Studienarbeit über die Schwachstelle CVE–2019–11005

Stack-based Buffer Overflow in GraphicsMagick

M. Heckel, C. Pöhlmann

Sommersemester 2019

Zusammenfassung

Bei der Schwachstelle CVE-2019-11005 handelt es sich um einen Buffer Overflow auf dem Call Stack. Die Schwachstelle ermöglicht es einem Angreifer, einen definierten Wert (0x00) an eine definierte Speicherstelle zu schreiben, die sich außerhalb eines Puffers befindet.

Im Rahmen dieser Arbeit wird das Hintergrundwissen vermittelt, welches nötig ist um die Schwachstelle und deren Ausnutzung zu verstehen. Im Anschluss wird erklärt, worin die Schwachstelle konkret besteht und wie sie sich ausnutzen lässt. Es wird begründet, warum diese Ausnutzung in der Praxis (zumindest unter den in dieser Arbeit untersuchten Bedingungen) nicht relevant ist. Im Folgenden wird ein Fall konstruiert, der demonstriert, welche Konsequenzen die Ausnutzung dieser Schwachstelle unter anderen Bedingungen hätte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Auflistungsverzeichnis

Abkürzungsverzeichnis

- **CVE**: Common Vulnerabilities and Exposures
- **GDB**: GNU-Debugger
- **GNU**: “GNU’s Not Unix!”
- **GEF**: GDB Enhanced Features (Plugin für GDB)
- **BOF**: Buffer Overflow
- **SVG**: Scalable Vector Graphics
- **API**: Application Programming Interface
- **DoS**: Denial of Service
- **CVSS v3.0**: Common Vulnerability Scoring System (Version 3.0)
- **IPC**: Inter Process Communication
- **LIFO**: Last In First Out
- **CPU**: Central Processing Unit
- **ebp**: Extended Base Pointer (x86)
- **esp**: Extended Stack Pointer (x86)
- **eip**: Extended Instruction Pointer (x86)
- **rbp**: Base Pointer (x86–64)
- **rsp**: Stack Pointer (x86–64)
- **rip**: Instruction Pointer (x86–64)
- **GCC**: GNU Compiler Collection
- **MSB**: Most Significant Bit
- **LSB**: Least Significant Bit
- **qword**: Quad-Word (8 Bytes)

1 Einführung

GraphicsMagick [1] ist eine Bibliothek zum Manipulieren von Bildern. Es können Bilder bezüglich ihrer Größe, Farbe, Schärfe, Drehung, etc. bearbeitet und in den meisten Bildformaten gespeichert werden. 2002 wurde das Projekt GraphicsMagick durch einem Fork von ImageMagick gegründet [2]. GraphicsMagick wirbt auf der Projekthomepage [1] mit einer stabilen API, erhöhter Effizienz und weiteren Vorteilen gegenüber ImageMagick und anderen ähnlichen Tools. Außerdem ist die Entwicklung nicht auf ein geschlossenes Team begrenzt, wodurch jeder die Möglichkeit hat, an GraphicsMagick mitzuwirken [3]

GraphicsMagick wird aktiv entwickelt und von vielen Benutzern weltweit genutzt. Aus den folgenden Statistiken [6] geht hervor, dass es sehr viele Downloads allein von der Sourceforge-Seite gab. In diesen Statistiken sind die Pakete, die diverse Linux-Distributionen anbieten, nicht berücksichtigt.

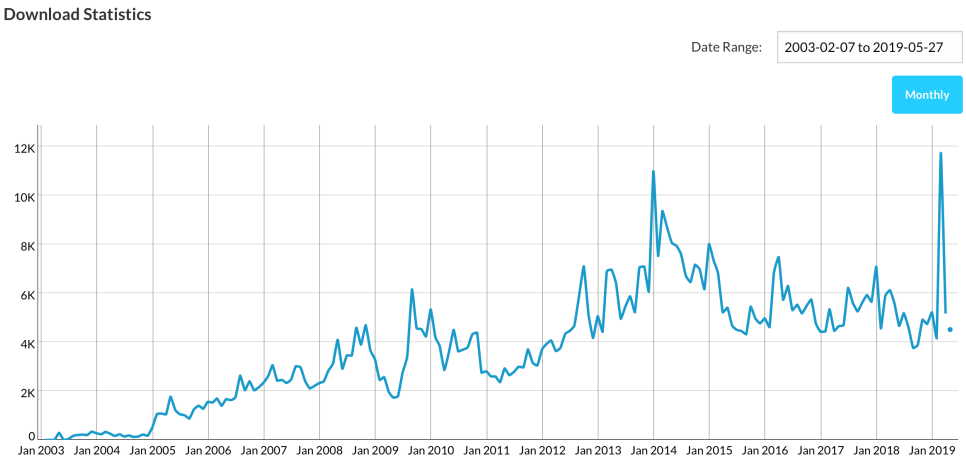


Abbildung 1: Statistik der Downloadanzahl nach Zeitraum [4]

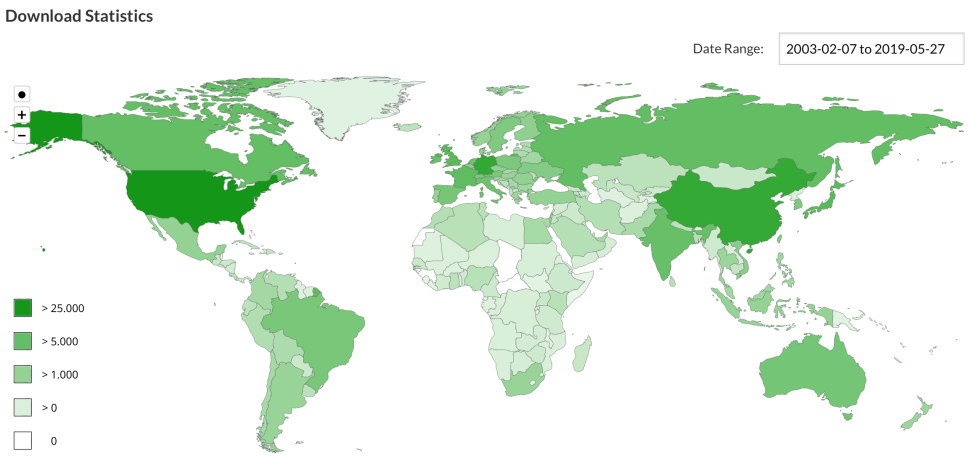


Abbildung 2: Statistik der Downloadanzahl nach Ländern [5]

Zusätzlich zu der Bibliotheksfunktionalität bietet GraphicsMagick das Kommandozeilentool **gm**, mit dem Bilder einfach auf der Kommandozeile manipuliert werden können. Die im Rahmen dieser Arbeit gezeigten Beispiele beziehen sich auf dieses Kommandozeilentool.

Bei der in dieser Arbeit behandelten Schwachstelle handelt es sich um einen Buffer Overflow auf dem Stack, der in Version 1.4 von GraphicsMagic auftritt. In dem zugehörigen CVE [7] wird beschrieben, dass die Schwachstelle zum Denial of Service (DoS) und möglicherweise zu weiteren nicht genauer spezifizierten Auswirkungen führen kann. Ein DoS bei einer Kommandozeilenanwendung wäre nicht besonders kritisch und würde den Score (CVSS v3.0) von 9.8 CRITICAL [8] nicht rechtfertigen. Da es sich bei GraphicsMagick allerdings um eine Bibliothek handelt, die bei verschiedensten Anwendungen eingebunden werden kann, um Bilder zu bearbeiten, betrifft dieses Problem auch alle Programme, die diese Bibliothek verwenden.

Die Auswirkungen können zwar durch technische Maßnahmen (z.B. Starten der Bibliothksfunktionalität in einem eigenen Prozess) verhindert werden (in diesem Fall würde tatsächlich nur die Bibliothek, nicht die gesamte Anwendung abstürzen), allerdings ist ein plötzlicher Absturz der Bibliothek kein vorhersehbares Verhalten, weshalb diese Gegenmaßnahme in der Realität häufig nicht ergriffen wird. Dies ist insbesondere der Fall, da das erstellen eines neuen Prozesses und die Kommunikation über IPC (Inter-Process Communication) im Bezug auf Rechenleistung und Speicherverbrauch deutlich aufwändiger sind als ein direkter Aufruf. Solche Maßnahmen werden im Rahmen dieser Arbeit nicht weiter untersucht.

Der Ursprüngliche Exploit [9] wurde sehr wahrscheinlich von einem Fuzzing-Tool erstellt. Durch den aktivierten AddressSanitizer stürzt das Programm mit einer Fehlermeldung ab, wenn ungültige Speicherzugriffe erkannt werden. Der AddressSanitizer ist in produktiv eingesetzten Version normalerweise nicht aktiviert [10] und dieser Exploit führt in einer Version ohne AddressSanitizer zum kontrollierten Beenden von GraphicsMagick aufgrund ungültiger Attribute (Fehlermeldung: “gm convert: attributes construct error”).

