen, erwartet die Funktion eine vorzeichenlose 64-Bit-Zahl als Parameter; nehmen Sie hierfür (Dezimaldarstellung):

```
1507192586462
```

Um die Zahl vor dem Aufruf in das korrekte Register zu schreiben, müssen Sie mit Ihrer Eingabe etwas eigenen Code injizieren. Praktischerweise ist der Stack als ausführbar markiert, sodass Sie in Ihrer Eingabe die entsprechenden Maschinenbefehle ablegen und anschließend durch Modifikation der Rücksprungadresse zur entsprechenden Position auf dem Stack springen können. Ihr Code wird dann ausgeführt. Sie finden am Ende dieser Praktikumsbeschreibung einen kleinen Anhang, der die Generierung von Maschinencode aus Assemblerbefehlen erklärt.

Hinweis: Nutzen Sie zum Springen in die Funktion `target_function` nicht die `jmp`- oder `call`-Instruktionen. Die Codierung der Zieladressen ist bei diesen Instruktionen schwierig und fehleranfällig. Nutzen Sie stattdessen die `ret`-Instruktion, auch wenn Sie nicht von einem `call` zurückkehren – Sie haben ja die Kontrolle über den Stack.

Aufgabe 4 Return-oriented Programming 1

◀ 105 / 0

Erstellen Sie eine Eingabe, die das Programm `prog4` dazu bringt, die folgende Funktion aufzurufen:

```
void target_function(uint64_t param)
{
    // Output message and parameter
    printf("Exploited with parameter '%" PRIx64 "'!\n", param);

    // Exit program (else it will crash)
    exit(0);
}
```

Die Zielfunktion ist identisch zu der in `prog2`, als Parameter soll wieder die Zahl

```
1507192586462
```

übergeben werden; allerdings ist der Stack hier nicht ausführbar, sodass Sie nicht unmittelbar eigenen Code injizieren können.

Glücklicherweise hat `prog4` eine Menge anderer Funktionen `gadget_source<X>`, die wiederum eine Menge Maschinencode mit sich bringen, der sich mittels *Return-oriented Programming* (ROP) zweckentfremden lässt. Suchen Sie hierfür nach Byte-Sequenzen, die eine oder mehrere Instruktionen gefolgt von einer `ret`-Instruktion enthalten (sogenannte *Gadgets*).

Sie sind hierbei nicht mal auf unmittelbar vom Compiler assemblierte Instruktionen beschränkt: Da x86 teils sehr lange Maschinenbefehle benutzt, können Sie Ihren Code genauso gut mitten in einer Instruktion beginnen lassen. Beispiel: Die Funktion

```
void gadget_source1(uint64_t *ptr)
{
    *ptr = 3616098542;
}
```

wird vom Compiler zu

```
0000000000400f15 <gadget_source1>:
    400f15: c7 07 ee 48 89 d7   mov [rdi], 0xd78948ee
    400f1b: c3                  ret
```

übersetzt. Während dies auf den ersten Blick nicht nach einem wirklich nutzbaren Codefragment aussieht, lässt es sich dennoch in ein Gadget umfunktionieren: Die Sequenz `48 89 d7` kodiert die Instruktion `mov rdi, rdx`. Auf diese Sequenz folgt das Byte `0xc3`, welches die `ret`-Instruktion kodiert. Die Funktion `gadget_source1` startet an Adresse `0x400f15`, und die Sequenz beginnt drei Byte später. Somit enthält der Code ein Gadget, das an Adresse `0x400f18` beginnt und das einen 64-Bit Wert von Register `rdx` in das Register `rdi` kopiert.

Hinweise:

- Nutzen Sie für Gadgets ausschließlich die entsprechend benannten Funktionen (`gadget_source<X>`).
- Sie können mit GDB beliebige Offsets in den Gadgets disassemblieren:

  ```
  > x/2i gadget_source1+0x3
  0x555555555374 <gadget_source1+3>:   mov    rdi,rdx
  0x555555555377 <gadget_source1+6>:   ret
  ```

  Dieser Befehl weist GDB an, an dem Offset `gadget_source1+0x3` zwei Instruktionen zu disassemblieren.
- Alternativ finden Sie im Anhang einige Tabellen mit Instruktionen und den zugehörigen Maschinenbefehlen, sodass Sie diese unmittelbar ablesen können und nicht erst disassemblieren müssen.
- Es kann helfen, zuerst die Instruktionen aller gefundenen potenziellen Gadgets aufzuschreiben, und daraus dann eine ROP-Kette zu konstruieren.
- Sie können Gadgets auch mehrfach benutzen.

Aufgabe 5 Ausführbarer Stack 2 (freiwillig)

Erstellen Sie eine Eingabe, die das Programm `prog3` dazu bringt, die folgende Funktion aufzurufen: ◁ 125 / 0

```
void target_function(const char *str)
{
    // Output message and parameter
    printf("Exploited with parameter '%s'!\n", str);

    // Exit program (else it will crash)
    exit(0);
}
```

Die Funktion erwartet einen String als Parameter; nehmen Sie hier die Zeichenfolge

```
r00t
```

(terminierendes 0-Byte nicht vergessen!)

