Studiengang Informatik
apsi 2020

Fachhochschule Nordwestschweiz
Arno Wagner, Michael Schläpfer, Rolf Wagner

Übungsblatt 2

21. September 2020

Aufgabe 2-1: Buffer Overflow

In der Vorlesung haben Sie erfahren, was ein Buffer Overflow ist und haben dazu ein Beispiel gesehen. In dieser Aufgabe geht es nun darum, das Beispiel nachzustellen und anschliessend einen "echten" Exploit zu verwenden.

i) Simple Buffer Overflow

Kompilieren Sie das Beispiel aus der Vorlesung und achten Sie dabei darauf, dass Stack-Protection deaktiviert ist (-fno-stack-protector).

```c
#include <string.h>
#include <stdio.h>

void test (int argc, char *argv[]) {
  char s2[4] = "yes";     // set s2 to "yes"
  char s1[4] = "abc";     // set s1 to "abc"
  strcpy(s1, argv[1]);    // copy argv[1] into s1
  puts(s2);               // print s2
}

int main (int argc, char *argv[]) {
  test(argc, argv);
  puts("end\n");
}
```

Verwenden Sie nun den GNU Debugger (gdb). Versuchen Sie mittels Ihrer Eingaben einen *Segfault* zu verursachen und dabei folgende Fragen zu beantworten:

(a) Was bedeutet die Fehlermeldung *Segfault*? Weshalb erhalten Sie diese?

(b) Gehen Sie Schritt für Schritt durch das Programm und notieren Sie sich die Werte welche den Variablen *s1* und *s2*, sowie der *Return Address* zugewiesen sind.

(c) Bei welchem Schritt in obigem Programm tritt der *Segfault* auf?

(d) Welche Werte stehen in der *Return Address* des Stackframes zu Beginn und am Ende?

ii) Buffer Overflow Exploit

Nachdem Sie sich nun intensiver mit Buffer Overflows und gdb auseinander gesetzt haben, wollen wir nun einen einfachen Exploit verwenden, welcher den Programmablauf verändern soll.

Wir basieren hierfür auf einem Tutorial von Dhaval Kapil (2015) und ergänzen hier noch einige Hinweise. Versuchen Sie das Tutorial nachzustellen und schlussendlich die *geheime Funktion* aufzurufen, indem Sie dem verwundbaren Programm einen entsprechenden Payload mitgeben.

`https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/`

Das Tutorial verwendet den nachfolgenden Code. Versuchen Sie diesen zuerst zu verstehen, bevor Sie mit dem Tutorial starten.

```c
#include <stdio.h>

void secretFunction() {
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main() {
    echo();

    return 0;
}
```

Zur Vereinfachung können Sie durch das Installieren der GCC Multilib (apt-get install gcc-multilib) und dem Setzen des des Flags -m32 ein 32 bit Binary erstellen. Verwenden Sie für die Aufgabe auch das -static Flag und überlegen Sie sich, was dieses macht.

`gcc -fno-stack-protector -m32 -static -ggdb -o vuln vuln.c`

Beschreiben Sie stichwortartig ihr Vorgehen und geben Sie die Berechnungen für den Payload in Ihrem Programm an. Welches ist Ihr erfolgreicher Aufruf?

Aufgabe 2-2: Algorithmic Complexity Attacks

In der Vorlesung wurde darauf eingegangen, dass ein Angreifer die Worst-Case Eigenschaften verwendeter Algorithmen ausnutzen und damit ein verwundbares System verlangsamen oder gar vollständig auslasten kann. Im folgenden beschäftigen wir uns eingehender mit solchen Verwundbarkeiten.

i) Hashtables

Hashtables sind omnipräsent und ermöglichen einen effizienten Zugriff auf Daten. In der Vorlesung haben Sie bereits gehört, dass diese aber auch anfällig sind auf *Algorithmic Complexity Attacks*. Der folgende Artikel zeigt auf, welches Ausmass diese Problematik tatsächlich hatte und weiterhin hat:

`https://www.freecodecamp.org/news/hash-table-attack-8e4371fc5261/`

Beantworten Sie folgende Fragen:

(a) Was bedeutet eine Komplexität von $O(1)$. Unter welchen Umständen erreicht eine Hashtable diese Zeitkomplexität nicht? Was ist die Worst-Case Komplexität von Hashtables?

(b) Welches sind die zentral wichtigen Eigenschaften einer Hashfunktion, welche für Hashtables verwendet wird?

(c) Welche Verfahren werden im Umgang mit Kollisionen üblicherweise verwendet?

(d) Weshalb werden nicht einfach kollisionsresistente kryptographische Hashfunktionen verwendet?

ii) ReDoS Attacks

Regular Expressions werden vielerorts verwendet, unter anderem sehr oft, um Eingaben zu prüfen. Sie werden künftig mit Sicherheit ebenfalls mit *Regex* in Berührung kommen, falls dies nicht bereits geschehen ist. Hier können Sie vieles falsch oder eben richtig machen.

Lesen Sie sich anhand des nachfolgenden Links vertiefter in die Thematik von *ReDoS Attacks* ein beantworten Sie anschliessend die Fragen:

`https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS`

(a) Welches sind typische Anwendungsbereiche von Regex.

(b) Was macht ein schlechtes Pattern aus? Was ein gutes?

(c) Services, welche auf Cloud-Computing Infrastruktur laufen, sind hochskalierbar. Sehen Sie hier Angriffsszenarien, welche über blosses DoS hinaus gehen? Wer könnte ein Interesse an solchen Angriffen haben?

iii) Email Address Validation

Ein sehr gängiger Anwendungsfall von *Regex* ist die Prüfung von Eingaben. Dazu gehört beispielsweise auch die Überprüfung von Email-Adressen.

In diesem Zusammenhang erklärt Ihnen ein Kollege stolz, dass er für seinen Arbeitgeber eine Email-Adressen-Überprüfung umgesetzt hat. Dabei hat er folgende *Regex* entwickelt:

```
1 ^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-9]+[.]{1}(([a-z
      ]{2,3})|([a-z]{2,3}[.]{1}[a-z]{2,3}))$
```

Wenn Sie den Ausdruck zur Analyse kopieren müssen Sie darauf achten, dass die Zeichen korrekt übernommen werden, insbesondere "-" und "*".

Beantworten Sie folgende Fragen:

(a) Erfüllt der Ausdruck den gewünschten Zweck? Wenn nein, wieso nicht?

(b) Wie sieht die Worst-Case Komplexität aus? Verwenden Sie einen Regex Tester um zu experimentieren.

(c) Machen Sie ein Beispiel eines "schlechten" Inputs um Ihrem Kollegen das Problem anschaulich zu demonstrieren.