In dieser Arbeit werden einige Grundlagen erklärt. Es wird gezeigt, dass die Schwachstelle in der Praxis unter den in diesem Rahmen verwendeten Bedingungen nicht ausnutzbar ist. Um zu demonstrieren, dass die Schwachstelle theoretisch trotzdem kritisch sein kann, wird ein Fall konstruiert der dies zeigt. Im Anschluss wird erklärt, wie die Schwachstelle behoben wurde.

2 Hintergrund

2.1 Call Stack

Ein Stack ist eine LIFO (**L**ast **I**n **F**irst **O**ut) Datenstruktur, d.h. das letzte auf den Stack gelegte Element wird als erstes wieder vom Stack genommen. Ein Stack verfügt in der Regel über zwei Zugriffsfunktionen: **Push** zum Legen eines Elements auf den Stack und **Pop** zum Entfernen des letzten Elements von Stack. Diese Zugriffsfunktionen sind in der folgenden Abbildung grafisch dargestellt:

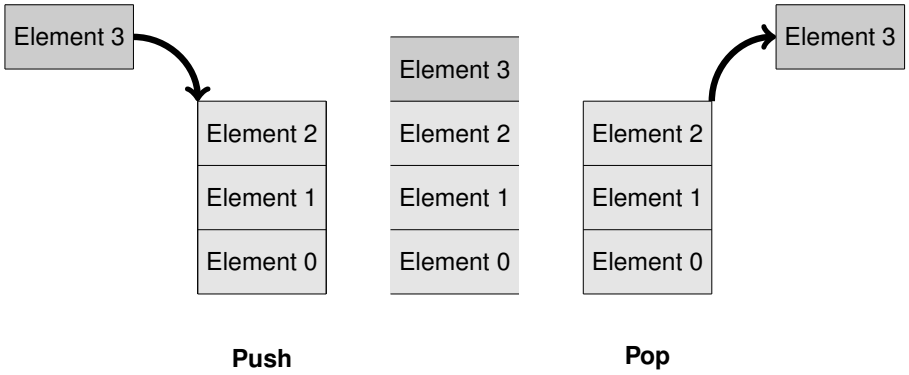


Abbildung 3: Darstellung der grundlegenden Zugriffsfunktionen eines Stack

Ein Stack kann auch zusätzlich über die Zugriffsfunktion **Peek** verfügen, welche das Element zurückgibt, welches durch **Pop** vom Stack entfernt werden würde, ohne dieses Element tatsächlich zu entfernen.

Der C-Standard [11] definiert Speicherdauern (“Storage Durations”), die angeben, wie lange Daten (z.B. Variablen) gültig sind und entsprechend im Speicher gehalten werden müssen. Es ist spezifiziert, dass die Lebenszeit von Objekten (nicht als “Objekt” im objektorientieren Sinne zu verstehen) von der Stelle des Programms abhängt, an der das Objekt deklariert wurde. Während der gesamten Lebenszeit ist garantiert, dass ein Objekt an der gleichen Adresse im Speicher liegt und seinen Wert nicht ändert (außer man überschreibt den Wert im Programm). Findet ein Zugriff auf ein Objekt außerhalb seiner Lebenszeit statt, ist das Verhalten im Standard nicht definiert.

Obwohl der Standard das nicht vorgibt, verwenden viele Compiler einen Stack für die praktische Umsetzung dieser im Standard geforderten Anforderungen. Der dafür verwendete Stack wird als **Call Stack** bezeichnet. Im Folgenden wird das generelle Verhalten des Call Stack beschrieben, um das Funktionsprinzip zu veranschaulichen. Im folgenden Abschnitt 2.2 “Optimierung des Call Stack” werden in der Praxis auftretende Abweichungen von dem hier beschriebenen Verhalten erklärt. Bei der x86 Architektur wächst der Stack von unten nach oben, d.h. das zuerst auf dem Stack abgelegte Element hat die höchste Adresse.

Die Verschachtelung von Funktionsaufrufen lässt sich mit Hilfe eines Stack sehr gut abbilden, da der Aufruf einer anderen Funktion praktisch dem Anlegen eines neuen Elements entspricht. Am Ende einer Funktion wird dieses Element wieder vom Stack entfernt. Wenn die aufgerufene Funktion eine weitere Funktion aufruft, wird dafür ein zusätzliches Element auf den Stack gelegt. Diese Elemente werden als **Stack Frame** bezeichnet.

```
1  void function3() {
2
3  }
4
5  void function2() {
6      function3();
7  }
8
9  void function1() {
10     function2();
11 }
12
13 int main(int argc, const char *argv[]) {
14     function1();
15     return 0;
16 }
```

Listing 1: Einfache Funktionsaufrufe in C

Der im vorherigen Listing 1 gezeigte Code sorgt dafür, dass bei jedem der verschachtelten Funktionsaufrufe ein weiteres Stack Frame auf den Call Stack gepusht wird. Ist eine Funktion beendet, wird das Stack Frame von Call Stack gepopt:

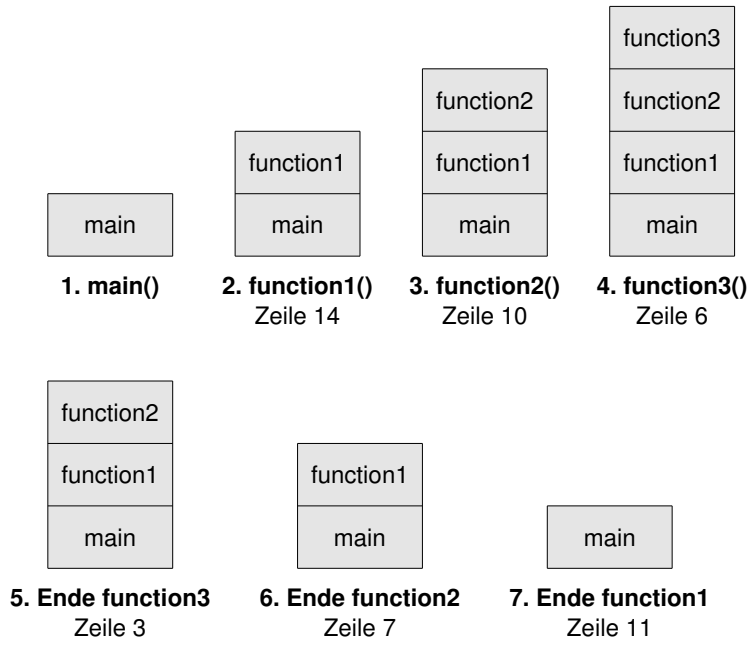


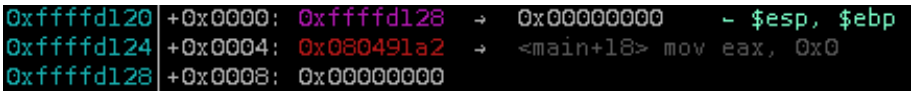
Abbildung 4: Darstellung eines vereinfachten Call Stack

Um die einzelnen Stack Frames zu verwalten, werden zwei Pointer verwendet: Der **Base Pointer**, der auf das Ende des StackFrame verweist und der **Stack Pointer**, der auf den Anfang des Stacks verweist. Da jeweils nur das erste Element des Stacks relevant ist (push legt ein weiteres Element darauf, Pop nimmt dieses Element vom Stack), entspricht der Anfang des aktuellen Stack Frames auch dem Anfang des Stack. Diese Pointer sind in CPU-Registern gespeichert. Bei einem x86 Rechner heißen diese Register **ebp** für den Base Pointer und **esp** für den Stack Pointer, bei einem x86–64 Rechner **rbp** für den Base Pointer und **rsp** für den Stack Pointer.

Zusätzlich gibt es einen **Instruction Pointer**, der bei x86 Rechnern in dem Register **eip**, bei x86–64 in dem Register **rip** abgelegt ist. Der Instruction Pointer enthält die Adresse der Instruktion, die als nächstes ausgeführt werden soll. Innerhalb einer Funktion liegen diese Instruktionen im Speicher hintereinander, d.h. der Instruction Pointer wird nach dem Ausführen einer Instruktion erhöht und zeigt dadurch automatisch auf die nächste Instruktion. Wird allerdings eine Funktion aufgerufen, muss der Instruction Pointer verändert werden, um auf die erste Instruktion der Funktion zu verweisen. Innerhalb dieser Funktion wird der Instruction Pointer nach jeder Instruktion wieder erhöht. Ist die Funktion beendet, muss der Instruction Pointer den alten Wert vor dem Aufruf der Funktion wieder laden, um an der richtigen Stelle mit der aufrufenden Funktion fortzufahren.