Der Angriff nutzt wie schon bei `prog2` den ausführbaren Stack; allerdings wird hier statt einer festen Zahl die Adresse des String-Parameters übergeben, der sich ebenfalls auf dem Stack befinden soll.

Aufgabe 6 Return-oriented Programming 2 (freiwillig)

Erstellen Sie eine Eingabe, die das Programm `prog5` dazu bringt, die folgende Funktion aufzurufen: ◁ 165 / 0

```
void target_function(uint64_t param)
{
    // Output message and parameter
    printf("Exploited with parameter '%" PRIx64 "'!\n", param);

    // Exit program (else it will crash)
    exit(0);
}
```

Die Zielfunktion ist identisch zu der in `prog4`, als Parameter soll also wieder die Zahl

```
1507192586462
```

übergeben werden. Der Stack ist wieder nicht ausführbar.

Nutzen Sie auch hier ausschließlich die Funktionen `gadget_source<X>` als Quelle von ROP-Gadgets.

Anhang A: Byte-Code-Generierung

Nutzt man `as` als Assembler und `objdump` als Disassembler, so ist es einfach, Instruktionssequenzen in Maschinencode zu übersetzen. Gegeben sei beispielsweise die Datei `example.s` mit folgendem Inhalt:

```
# Beispiel fuer von Hand generierten Assembler-Code
.intel_syntax noprefix  # Diese Zeile ist wichtig!
push    0xabcdef
add     rax, 0x11
mov     edx, eax
```

Der Code darf eine Mischung aus Instruktionen und Daten enthalten. Alles, was auf ein # folgt, wird als Kommentar gewertet. Die Datei sollte mit einer Leerzeile enden, um Warnungen zu verhindern.

Sie können die Datei nun assemblieren und disassemblieren:

```
$ as example.s -o example.o
$ objdump -d -M intel example.o > example.d
```

Die erstelle Datei `example.d` enthält Folgendes:

```
example.o:     file format pe-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
0:   68 ef cd ab 00          push   0xabcdef
5:   48 83 c0 11             add    rax,0x11
9:   89 c2                   mov    edx,eax
```

Die letzten drei Zeilen enthalten den Maschinencode, der aus den Assembler-Instruktionen generiert wurde. Wir können also ablesen, dass die Instruktion `push 0xabcdef` den Maschinencode `68 ef cd ab 00` besitzt.

Aus dieser Datei erhalten wir damit die Byte-Sequenz für unseren Assembler-Code:

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

Dieser String könnte nun an `hex2raw` übergeben werden, um eine binäre Eingabe für das Zielprogramm zu erzeugen.

Anhang B: Instruktionen und Maschinenbefehle

`mov D, S` (64-Bit):

Quelle S	Ziel D							
	`rax`	`rcx`	`rdx`	`rbx`	`rsp`	`rbp`	`rsi`	`rdi`
`rax`	`48 89 c0`	`48 89 c1`	`48 89 c2`	`48 89 c3`	`48 89 c4`	`48 89 c5`	`48 89 c6`	`48 89 c7`
`rcx`	`48 89 c8`	`48 89 c9`	`48 89 ca`	`48 89 cb`	`48 89 cc`	`48 89 cd`	`48 89 ce`	`48 89 cf`
`rdx`	`48 89 d0`	`48 89 d1`	`48 89 d2`	`48 89 d3`	`48 89 d4`	`48 89 d5`	`48 89 d6`	`48 89 d7`
`rbx`	`48 89 d8`	`48 89 d9`	`48 89 da`	`48 89 db`	`48 89 dc`	`48 89 dd`	`48 89 de`	`48 89 df`
`rsp`	`48 89 e0`	`48 89 e1`	`48 89 e2`	`48 89 e3`	`48 89 e4`	`48 89 e5`	`48 89 e6`	`48 89 e7`
`rbp`	`48 89 e8`	`48 89 e9`	`48 89 ea`	`48 89 eb`	`48 89 ec`	`48 89 ed`	`48 89 ee`	`48 89 ef`
`rsi`	`48 89 f0`	`48 89 f1`	`48 89 f2`	`48 89 f3`	`48 89 f4`	`48 89 f5`	`48 89 f6`	`48 89 f7`
`rdi`	`48 89 f8`	`48 89 f9`	`48 89 fa`	`48 89 fb`	`48 89 fc`	`48 89 fd`	`48 89 fe`	`48 89 ff`

Andere Instruktionen:

Instruktion	Register R							
	`rax`	`rcx`	`rdx`	`rbx`	`rsp`	`rbp`	`rsi`	`rdi`
`pop` R	`58`	`59`	`5a`	`5b`	`5c`	`5d`	`5e`	`5f`