UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Übungsblatt 10

Software Security

Abgabe bis 27. Juni 2023 um 12:00 Uhr im Moodle

Cybersecurity im Sommersemester 2023

Thomas Eisenbarth, Jan Wichelmann, Anja Köhl

Einführung

Die Aufräumarbeiten nach dem Malware-Angriff bei ITS sind inzwischen abgeschlossen. Nach wie vor ist jedoch unklar, wie es dazu überhaupt kommen konnte; grundsätzlich haben die Angreifer offenbar den extern erreichbaren Techniker-Login des neuen Kontextwechsel-Abrechungssystems umgangen, und dadurch Adminzugriff auf mehrere Server erlangt. Allerdings konnte bisher noch niemand irgendeinen offensichtlichen Denkfehler in besagtem Code entdecken. Dies bedeutet, dass der Fehler entweder in einer anderen Komponente liegt (unwahrscheinlich), oder dass die Programmierer bei ITS etwas übersehen haben.

Aufgabe 1 Bugs finden (6 Punkte)

Am Ende des Übungsblattes finden Sie den sicherheitsrelevanten Login-Code der Implementierung. Da diese noch im Testlauf war, hatten nur fünf Benutzer Zugang zu diesem System; dennoch ist es EVE (oder wem auch immer) trotzdem gelungen, Zugriff zu bekommen. Dies kann nicht daran gelegen haben, dass ein Benutzername oder eine PIN erraten wurde, da als Schutz vor Brute-Force-Angriffen bei jedem fehlgeschlagenen Login die entsprechende IP-Adresse eine Stunde lang blockiert wird, und das System insgesamt nur wenige Tage lang online war.

Untersuchen Sie den gegebenen Code nach Stellen, wo dieser sich in vermutlich unbeabsichtigter Weise verhält, und bearbeiten Sie für jede die folgenden Punkte:

1. An welcher Position (Zeilennummer) im Programm befindet sich die Schwachstelle?
2. Erklären Sie kurz, was diese Schwachstelle verursacht.
3. Beschreiben Sie kurz, wie und wofür die Schwachstelle ausgenutzt werden kann.
4. Nennen Sie eine Möglichkeit, die Schwachstelle zu beheben.

Hinweise:

- Es gibt mindestens *drei* Schwachstellen, die sich in ihrer Art und Ausnutzung unterscheiden.
- Bei einem Netzwerkprogramm würden natürlich nicht `printf` und `gets` verwendet, da diese nur lokal funktionieren; die entsprechenden verschlüsselten Sende-/Empfang-Befehle wurden hier nur der Lesbarkeit halber ersetzt. Netzwerkfunktionen sind sehr anfällig gegenüber Out-of-Bounds-Zugriffen, da sie die nach außen sichtbare Oberfläche eines Systems bilden, und damit unmittelbar Angriffen ausgesetzt sind.

Aufgabe 2 Exploitation Methoden (7 Punkte) *[optional bei bestandenem Praktikum]*

Diese Aufgabe muss nicht bearbeitet werden, wenn Sie stattdessen das "Buffer Overflows"-Praktikum bestehen. Bitte geben Sie Ihren Gruppennamen aus dem Praktikum an, um die Korrektur zu erleichtern.

Im vorliegenden Login-Code sticht die Verwendung der unsicheren Funktion `gets` heraus, die dafür bekannt ist, Buffer Overflows zu verursachen. Wie kann ein Angreifer diese jedoch ausnutzen?

Erklären Sie die verschiedenen Angriffstechniken *Code Injection*, *Return-oriented Programming* und *Jump-oriented Programming*, und vergleichen Sie diese hinsichtlich ihrer Vor- und Nachteile für den Angreifer. Welche der Angriffsmethoden sind auf Stack Buffer Overflows beschränkt, welche funktionieren auch auf dem Heap?

Aufgabe 3 Allgemeine Gegenmaßnahmen (7 Punkte)

ITS bittet Sie nun um Beratung zur zukünftigen Vermeidung von Software-Schwachstellen.

1. Erklären Sie den Unterschied zwischen Gegenmaßnahmen zur Implementierungszeit (*Implementation Countermeasures*) und *Mitigations*.
2. Nennen Sie drei Gegenmaßnahmen gegen Buffer-Overflows und diskutieren Sie Vor- und Nachteile.
3. Praktikant Peter hat sich im Rahmen der aktuellen Vorkommnisse selbst etwas in die Materie eingelesen und schlägt eine neue Technik vor, um in der ITS-Software Buffer Overflows auf dem Stack endgültig zu verhindern. Hierfür möchte er die Linux-64-Bit-Aufrufkonvention so verändern, dass die Rücksprungadresse einer Funktion nicht nur auf dem Stack, sondern zusätzlich auch im Register `rcx` abgelegt wird. Vor dem Rücksprung wird das Register dann mit dem entsprechenden Stack-Eintrag verglichen. Dies sieht beispielhaft so aus:

```
func1:
    ...
    mov rdi, [param1]            ; Load parameter 1 into rdi
    mov rsi, [param2]            ; Load parameter 2 into rsi
    lea rcx, [func1.after_call]  ; Get address after call
    call add_two_numbers
func1.after_call:
    ...

add_two_numbers:
    add rdi, rsi          ; Calculate sum...
    mov rax, rdi          ; ...and store it in rax as return value
    cmp rcx, [rsp]        ; Check for valid return address on stack
    jne exit_with_error   ; Return address is invalid, exit program safely
    ret                   ; Return address is valid, so return
```

   Ist dies eine geeignete Gegenmaßnahme gegen Buffer-Overflows? Was sind Vorteile, was sind Nachteile gegenüber anderen Techniken?