Wird eine Funktion aufgerufen, wird zuerst der Wert von eip auf den Stack gepusht, anschließend wird der Wert von ebp auf den Stack gepusht, da sonst die aktuellen Werte von Instruction Pointer und Base Pointer verloren gehen würden. Darauf hin wird der Wert des aktuellen Stack Pointers in den Base Pointer geschrieben. Der Base Pointer zeigt also auf die Adresse auf dem Stack, an der der Wert des alten Base Pointer gespeichert ist. Ist eine Funktion beendet, wird der gespeicherte Wert des Base Pointer vom Stack gepopt und in ebp gespeichert. Anschließend wird der Wert des Instruction Pointers vom Stack gepopt und in epi gespeichert.

(a) Zeile 14: ebp und esp zeigen auf 0xffffd128



(b) Zeile 10: ebp und esp zeigen auf 0xffffd120

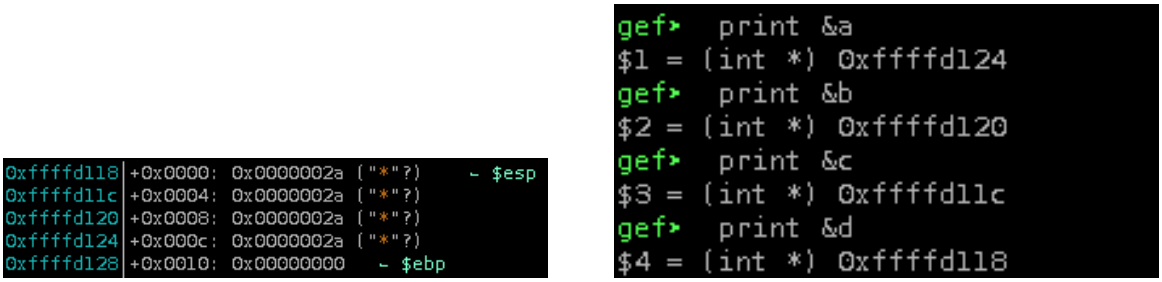
Abbildung 5: GDB-Ausgaben von Listing 1

Wie in Abbildung 5 (a) dargestellt, ist der Wert von ebp und esp 0xffffd128. In (b) ist zu sehen, dass der Wert der Adresse, auf die ebp und esp zeigen (ebp und esp zeigen auf 0xffffd120) 0xffffd128, also der alte Wert von ebp ist. In der Folgenden Zeile der gleichen Abbildung wird gezeigt, dass der Wert von eip gespeichert wurde. Wenn diese Funktion also beendet ist, wird der Wert 0x080491a2 in eip geladen, wodurch das Programm in der Main-Funktion fortgeführt wird.

Innerhalb eines Stack Frame werden die lokalen Variablen bei ihrer Deklaration auf den Stack gepusht, d.h. der Call Stack wächst dynamisch und die zuerst deklarierte Variable hat die höchste Adresse.

```
1 int main(int argc, const char *argv[]) {
2     int a = 42;
3     int b = 42;
4     int c = 42;
5     int d = 42;
6     return 0;
7 }
```

Listing 2: Variablendeklaration in C



(a) Stack Frame mit 4 int-Variablen

(b) Adressen der 4 int-Variablen

Abbildung 6: GDB-Ausgaben von Listing 2 an Zeile 6

Wie in Abbildung 6 (b) zu sehen ist, unterscheiden sich die Adressen der Variablen a, b, c und d um jeweils 4 Bytes, was der Größe eines Integers auf diesem Rechner entspricht. Für die Adressen gilt: $\&a > \&b > \&c > \&d$

2.2 Optimierung des Call Stack

Der Zugriff auf Variablen erfolgt häufig durch eine Adressierung der Form $\&var = ebp + offset$. Da einige Mikroprozessoren nur Offsets von 4 oder 6 Bit unterstützen [12], tritt bei der Adressierung lokaler Variablen mit einem größeren Offset zum Base Pointer das Problem auf, dass diese Variablen nicht direkt adressiert werden können. Um das Problem zu beheben, wird der Base Pointer in einem andren Register zwischengespeichert. Danach wird der Offset auf den Base Pointer addiert, auf die Variable zugegriffen, und darauf hin der ursprüngliche Wert wieder in den Base Pointer geschrieben. Dieses Vorgehen sorgt dafür, dass mehr Assembler-Code benötigt wird und die CPU mehr Instruktionen ausführen muss. Als Folge verschlechtert sich die Laufzeit des Programms.

Um die Laufzeit zu optimieren, werden, wie in [12] beschrieben, Optimierungen auf die Position der Variablen auf dem Stack angewandt. Dabei werden die Variablen in Abhängigkeit der Zugriffshäufigkeit und Variablengröße optimiert auf dem Stack abgelegt, wodurch die Reihen-

folge der Variablen von der im Abschnitt 2.1 beschriebenen Reihenfolge abweicht. GCC führt diese Optimierungen auch durch, wenn die Option **-O0** angewandt wurde, die Optimierungen abschaltet.

Außerdem wird der Stack Pointer beim Aufruf einer Funktion so weit verschoben, dass alle lokalen Variablen dieser Funktion auf den Stack passen. Der Stack wächst also nur dynamisch, wenn eine Funktion aufgerufen wird, allerdings nicht, wenn in einer Funktion Variablen deklariert werden. Diese Optimierung spart Instruktionen, da der Stack Pointer nur einmal beim Funktionsaufruf verändert werden muss. Es kann vorkommen, dass das Programm dadurch nicht so speichereffizient läuft wie es laufen könnte, da auch Platz für Variablen reserviert wird, die z.B. in einer If-Anweisung deklariert werden. Wenn die Anweisung nicht ausgeführt wird, da die Bedingung nicht zutrifft, wird also trotzdem Speicherplatz für die Variable reserviert, obwohl die Variable nicht benötigt wird.

```
1  void function1() {
2      int a = 42;
3      if(a == 23) {
4          int b[10];
5      }
6  }
7
8  int main (int argc, const char *argv[]) {
9      function1();
10     return 0;
11 }
```

Listing 3: Nicht ausgeführte Deklaration

```
0xffffd0f0 +0x0000: 0xf7f723bc  →  0xf7f731c0  →  0x00000000     ← $esp
0xffffd0f4 +0x0004: 0xffffd1c4  →  0xffffd37b
0xffffd0f8 +0x0008: 0xffffd1cc  →  0xffffd3b4
0xffffd0fc +0x000c: 0x080491e3  →  <__libc_csu_init+67> add edi, 0x1
0xffffd100 +0x0010: 0x00000001
0xffffd104 +0x0014: 0xffffd1c4  →  0xffffd37b
0xffffd108 +0x0018: 0xffffd1cc  →  0xffffd3b4
0xffffd10c +0x001c: 0x080491bb  →  <__libc_csu_init+27> lea esi, [ebx-0xf8]
0xffffd110 +0x0020: 0xf7fe41c0  →  <_dl_fini+0> push ebp
0xffffd114 +0x0024: 0x00000000
0xffffd118 +0x0028: 0x080491a9  →  <__libc_csu_init+9> add ebx, 0x2e57
0xffffd11c +0x002c: 0x0000002a ("*"?)
0xffffd120 +0x0030: 0xffffd128  →  0x00000000     ← $ebp
```

Abbildung 7: GDB-Ausgaben von Listing 4 an Zeile 6

Wie man an diesem Beispiel sehen kann, wird die Variable b niemals deklariert, da der Wert von a immer 42 und nie 23 ist. Somit ist es also nicht notwendig, Speicherplatz für b zu reservieren. Abbildung 7 zeigt allerdings, dass neben dem Speicherplatz für eine int-Variable mit dem Wert 42 (an Adresse 0xffffd11c) auch Speicherplatz für ein int-Array mit 10 Elementen reserviert wurde (an Adresse 0xffffd0f4 - 0xffffd118). Da der Speicher nicht initialisiert wurde, stehen darin noch alte Werte, die für dieses Beispiel nicht relevant sind.

Die von GCC immer angewendete Optimierung sorgt dafür, dass Instruktionen gespart werden, wodurch das Programm kleiner wird und weniger Rechenzeit benötigt. Allerdings gibt es auch Fälle, in denen diese Optimierung dafür sorgt, dass das Programm mehr Speicher reserviert als es benötigt. Weitere Optimierungen können diesem Verhalten entgegenwirken [13].

2.3 Endianness

Die Endianness [14] gibt in diesem Kontext an, in welcher Reihenfolge die Bytes im Speicher abgelegt werden. Prinzipiell unterscheidet man zwischen Little Endian und Big Endian. Bei Little Endian wird das Least Significant Bit (LSB) an der niedrigsten Speicheradresse und das Most Significant Bit (MSB) an der höchsten Speicheradresse abgelegt. Bei Big Endian ist es genau anders herum: das MSB liegt an der niedrigsten und das LSB an der höchsten Speicheradresse. Aus diesen verschiedenen Ansätzen ergeben sich besonders bei Datenübertragungen zwischen Systemen verschiedener Endianness Probleme [15].

Im Folgenden betrachten wir die Zahl 42 als int, hexadezimal 0x2a. Da ein int auf diesem System 4 Bytes groß ist, werden folgende Bytes zur Repräsentation der 42 verwendet: 0x00 0x00 0x00 0x2a.

- Big Endian:
  Bei Big Endian liegt das MSB an der niedrigsten Speicheradresse. In diesem Beispiel ist 0x00 das MSB und 0x2a das LSB. Die Bytes liegen also wie Folgt im Speicher:

Offset	Byte
0x00	0x00
0x04	0x00
0x08	0x00
0x0c	0x2a

Tabelle 1: Big Endian: Reihenfolge der Bytes im Speicher

- Little Endian:
  Bei Little Endian liegt das LSB an der niedrigsten Speicheradresse. Die Bytes liegen also wie Folgt im Speicher:

Offset	Byte
0x00	0x2a
0x04	0x00
0x08	0x00
0x0c	0x00

Tabelle 2: Little Endian: Reihenfolge der Bytes im Speicher

2.4 Buffer Overflow

C ist eine sehr hardwarenahe Sprache, weshalb im Gegensatz zu vielen anderen Sprachen direkt auf Speicheradressen zugegriffen wird. Es ist also möglich, ein Programm zu schreiben, das auf beliebige Speicheradressen zugreift. Der C-Standard [11] definiert einen Zugriff auf ein Array wie Folgt: $E1[E2] = (*((E1) + (E2)))$. Daraus folgt, dass ein Array des Typs T ein Pointer auf T ist. Der in diesem Beispiel durch $E2$ angegebene Offset entspricht $E2 \times sizeof(T)$ Bytes.

Da ein Pointer nur eine Adresse speichert und Arrays und Pointer praktisch äquivalent sind (bis auf wenige Ausnahmen), ist es also z.B. möglich, auf die 6. Stelle eines 6-Stelligen Array zuzugreifen (die Indizierung von Array beginnt bei 0, weshalb an der 6. Stelle das 7. Element liegen würde):

```
1 int main(int argc, const char *argv[]) {
2     int a[6];
3     int b = a[6];
4     return 0;
5 }
```

Listing 4: Zugriff auf ein Array außerhalb der Grenzen

Analog zum lesenden Zugriff außerhalb der Grenzen eines Array kann auch außerhalb dieser Grenzen geschrieben werden. Abhängig davon, in welchem Kontext sich das Array befindet, werden ggf. andere Variablen überschrieben. Das folgende Beispiel überschreibt z.B. den Wert von b mit 42:

```c
#include<stdio.h>

int main(int argc, const char *argv[]) {
    int a[6];
    int b = 0;
    printf("b = %d\n", b);
    a[6] = 42;
    printf("b = %d\n", b);
    return 0;
}
```

Listing 5: Schreiben außerhalb der Grenzen eines Array

Der in Listing 5 dargstellte Code führt zu folgender Ausgabe:

```
b = 0
b = 42
```

Listing 6: Ausgabe des in Listing 5 dargestellten Codes (ohne BOF)

Generell kann dieses Problem sowohl auf dem Stack als auch auf dem Heap auftreten. Man spricht dann von einem Stack-based Buffer Overflow oder einem Heap-based Buffer Overflow. Im Folgenden wird der Stack-based Buffer Overflow genauer betrachtet.

Praktisch bedeutet das Schreiben vor oder nach den Grenzen eines Array, dass die Werte, die vor oder nach diesem Array auf dem Stack liegen, überschrieben werden. In Abschnitt 2.2 wurde der grundlegende Aufbau des Call Stack beschrieben. Wenn es durch einen Buffer Overflow auf dem Stack möglich ist, die Elemente vor oder nach einem Array zu überschreiben, können wie in Listing 5 dargestellt Variablen überschrieben werden. Außerdem besteht die Möglichkeit, den Base Pointer und daraufhin den Instruction Pointer zu überschreiben und dadurch die Ausführung des Programms an einer anderen Stelle fortzusetzen:

```c
#include<stdio.h>

void function2() {
    printf("Function 2 called.\n");
}

void function1() {
    char buffer[8];
    gets(buffer);
    printf("You typed: %s\n", buffer);
}

int main(int argc, const char *argv[]) {
    function1();
}
```

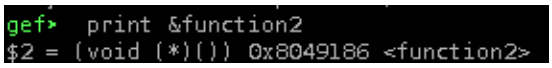
Listing 7: Einfacher Buffer Overflow

Da die Funktion gets() nicht prüft, ob die Eingabe die Länge des Buffers überschreitet, kann ein beliebig langer Text eingegeben werden. Ist der Text länger als der Buffer, werden die Elemente, die auf dem Stack hinter dem Buffer liegen, überschrieben. In Zeile 9 sieht der Stack folgendermaßen aus:

Abbildung 8: GDB-Ausgaben von Listing 7 an Zeile 9

Der Buffer ist 8 Bytes ($8 \times sizeof(char)$) groß und liegt an der Speicherstelle 0xffffd108:

(a) Adresse von buffer



(b) Adresse von function2

Abbildung 9: GDB-Ausgabe der Adressen von Listing 7

Nun kann die Rücksprungadresse der Funktion function1 (an Speicherstelle 0xffffd11c) durch die Adresse von function2 überschrieben werden. Dafür werden einige A eingegeben, um den Speicher von 0xffffd108 bis 0xffffd11b zu überschreiben. Anschließend wird die Adresse von function2 eingegeben. Da die hier verwendete Rechnerarchitektur Little Endian verwendet, muss die Adresse in inverser Reihenfolge eingegeben werden. Insgesamt ergibt sich also folgende Eingabe:

```
1 AAAAAAAAAAAAAAAAAAAA\x86\x91\x04\x08
```

Listing 8: Eingabe für Listing 5

Nun wurden die auf buffer folgenden Speicherbereiche bis einschließlich der Rücksprungadresse überschrieben:

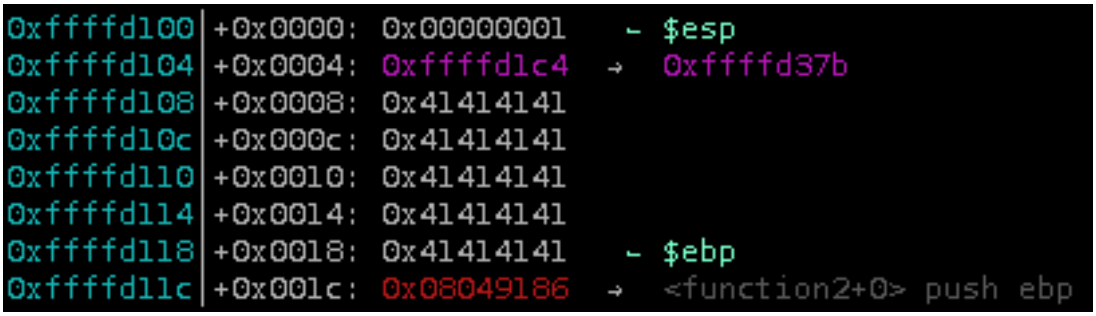


Abbildung 10: GDB-Ausgaben von Listing 7 nach der Eingabe von 8

```
1 You typed: AAAAAAAAAAAAAAAAAAAA??
2 Function 2 called.
```

Listing 9: Ausgabe des in Listing 5 dargestellten Codes (mit BOF)

Durch das Überschreiben der Rücksprungadresse wurde der Verlauf des Programms verändert, wodurch function2 aufgerufen wurde.

Da der Benutzer den eingegebenen String kontrolliert, kann er mit dem String auch Shellcode eingeben. Dabei handelt es sich um die binären Instruktionen, die ausgeführt werden sollen, um ein bestimmtes Ziel zu erreichen. Häufig besteht das Ziel darin, eine Shell auf dem System zu bekommen. Durch das Überschreiben der Rücksprungadresse mit der Adresse, an der der Shellcode liegt, kann der Angreifer also beliebigen Code ausführen.