Aufgabe 4 Automatisierte Gegenmaßnahmen (6 Punkte)

Zusätzlich zu Ihren Erklärungen zu allgemeinen Gegenmaßnahmen wünscht sich ITS eine Prüfung nach Sicherheitsproblemen zur Laufzeit. Solche Funktionen sind glücklicherweise bereits leicht nutzbar: Der Compiler `gcc` beispielsweise fügt standardmäßig bei Funktionen, die mit Strings arbeiten, automatische Stack-Überprüfungen ein. Zudem erlaubt er die Nutzung Hardware-basierter Technologien zur Sicherstellung der Kontrollflussintegrität. Aber sind diese Maßnahmen auch ausreichend?

Im Moodle finden Sie den Quelltext und die Disassembly des Programms `protectiontest.c`, das mit den folgenden Befehlen kompiliert und disassembliert wurde:

```
gcc protectiontest.c -o protectiontest -Os -fomit-frame-pointer -fno-inline
objdump protectiontest -d -M intel > protectiontest.asm
```

(die drei Compilerflags sorgen dafür, dass der Compileroutput halbwegs brauchbar im Sinne der Aufgabenstellung wird; genutzt wurde GCC 9.3.0).

1. Informieren Sie sich im *Intel Software Developers Manual*[1] (Vol. 1, Chapter 18) über die *Control-flow Enforcement Technology* (CET)-Erweiterung. Welche Funktionen bietet diese? Welchen Zweck erfüllt die `endbr64`-Instruktion?

2. Beschreiben Sie den vom Compiler in der Funktion `toUpperCase` eingesetzten Stack-Schutzmechanismus, und erklären Sie ob dieser zum Schutz vor Buffer-Overflows ausreichend ist.

   Hinweis: Manche Instruktionen verwenden Operanden vom Typ `fs:0x**` (`*` steht hier für eine Hexadezimalzahl). Dies sind Zugriffe über sogenannte *Segmentregister*; diese funktionieren im Allgemeinen genauso wie normale direkte Speicherzugriffe, nur dass jeder Thread hier seinen eigenen Speicherbereich hat (Stichwort *Thread Local Storage*).

Verwundbarer Code zu Aufgabe 1

```c
#include <stdio.h>

char *usernames[5] = {
    "wolfgang",
    "mona",
    "chef",
    "lisa",
    "amadeus"
};

int digitToIntLookup[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int pins[5][4] = {
    { 2, 2, 3, 4 },
    { 4, 2, 4, 3 },
    { 9, 9, 7, 9 },
    { 1, 3, 3, 7 },
    { 2, 4, 6, 8 }
};

int securityLevels[5] = { 2, 1, 4, 3, 1 };

char getCharAtIndex(char *str, int index)
{
    return str[index];
}

int digitToInt(char digit)
{
    // Use lookup table to translate ASCII digit to integer
    return digitToIntLookup[digit - '0']; // Get offset in lookup table
}

int login()
{
    // Read user name
    printf("User: ");
    char username[4096]; // Enough memory to avoid buffer overflows even for
        very long user names
    if(gets(username) == 0)
        return 0;

    // Read password
    printf("PIN: ");
    char pinString[4096]; // same here
```

[1] https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

```
45      if(gets(pinString) == 0)
46          return 0;
47
48      // Find user with given name
49      int usernameLength = strlen(username);
50      int userId = -1;
51      for(int u = 0; u < 5; ++u)
52      {
53          // Matching user name?
54          int match = 1;
55          for(int i = 0; i < usernameLength; ++i)
56          {
57              // Does the current input char match the corresponding one in the
                    user name?
58              if(getCharAtIndex(usernames[u], i) != getCharAtIndex(username, i))
59              {
60                  match = 0;
61                  break;
62              }
63          }
64          if(match == 1)
65          {
66              userId = u;
67              break;
68          }
69      }
70
71      // User found?
72      if(userId == -1)
73          return 0;
74
75      // Convert input PIN into integer representation
76      int pin[4];
77      for(int i = 0; i < 4; ++i)
78          pin[i] = digitToInt(pinString[i]);
79
80      // Compare PIN
81      for(int i = 0; i < 4; ++i)
82          if(pin[i] != pins[userId][i])
83              return 0;
84      return 1;
85  }
86
87  int main()
88  {
89      // ... Code ...
90
91      printf("Welcome at ITS context switch counting system!\n");
92      if(login() == 0)
93      {
94          // Error message
95          printf("Access denied.\n");
96
97          // Block IP address for 1 hour
98          // Code...
99
100         // Kill connection
101         return 0;
102     }
```

```
103     printf("Access granted.\n");
104
105     // ... Code ...
106 }
```