3 CVE–2019–11005: Stack-based Buffer Overflow in GraphicsMagick

3.1 Details der Schwachstelle

Die Schwachstelle [7] in der Funktion SVGStartElement befindet sich in einer großen stark verschachtelten Mehrfachauswahl, die die Attribute eines öffnenden Tag in einer SVG-Datei auswertet. Der Fehler tritt bei der Auswertung des Attributs “font-family” auf:

```
1  if (LocaleCompare(keyword,"font-family") == 0)
2  {
3      /*
4         Deal with Adobe Illustrator 10.0 which double-quotes
5         font-family.  Maybe we need a generalized solution for
6         this.
7         */
8      if ((value[0] == '\'') && (value[strlen(value)-1] == '\''))
9      {
10          char nvalue[MaxTextExtent];
11          (void) strlcpy(nvalue,value+1,sizeof(nvalue));
12          nvalue[strlen(nvalue)-1]='\0';
13          MVGPrintf(svg_info->file,"font-family '%s'\n",nvalue);
14     }
15     else
16     {
17          MVGPrintf(svg_info->file,"font-family '%s'\n",value);
18     }
19     break;
20 }
```

Listing 10: Code für die Auswertung des Attributs “font-family”

Wie dem Kommentar zu entnehmen ist, setzt Adobe Illustrator 10.0 die font-family in Quotes, wodurch diese Schriftart auf dem Rechner nicht gefunden werden würde. Beispielsweise ist die Schriftart Open Sans installiert, allerdings nicht die Schriftart ’Open Sans’. Um dieses Problem zu beheben, muss GraphicsMagick also erkennen, ob die Schriftart in Quotes steht oder nicht. Diese Unterscheidung findet in Zeile 8 Statt: Wenn sowohl das erste als auch das letzte Zeichen der Schriftart ’ ist, wird der Name als gequotet behandelt.

Wenn die Schriftart gequotet ist, wird ein neues Char-Array nvalue angelegt, welches MaxTextExtend = 2053 Elemente groß ist. Anschließend wird mit der Funktion strlcopy der Inhalt von value +1 (value ohne das erste Zeichen, welches ja ein ’ ist) in nvalue kopiert. Dabei wird auch geprüft, dass nicht über die Grenzen von nvalue geschrieben wird, indem die Anzahl der maximal zu kopierenden Zeichen (sizeof(nvalue) = 2053) angegeben wird.

Im nächsten Schritt wird das letzte Zeichen von nvalue (welches ja auch ein ’ ist) mit dem Nullbyte 0x0 überschrieben. Dadurch ist in nvalue der Wert von value ohne die Quotes gespeichert.

Ist die Schriftart nicht gequotet, wird direkt mit der Verarbeitung von value fortgefahren, sonst wird der eben erstellte Wert nvalue verwendet, bei dem die Quotes entfernt wurden.

Wenn die Schriftart z.B. 'Open Sans' ist, dann trifft die If-Bedingung zu und die Variable nvalue wird angelegt. Anschließend wird value ab der Stelle 1 in nvalue kopiert. Entsprechend wird das erste ' nicht übernommen und in nvalue steht nun Open Sans'. Daraufhin wird nvalue an der Stelle sizeof(nvalue) - 1 mit 0x00 überschrieben. In nvalue stehen momentan 9 Zeichen, entsprechend wird nvalue[9-1] = nvalue[8] mit 0x00 überschrieben, wodurch der String effektiv um ein Zeichen kürzer wird. In nvalue steht nun also der String Open Sans. Dieser wird darauf hin genau wie ein nicht gequoteter Wert verarbeitet.

Die Schwachstelle besteht darin, dass die Entwickler übersehen haben, dass jemand den String ' als font-family angeben könnte. In diesem Fall wird value ab der Stelle 1 in nvalue kopiert, wodurch in nvalue[0] das Nullbyte 0x00 steht. Im nächsten Schritt wird nvalue[sizeof(nvalue) - 1] = 0x00 gesetzt. sizeof(nvalue) ist momentan allerdings 0. nvalue[0-1] = nvalue[-1] wird nun entsprechend mit 0x00 überschrieben. nvalue[-1] ist die Adresse direkt vor nvalue, es wird also das Byte im Speicher überschrieben, das direkt vor nvalue liegt.

Normalerweise wäre dieser Fehler nicht kritisch, da nvalue die letzte deklarierte Variable ist. Vor nvalue liegt also (noch) nichts auf dem Stack. Es würde also eine Speicherstelle überschrieben werden, die evtl. im weiteren Verlauf einer Variable zugewiesen wird. Wenn eine Variable deklariert und nicht initialisiert wird, ist nicht garantiert, dass in der Variable ein bestimmter Wert steht, weshalb die Variable ohnehin initialisiert werden würde, was die Änderung überschreiben würde.

Das Problem besteht an dieser Stelle darin, dass die Anordnung der Elemente auf dem Stack optimiert wird [12], wodurch vor nvalue tatäschlich eine Variable liegt. Durch die im Folgenden beschriebenen Vorgehensweise wurde herausgefunden und bestätigt, welche Variable mit den im Rahmen dieser Arbeit verwendeten Compilerflags (CFLAGS=-g -fno-stack-protector -z execstack -no-pie -m32) im Speicher vor nvalue liegt:

Im ersten Schritt wurde die Funktion SVGStartElement mit einem Disassembler in Assembler-Code übersetzt. Anschließend wurde die Speicherstelle von nvalue herausgefunden:

```
1          (void) strlcpy(nvalue,value+1,sizeof(nvalue));
2  820f8b4:   8b 45 f4              mov    -0xc(%ebp),%eax
3  820f8b7:   83 c0 01              add    $0x1,%eax
4  820f8ba:   83 ec 04              sub    $0x4,%esp
5  820f8bd:   68 05 08 00 00        push   $0x805
6  820f8c2:   50                    push   %eax
7  820f8c3:   8d 85 d8 de ff ff     lea    -0x2128(%ebp),%eax
8  820f8c9:   50                    push   %eax
9  820f8ca:   e8 30 dc ee ff        call   80fd4ff <MagickStrlCpy>
10 820f8cf:   83 c4 10              add    $0x10,%esp
```

Listing 11: Assembler-Code der Instruktion von Zeile 11 in Listing 10

Aus Listing 11 geht hervor, dass nvalue an der Speicherstelle **%ebp-0x2128** liegt. Um herauszufinden, welche Variable der Funktion SVGStartElement direkt vor dieser Speicherstelle liegt, war es notwendig, alle Statements der Form %ebp-0xDEADBEEF zu finden. Da der Stack nach oben wächst, liegen kleinere Adressen weiter vor dem Base Pointer. Nun musste die Zeile gefunden werden, in der der nächstkleinere Offset zu %ebp vorkommt. Um diese Zeile zu finden, wurde ein Python-Script geschrieben, welches die relevanten Zeilen nach Offset sortiert:

```
1  with open("lines.txt") as f:
2      lines = f.readlines()
3  offsets = {}
4  keys = []
5  for line in lines:
6      try:
7          hvalue = "0x" + line.split(" -0x")[1].split("(%ebp")[0]
8      except IndexError:
9          hvalue = "0x" + line.split(",-0x")[1].split("(%ebp")[0]
10     value = int(hvalue, 16)
11     if value in keys:
12         offsets[value] += " " + line
13     else:
14         offsets[value] = line
15         keys.append(value)
16 keys = sorted(keys)
17 for key in keys:
18     pass
19     print(str(key) + ":\n " + offsets[key])
```

Listing 12: Python-Script zum Sortieren der relevanten Zeilen nach Offset

Die relevanten Zeilen wurden vorher mit Hilfe von egrep in die Datei “lines.txt” geschrieben.

Die Ausgabe dieses Scripts liefert die Zeilen, die ein Statement nach dem oben beschriebenen Schema enthalten und sortiert diese aufsteigend nach dem Betrag des Offsets (da nur negative Offsets berücksichtigt wurden, steht das Relevante Statement also unter den Statements mit **%ebp-0x2128**:

```
8488:
  820f8c3:  8d 85 d8 de ff ff      lea    -0x2128(%ebp),%eax
  820f8d5:  8d 85 d8 de ff ff      lea    -0x2128(%ebp),%eax
  820f8e7:  c6 84 05 d8 de ff ff   movb   $0x0,-0x2128(%ebp,%eax,1)
  820f8f7:  8d 95 d8 de ff ff      lea    -0x2128(%ebp),%edx
  820fa5b:  8d 85 d8 de ff ff      lea    -0x2128(%ebp),%eax
  820fb38:  8b 85 d8 de ff ff      mov    -0x2128(%ebp),%eax
  82102b7:  dd 9d d8 de ff ff      fstpl  -0x2128(%ebp)
  82103b8:  dd 85 d8 de ff ff      fldl   -0x2128(%ebp)

8492:
  820fbc2:  89 85 d4 de ff ff      mov    %eax,-0x212c(%ebp)
```

Listing 13: Relevanter Teil der von Listing 12 generierten Ausgabe

Aus der Ausgabe lässt sich schließen, dass sich die Variable, die vor nvalue auf dem Stack liegt, an der Speicherstelle **%ebp-0x212c** befindet. Im Anschluss kann die Assembler-Datei der Funktion SVGStartElement nach dieser Adresse durchsucht werden. Die Suche liefert folgenden Bereich:

```
                          current=transform;
 820fb38:   8b 85 d8 de ff ff      mov    -0x2128(%ebp),%eax
 820fb3e:   89 85 a8 de ff ff      mov    %eax,-0x2158(%ebp)
 820fb44:   8b 85 dc de ff ff      mov    -0x2124(%ebp),%eax
 820fb4a:   89 85 ac de ff ff      mov    %eax,-0x2154(%ebp)
 820fb50:   8b 85 e0 de ff ff      mov    -0x2120(%ebp),%eax
 820fb56:   89 85 b0 de ff ff      mov    %eax,-0x2150(%ebp)
 820fb5c:   8b 85 e4 de ff ff      mov    -0x211c(%ebp),%eax
 820fb62:   89 85 b4 de ff ff      mov    %eax,-0x214c(%ebp)
 820fb68:   8b 85 e8 de ff ff      mov    -0x2118(%ebp),%eax
 820fb6e:   89 85 b8 de ff ff      mov    %eax,-0x2148(%ebp)
 820fb74:   8b 85 ec de ff ff      mov    -0x2114(%ebp),%eax
 820fb7a:   89 85 bc de ff ff      mov    %eax,-0x2144(%ebp)
 820fb80:   8b 85 f0 de ff ff      mov    -0x2110(%ebp),%eax
 820fb86:   89 85 c0 de ff ff      mov    %eax,-0x2140(%ebp)
 820fb8c:   8b 85 f4 de ff ff      mov    -0x210c(%ebp),%eax
 820fb92:   89 85 c4 de ff ff      mov    %eax,-0x213c(%ebp)
 820fb98:   8b 85 f8 de ff ff      mov    -0x2108(%ebp),%eax
 820fb9e:   89 85 c8 de ff ff      mov    %eax,-0x2138(%ebp)
 820fba4:   8b 85 fc de ff ff      mov    -0x2104(%ebp),%eax
 820fbaa:   89 85 cc de ff ff      mov    %eax,-0x2134(%ebp)
 820fbb0:   8b 85 00 df ff ff      mov    -0x2100(%ebp),%eax
 820fbb6:   89 85 d0 de ff ff      mov    %eax,-0x2130(%ebp)
 820fbbc:   8b 85 04 df ff ff      mov    -0x20fc(%ebp),%eax
 820fbc2:   89 85 d4 de ff ff      mov    %eax,-0x212c(%ebp)
```

Listing 14: Bereich mit dem Statement %ebp-0x212c

Nun kann im C-Code nach dem Statement **current=transform;** gesucht werden. Die Suche ergibt zwei Treffer. Aus dem Kontext kann allerdings geschlossen werden, dass die betreffende Stelle in Zeile 1811 der Datei coders/svg.c ist:

```
AffineMatrix
    affine,
    current,
    transform;

IdentityAffine(&transform);
(void) LogMagickEvent(CoderEvent,GetMagickModule()," ");
tokens=GetTransformTokens(context,value,&number_tokens);
if ((tokens != (char **) NULL) && (number_tokens > 0))
{
    for (j=0; j < (number_tokens-1); j+=2)
    {
        keyword=(char *) tokens[j];
        if (keyword == (char *) NULL)
            continue;
        value=(char *) tokens[j+1];
        (void) LogMagickEvent(CoderEvent,GetMagickModule(),
                "    %.1024s: %.1024s",keyword,value);
        current=transform;
```

Listing 15: Stelle des Zugriffs auf die Variable vor nvalue im C-Code

Bei der Variable muss es sich also um einen Teil der Struktur AffineMatrix handeln. In der Datei magick/image.h ist diese Datenstruktur wie Folgt definiert:

```
typedef struct _AffineMatrix
{
  double
    sx,
    rx,
    ry,
    sy,
    tx,
    ty;
} AffineMatrix;
```

Listing 16: Definition von AffineMatrix

Da die Elemente in einem Struct in der Reihenfolge angeordnet werden, in der sie deklariert sind, handelt es sich bei der Variable also um den Double current->ty. Der Schreibzugriff auf das Byte vor nvalue überschreibt also das letzte Byte von current->ty. Da sich der Scope von current und nvalue nicht überlagert, ist die Schwachstelle in dem im Rahmen dieser Arbeit untersuchten Fall nicht kritisch.

3.2 Ausnutzung der Schwachstelle

Dieser Abschnitt besteht aus zwei Teilbereichen: Im ersten Teilbereich wird mit Hilfe des GNU-Debuggers gezeigt, dass die im vorherigen Abschnitt aufgestellten Behauptungen, current->ty sei die Variable, die vor nvalue um Speicher liegt und das Überschreiben sei nicht kritisch, zutreffen. Im zweiten Teilbereich wird der Quellcode von GraphicsMagick so verändert, dass vor nvalue ein Pointer liegt, der darauf hin dereferenziert wird. Dadurch soll demonstriert werden, dass die Schwachstelle unter Umständen durchaus kritisch sein kann, da sie zu einem Absturz oder nicht definierten Verhalten führen kann.

3.2.1 Nachweis der in Abschnitt 3.1 aufgestellten Behauptungen

Wie in Listing 15 zu sehen ist, wird die Variable current durch eine Zuweisung mit den Werten der Variable transform überschrieben. Es gibt keine weiteren schreibenden Zugriffe auf diese Variable innerhalb dieses Scope. Die Variable transform wird durch die Funktion IdentityAffine in der Datei magick/gem.c initialisiert:

```
1 MagickExport void IdentityAffine(AffineMatrix *affine)
2 {
3   assert(affine != (AffineMatrix *) NULL);
4   (void) memset(affine,0,sizeof(AffineMatrix));
5   affine->sx=1.0;
6   affine->sy=1.0;
7 }
```

Listing 17: Funktion IdentityAffine zum Initialisieren einer AffineMatrix

Mit Memset werden alle Bytes aller Komponenten des Struct auf 0 gesetzt. Anschließend werden sx und sy auf 1.0 gesetzt. Da im Folgenden keine schreibenden Zugriffe auf die Variable current erfolgen, ist jedes Byte von current->ty also 0.

Im weiteren Verlauf wird die folgende SVG-Datei betrachtet:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <svg height="30" width="200">
3    <linearGradient id="linearGradient6593-0" gradientUnits="
       userSpaceOnUse" x1="74.658" y1="-398.92" x2="75.519" y2="-485.7"
        gradientTransform="matrix(1.0069 0 0 1.19 1.4571 709.77)">
4      <stop id="stop6595" stop-color="#be245a" offset="0"/>
5      <stop id="stop6600" stop-color="#e46e6f" offset=".48408"/>
6      <stop id="stop6597" stop-color="#f1a769" offset="1"/>
7    </linearGradient>
8    <text x="0" y="15" fill="red" dx="50%" font-family="'Helvetica, sans
       -serif'">No Problem here</text>
9    <text x="0" y="15" fill="red" dx="50%" font-family="'">BOF here</
       text>
10 </svg>
```

Listing 18: SVG-Datei zum Ausnutzen der Schwachstelle (exploit.svg)

Im GNU-Debugger wurden Breakpoints an Zeile 19 von Listing 15 und an Zeile 12 von Listing 10 erstellt. Durch die in Listing 18 dargestellte SVG wird zuerst der erste Breakpoint und dann zwei mal der zweite Brekpoint durchlaufen. Anschließend wurde die Kommandozeilenanwendung gm mit den Parametern “convert exploit.svg exploit.png” aufgerufen.

Beim Erreichen des ersten Breakpoint wird die aktuelle Adresse von current ausgegeben:

```
gef➤  print &current
$1 = (AffineMatrix *) 0xffff69a0
```

Abbildung 11: GDB-Ausgabe der Adresse von current

Anschließend werden beginnend von der Adresse von current die folgenden 6 qwords ausgegeben (ein double nimmt auf diesem System 8 Bytes ein, weshalb die Größe eines double einem qword entspricht). Daraufhin wird der nächste Schritt ausgeführt, wodurch der Wert von transform in current übernommen wird. Darauf folgend wird die Ausgabe der 6 qwords wiederholt:

```
gef➤  hexdump qword &current 6
0xffff69a0 +0x0000   0x0000000000000000
0xffff69a8 +0x0008   0x0000000000000000
0xffff69b0 +0x0010   0x0000000000000000
0xffff69b8 +0x0018   0x0000000000000000
0xffff69c0 +0x0020   0x0000000000000000
0xffff69c8 +0x0028   0x0000000000000000
```

(a) Vor der Zuweisung

```
gef➤  hexdump qword &current 6
0xffff69a0 +0x0000   0x3ff0000000000000
0xffff69a8 +0x0008   0x0000000000000000
0xffff69b0 +0x0010   0x0000000000000000
0xffff69b8 +0x0018   0x3ff0000000000000
0xffff69c0 +0x0020   0x0000000000000000
0xffff69c8 +0x0028   0x0000000000000000
```

(b) Nach der Zuweisung

Abbildung 12: GDB-Ausgaben der 6 qwords/doubles der AffineMatrix current

Offensichtlich wurden der erste und der vierte double auf den Wert 0x3ff0000000000000 gesetzt, was der double-Repräsentation von 1.0 entspricht. Die anderen Werte wurden nicht verändert und stehen auf 0x0000000000000000. Da es durch den Programmablauf bedingt nicht möglich ist, den Wert von ty zu verändern, wird diese Veränderung zu Demonstrationszwecken manuell vorgenommen:

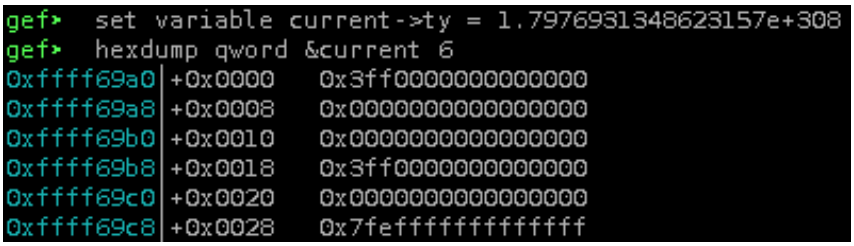


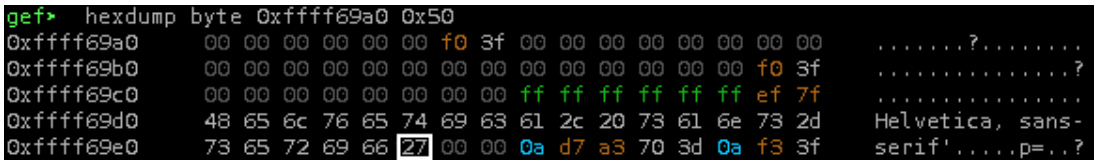
Abbildung 13: Überschreiben von ty mit GDB

In dem Hexdump in Abbildung 13 ist zu sehen, dass der Wert von ty nun 0x7fefffffffffffff ist.

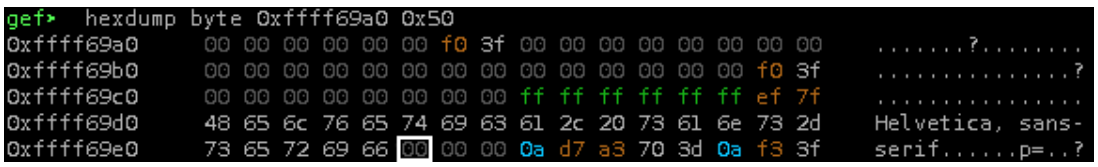
Nun wird der Ablauf des Programms bis zum nächsten Breakpoint fortgesetzt. Anschließend folgt die Ausgabe der Adresse von nvalue:

Abbildung 14: GDB-Ausgabe der Adresse von nvalue

Die Ausgabe des Hexdumps der Adresse von current (diesmal allerdings in Byte-Darstellung) zeigt, dass der Name der Schriftart ohne das erste aber mit dem letzten Quote im Speicher direkt auf den Wert 0x7fefffffffffffff, also current->ty, folgt. Die inverse Anordnung der Bytes von current->ty lässt sich damit erklären, dass dieser Rechner Little Endian (s. Abschnitt 2.3) verwendet:



(a) Vor der Überschreibung des letzten Quote



(b) Nach der Überschreibung des letzten Quote

Abbildung 15: GDB-Ausgaben der Bytes von current und nvalue

Durch das Überschreiben des letzten Zeichens von nvalue (das Zeichen vor 0x00) durch 0x00 wird das letzte Quote überschrieben. In nvalue steht nun der Name der angegebenen Schriftart ohne Quotes.

Im Anschluss wird der Ablauf des Programms bis zum nächsten Breakpoint fortgesetzt. Nun ist der Name der zu verarbeitenden Schriftart ’, was zu dem in Abschnitt 3.1 beschriebenen Verhalten führt. Dies lässt sich mit Hilfe des Hexdump nachvollziehen:

```
gef➤  hexdump byte 0xffff69a0 0x50
0xffff69a0     00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 00 00    .......?........
0xffff69b0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0 3f    ...............?
0xffff69c0     00 00 00 00 00 00 00 00 ff ff ff ff ff ff ef 7f    ................
0xffff69d0     00 65 6c 76 65 74 69 63 61 2c 20 73 61 6e 73 2d    .elvetica, sans-
0xffff69e0     73 65 72 69 66 00 00 00 0a d7 a3 70 3d 0a f3 3f    serif......p=..?
```

(a) Vor der Überschreibung des letzten Quote

```
gef➤  hexdump byte 0xffff69a0 0x50
0xffff69a0     00 00 00 00 00 00 f0 3f 00 00 00 00 00 00 00 00    .......?........
0xffff69b0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0 3f    ...............?
0xffff69c0     00 00 00 00 00 00 00 00 ff ff ff ff ff ff ef 00    ................
0xffff69d0     00 65 6c 76 65 74 69 63 61 2c 20 73 61 6e 73 2d    .elvetica, sans-
0xffff69e0     73 65 72 69 66 00 00 00 0a d7 a3 70 3d 0a f3 3f    serif......p=..?
```

(b) Nach der Überschreibung des letzten Quote

Abbildung 16: GDB-Ausgaben der Bytes von current und nvalue (nvalue ist leer)

Wie gut zu erkennen ist, wurde das letzte Byte von current->ty (0x7f) mit 0x00 überschrieben. Das könnte theoretisch zu Problemen führen, wenn im weiteren Verlauf des Programms ein Zugriff auf current->ty stattfinden würde. Da dieser Code allerdings außerhalb des Scope von current liegt, müsste die Variable ohnehin neu initialisiert werden, wodurch diese Änderung wieder überschrieben werden würde. Außerdem war der ursprüngliche Wert von current->ty 0x0000000000000000, weshalb das Byte 0x00 mit 0x00 überschrieben werden würde, was keine praktischen Auswirkungen hat.

3.2.2 Demonstration der theoretischen Ausnutzung durch Anpassen des Programmcodes

Um eine mögliche Ausnutzung der Schwachstelle zu demonstrieren, wurde der Code von Listing 10 wie Folgt angepasst:

```
if (LocaleCompare(keyword,"font-family") == 0)
{
    /*
       Deal with Adobe Illustrator 10.0 which double-quotes
       font-family.  Maybe we need a generalized solution for
       this.
       */
    if ((value[0] == '\'') && (value[strlen(value)-1] == '\''))
    {
        typedef struct {
            int *a;
            char nvalue[MaxTextExtent];
            int b;
        } Structure;
        Structure s;

        s.b = 42;
        s.a = &(s.b);

        printf("%d\n", *(s.a));

        (void) strlcpy(s.nvalue,value+1,sizeof(s.nvalue));
        s.nvalue[strlen(s.nvalue)-1]='\0';

        printf("%d\n", *(s.a));

        MVGPrintf(svg_info->file,"font-family '%s'\n",s.nvalue);
    }
    else
    {
        MVGPrintf(svg_info->file,"font-family '%s'\n",value);
    }
    break;
}
```

Listing 19: Angepasster Code für die Auswertung des Attributs “font-family”

Die Ausnutzung der Schwachstelle funktioniert nur, wenn der Pointer direkt vor nvalue im Speicher liegt. Um zu verhindern, dass der Pointer durch die in Abschnitt 2.2 beschriebenen Optimierungen an einer anderen Stelle im Speicher angelegt wird, wurde ein Struct verwendet, welches sicher stellt, dass die Variablen in der Reihenfolge ihrer Deklaration im Speicher abgelegt werden. Im Gegensatz zu der in Abschnitt 2.1 beschriebenen Reihenfolge von Variablen auf dem Stack ist die Reihenfolge der Variablen innerhalb eines Struct nicht invers, da das gesamte

Struct auf den Call Stack gepusht wird. Das bedeutet, dass $\&a < \&nvalue < \&b$.

Auch in diesem Abschnitt wird die in Listing 18 dargestellte SVG-Datei verwendet und mit dem Kommandozeilentool gm in eine png umgewandelt. Der Breakpoint wird an Zeile 23 von Listing 19 gesetzt. Daraufhin wird das Programm mit den Parametern “convert exploit.svg exploit.png” aufgerufen.

```
gef➤  hexdump byte &(s,a) 0x40
0xffff69c8     d4 71 ff ff 48 65 6c 76 65 74 69 63 61 2c 20 73    .q..Helvetica, s
0xffff69d8     61 6e 73 2d 73 65 72 69 66 27 00 70 3d 0a f3 3f    ans-serif'.p=..?
0xffff69e8     8e 06 f0 16 48 50 f7 3f 5c 8f c2 f5 28 2e 86 40    ....HP.?\...(..@
0xffff69f8     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

(a) Vor der Überschreibung des letzten Quote

```
gef➤  hexdump byte &(s,a) 0x40
0xffff69c8     d4 71 ff ff 48 65 6c 76 65 74 69 63 61 2c 20 73    .q..Helvetica, s
0xffff69d8     61 6e 73 2d 73 65 72 69 66 00 00 70 3d 0a f3 3f    ans-serif..p=..?
0xffff69e8     8e 06 f0 16 48 50 f7 3f 5c 8f c2 f5 28 2e 86 40    ....HP.?\...(..@
0xffff69f8     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

(b) Nach der Überschreibung des letzten Quote

Abbildung 17: GDB-Ausgaben der Bytes des Pointers und nvalue

Da der Pointer in Little Endian Codierung im Speicher liegt, ist die Adresse, auf die der Pointer zeigt, 0xffff71d4. Da im ersten Durchlauf am Breakpoint der Name der Schriftart gültig ist, verhält sich das Programm wie erwartet. Danach läuft das Programm bis zum nächsten Breakpoint. Nun ist der Name der Schriftart ungültig. In diesem Fall ergibt sich folgender Hexdump:

```
gef➤  hexdump byte &(s,a) 0x40
0xffff69c8     d4 71 ff ff 00 65 6c 76 65 74 69 63 61 2c 20 73    .q...elvetica, s
0xffff69d8     61 6e 73 2d 73 65 72 69 66 00 00 70 3d 0a f3 3f    ans-serif..p=..?
0xffff69e8     8e 06 f0 16 48 50 f7 3f 5c 8f c2 f5 28 2e 86 40    ....HP.?\...(..@
0xffff69f8     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

(a) Vor der Überschreibung des letzten Quote

```
gef➤  hexdump byte &(s,a) 0x40
0xffff69c8     d4 71 ff 00 00 65 6c 76 65 74 69 63 61 2c 20 73    .q...elvetica, s
0xffff69d8     61 6e 73 2d 73 65 72 69 66 00 00 70 3d 0a f3 3f    ans-serif..p=..?
0xffff69e8     8e 06 f0 16 48 50 f7 3f 5c 8f c2 f5 28 2e 86 40    ....HP.?\...(..@
0xffff69f8     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

(b) Nach der Überschreibung des letzten Quote

Abbildung 18: GDB-Ausgaben der Bytes des Pointers und nvalue (nvalue ist leer)

Wie zu erwarten war, wurde das letzte Byte des Pointers im Speicher mit 0x00 überschrieben. Entsprechend zeigt der Pointer nun auf 0x00ff71d4. Da diese Speicheradresse außerhalb des Bereichs von gm liegt, bekommt das Programm Signal 11, SIGSEGV. Das Programm fängt die Segmentation Fault ab und wird dann mit SIGABRT beendet. Im folgenden Listing wird die Ausgabe des Programmaufrufs dargestellt:

```
42
42
42
gm convert: abort due to signal 11 (SIGSEGV) "Segmentation Fault"...
fish: "gm convert exploit.svg exploit.png" terminated by signal
    SIGABRT (Abort)
```

Listing 20: Ausgabe des angepassten Codes für die Auswertung des Attributs “font-family”

Offensichtlich ist es also durchaus möglich, die Schwachstelle auszunutzen und das Programm zum Absturz zu bringen. Allerdings ist die Voraussetzung dafür, dass direkt vor nvalue ein Pointer liegt, der im weiteren Verlauf dereferenziert wird.

3.3 Verteidigung der Schwachstelle

Die Schachstelle wurde in Commit b6fb77d7d54d behoben. Der in Listing 10 gezeigt Code wurde wie Folgt verändert:

```
1  if (LocaleCompare(keyword,"font-family") == 0)
2  {
3      /*
4         Deal with Adobe Illustrator 10.0 which double-quotes
5         font-family.  Maybe we need a generalized solution for
6         this.
7         */
8      int value_length;
9      if ((value[0] == '\'') && ((value_length=(int) strlen(value)) > 2)
10             && (value[value_length-1] == '\''))
11     {
12         MVGPrintf(svg_info->file,"font-family '%.*s'\n",
13                 (int)(value_length-2),value+1);
14     }
15     else
16     {
17         MVGPrintf(svg_info->file,"font-family '%s'\n",value);
18     }
19     break;
20 }
```

Listing 21: Code für die Auswertung des Attributs “font-family” mit Patch

Es wurde eine neue Variable value_length deklariert und mit strlen(value) initialisiert. Der Name der Font zählt nun nur noch als gequotet, wenn das erste und das letzte Zeichen ein ’ sind sowie der Name mindestens zwei Zeichen lang ist (ein einfaches ’ würde also durch den Else-Zweig abgedeckt werden).

Zusätzlich wurde der Workaround mit nvalue durch die Zeilen 12 und 13 ersetzt. Durch die Verwendung von ‘%.*s’ kann als Argument angegeben werden, wie viele Zeichen ausgegeben

werden sollen. In dem darauf folgenden Parameter wird angegeben, dass value_length - 2 Zeichen ausgegeben werden sollen. Der nächste Parameter übergibt den Pointer auf das erste Zeichen, value + 1. Dadurch fallen die Quotes weg und es kommt nicht mehr zum Schreiben von Werten außerhalb der dafür vorgesehenen Bereiche.

4 Fazit

Die Schachstelle ist unter den in dieser Arbeit betrachteten Bedingungen nicht kritisch und der Score (CVSS v3.0) von 9.8 CRITICAL [8] ist nicht gerechtfertigt, da die Schwachstelle zwar theoretisch vorhanden ist, sich aber praktisch nicht ausnutzen lässt. Dennoch ist es empfehlenswert, den Patch anzuwenden, da im Rahmen dieser Arbeit nur eine von vielen möglichen Konfiguration untersucht wurde. Es ist durchaus möglich, dass sich die Schwachstelle mit anderen Compiler-Optionen ausnutzen lässt und analog zu dem in Abschnitt 3.2.2 gezeigten Fall zu einem Absturz oder zu nicht definierten Verhalten führen kann.

Dieser Fall wäre besonders kritisch, da es sich um eine Bibliothek handelt, die in andere Projekte eingebunden werden kann. Das Ausnutzen dieser Schwachstelle könnte, sofern das möglich ist, zum Absturz von Software führen, die die Funktion SVGStartElement() dieser Bibliothek direkt oder indirekt aufruft.

Literatur

[1] Homepage von GraphicsMagick (http://www.graphicsmagick.org/)

[2] Gründe für das Forken von ImageMagick (https://marc.info/?l=imagemagick-developer&m=104777007831767&w=2)

[3] Repository von GraphicsMagick (http://hg.code.sf.net/p/graphicsmagick/code/)

[4] Downloadstatistik des GraphicsMagick Repositories nach Downladzahl, leicht modifiziert (https://sourceforge.net/projects/graphicsmagick/files/stats/timeline?dates=2003-02-07%20to%202019-05-27)

[5] Downloadstatistik des GraphicsMagick Repositories nach Ländern, leicht modifiziert (https://sourceforge.net/projects/graphicsmagick/files/stats/map?dates=2003-02-07%20to%202019-05-27)

[6] Sourceforge-Seite von GraphicsMagick (https://sourceforge.net/projects/graphicsmagick)

[7] Details zu CVE–2019–11005 (https://nvd.nist.gov/vuln/detail/CVE-2019-11005)

[8] Common Vulnerability Scoring System v3.0: Specification Document (https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf)

[9] Vorhandener Exploit (https://sourceforge.net/p/graphicsmagick/bugs/600/)

[10] PKGBUILD von GraphicsMagick unter Archlinux als Beispiel für nicht aktivierten AddressSanitizer (https://git.archlinux.org/svntogit/packages.git/tree/trunk/PKGBUILD?h=packages/graphicsmagick)

[11] ISO/IEC 9899:TC3 C-Standard (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

[12] Optimal Stack Slot Assignment in GCC (ftp://gcc.gnu.org/pub/gcc/summit/2003/Optimal%20Stack%20Slot%20Assignment.pdf)

[13] GCC Optimization Options (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)

[14] ON HOLY WARS AND A PLEA FOR PEACE (https://www.ietf.org/rfc/ien/ien137.txt)

[15] Das NUXI Problem (http://catb.org/jargon/html/N/NUXI-problem.html)